TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

D I P L O M A R B E I T

Synthetic Log Data Modeling for the Evaluation of Intrusion Detection Systems

Ausgeführt am Institut für
Wirtschaftsmathematik
der Technischen Universität Wien

unter der Anleitung von
Ao.Univ.Prof. Wolfgang Scherrer

durch

Markus Wurzenberger
Gußriegelstraße 15/7/23
1100 Wien

Wien, 10. Oktober 2015

Technische Universität Wien
A-1040 Wien . Karlsplatz 13 . Tel. +43-1-58801-0 . www.tuwien.ac.at

ERKLÄRUNG ZUR VERFASSUNG DER ARBEIT

Markus Wurzenberger
Gußriegelstrße 15/7/23
1100 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift)

KURZFASSUNG

Heutzutage sind Informations- und Kommunikationstechnik (IKT) Netzwerke ein fester Bestandteil unseres täglichen Lebens. Daher sind Intrusion Detection Systeme (IDS) ein essentieller Baustein in einer funktionierenden Sicherheitsinfrastruktur von heutigen Computernetzwerken. IDSs zielen darauf ab zeitnah Cyber-Attacken und unautorisierte Systemzugriffe zu erkennen. Während es dafür auf dem Markt viele Produkte gibt, die auf verschiedenen Ansätzen basieren, ist die Identifikation der effizientesten Lösung für eine spezifische Infrastruktur und die optimale Konfiguration dieser, immer noch ein ungelöstes Problem. Auf Grund des Fehlens von passenden Testumgebungen werden neue Ansätze zur Erstellung von Testdaten benötigt. Das Ziel dieser Arbeit ist es ein Modell zu definieren, das auf Log-Zeilen Clustering und Simulation von Markov Ketten basiert, um realistische synthetische Log Daten zu erstellen. Das präsentierte Modell benötigt nur ein kleines Set realer Netzwerkdaten als Input und erstellt ausgehend von dessen Eigenschaften eine vom Anwender vorgegene lange Folge von hochrealistischen synthetischen Log Daten. Um die Anwendbarkeit des in dieser Arbeit entwickelten Konzeptes zu zeigen, schließen wir die Arbeit mit einem illustrativen Beispiel zur Evaluierung und Test eines existierenden IDS unter Verwendung von generierten synthetischen Log Daten.

ABSTRACT

Today Information and Communications Technology (ICT) networks are a dominating component of our daily life. Hence Intrusion Detection Systems (IDSs) are an essential part of a working security-infrastructure of today's computer-networks. IDSs aim to timly detect cyber attacks and unauthorized system access. 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. Due to the lack of suitable test environments novel approaches for the generation of test data are required. The goal of this thesis is to define a model, based on log line clustering and Markov chain simulation, for generating realistic synthetic log data. The presented model requires only a small set of real network data as input and based on its characteristics generates a customer specified long sequence of highly realistic synthetic log data. To prove the applicability of the concept developed in this work, we conclude this thesis with an illustrative example of evaluation and test of an existing IDS by usage of generated synthetic log data.

DANKSAGUNG

An dieser Stelle möchte ich mich bei all jenen bedanken, die mich sowohl während meiner Diplomarbeit, als auch während meines gesamten Studiums unterstützt haben.

Einerseits möchte ich mich ganz besonders bei Herrn Professor Wolfgang Scherrer bedanken, der mir durch seine Betreuung und hilfreichen Ratschläge das Verfassen dieser Arbeit ermöglicht hat. Andererseits bin ich Florian Skopik zu großem Dank verpflichtet, da es mir durch ihn möglich war die vorliegende Arbeit im Zuge einer Anstellung am Austrian Institute of Technology zu verfassen. Neben Florian standen mir ganz besonders meine beiden Kollegen Giuseppe Settanni und Ivo Friedberg immer mit Rat und Tat zur Seite. Aber auch bei all meinen anderen Arbeitskolleginnen und Kollegen am AIT möchte ich mich für die schöne und spannende Zeit bedanken.

Natürlich dürfen auch meine Studienkolleginnen und Kollegen nicht unerwähnt bleiben. Durch sie und all meine anderen Freundinnen und Freunde wurden die letzten Jahre zu einer der schönsten meines Lebens.

Zu guter Letzt danke ich meiner Familie, ganz besonders meinen Eltern, ohne deren Unterstützung mein Studium nicht möglich gewesen wäre.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

1.1 Motivation

Today Information and Communication Technology (ICT) networks are the economic's vital backbone. While their complexity continuously evolves, cyber attacks become more sophisticated. Current attacks aim at stealing intellectual properties and sabotaging systems. These multistage tailored Advanced Persistent Threats (APT) usually result in financial loss and tarnished reputation. Already in 2011, the global cost caused by cybercrimes was estimated to 1 trillion dollars every year and it is increasing [1].

Recent cyber security reports [2, 3, 4] demonstrate that 77% of companies world-wide already have been affected by APTs, including cyber security firms, and the remaining 23% are unaware that they could be. On average it takes more than 1 year until an APT is discovered. Due to the fact that existing security mechanisms are hardly sufficient to counter APTs, novel approaches for Intrusion Detection Systems (IDS), which guarantee timely detection of invaders, are required. Numerous IDS solutions exist on the market, but no mechanisms which allow to rate, compare and evaluate them. Reasons for this are on the one hand the lack of data for testing, on the other hand the fact that security companies are not keen to share their secrets with potential competitors. In this thesis we propose a novel approach based on log line clustering and Markov chain simulation, for generating synthetic log data, to be applied in the test and evaluation of IDSs.

1.2 Problem Statement and Research Question

Because of the lack of data for testing IDSs it is not possible to sufficiently evaluate IDSs outside running production systems. Testing IDSs in running production networks, exposes the networks and leads to privacy issues. Moreover testing an IDS, by simulating an attack in the network is not feasible without affecting the regular functions of the network. Novel approaches for testing environments are therefore required.

In this thesis we describe a model for the generation of synthetic log data to be employed as test data for log-based IDSs. Furthermore an example of application is given by using the generated synthetic test data for evaluating AECID - Automated Event Correlation for Incident Detection - an IDS, designed and developed by the Austrian Institute of Technologies (AIT) [5].

The thesis addresses the following questions:

i Is it possible to define a meaningful model for generating realistic synthetic log data, based on the analysis of a small set of real log data?

ii Does the generated synthetic log file represent the same properties as the original file?

iii Can the generated synthetic log data be used for the evaluation of IDSs outside a running production environment?

iv Is it possible to obtain a synthetic log file simulating an ongoing cyber attack?

1.3 Methodology

In this thesis we apply methods of the probability theory and use statistical techniques to verify the results. The model we propose for generating synthetic log data is based on log line clustering and Markov chain simulation. Clustering allows us to determine network specific properties. Applying a Markov chain simulation enables the generation of sequences of log lines based on the extracted properties. Furthermore we establish novel statistical metrics to verify our model and to demonstrate the similarity between the generated log data and the real log data.

To confirm that the introduced model can be used to test and evaluate IDSs we describe an illustrative example of application. We apply AECID on the generated log data. First we show that AECID obtains similar results when it is executed with a real and a generated log file. Furthermore we discuss how the optimal configuration of AECID can be found by executing the algorithm on generated log data sets. Finally we describe the results obtained from the execution of AECID with a synthetic input log file including manually injected anomalies.

1.4 Thesis Outline

The remainder of the thesis is structured as follows: Chapter 2 gives an overview of the related work and the state-of-the-art. Chapter 3 summarizes the applied mathematical methods. Sections 3.1 and 3.2 deal with the computer simulation of Markov Chains and section 3.3 summarizes additional results from the probability theory. In Chapter 4 the model for generating log data is described. Sections 4.1-4.5 discuss the theoretical model, while section 4.6 evaluates the quality of the proposed model. Finally chapter 5 describes an illustrative example of an application for our approach. We discuss here how the generated log data can be used to test and evaluate IDSs. Chapter 6 concludes the thesis and gives an outlook about future work.

CHAPTER 2 RELATED WORK AND STATE-OF-THE-ART

Since the topic of this master thesis is not only related to the mathematical science, but also to Information and Communication Technology (ICT) Security, which is part of the computer science, the following chapter is split into two sections. The first one givitales a short introduction about ICT Security, in particular to the parts which are required for understanding this thesis. The second section summarizes the mathematical background.

2.1 Computer Scientifical Background

2.1.1 ICT Security

The research area of ICT Security deals with inventing security mechanisms to guarantee and optimize the proper function of information and communication technologies in computer networks. In the past few years ICT networks more and more become economics's vital backbone. Therefore, it is indispensable to put special emphasis on their safety and security. ICT Security is currently a very important topic, which also comes up more and more often in the daily news. The research area of ICT Security can be divided into several sub domains, like [6]:

- **Cyber Security & Cyber Situational Awareness**[7][8]: Information sharing about vulnerabilities in Connection Oriented Transport Services (COTS) and cyber security incidents, for raising cyber situational awareness and increasing resilience.
- **Cloud Security**[9]: Protection of data, which is not stored locally, but in a 'cloud' - distant server - and also protection, while executing programs, which are not installed on a local computer or server, but in a cloud.

- **Industrial Control System (ICS) Security and Smart Grid Security**[10]: Deals with the security of ICSs and smart grids. The term smart grid includes the communicative networking and controlling of electricity producer, storage, consumer and network equipment in energy transmission and distribution of electricity supply.

In these domains the following methods are used:

- **Risk Assessment**[11]: Application of risk assessment in the area of information technology.
- **Next Generation Cryptography Tools**[12]: Invention of novel concepts and approaches for information systems, which prevent from unauthorized reading and modifying private data.
- **Intrusion Detection Systems (IDSs)**[13] **and Anomaly Detection**[14]: Anomaly detection is a common procedure, for detecting irregularities. It is especially applied in IDSs.

The main topic of this master thesis is inventing highly realistic synthetic log data, for testing, evaluating and improving IDSs.

2.1.2 Logging

A lot of IDSs are operating on log data[15, 16]. Log data contains automatically generated protocols about all processes in a computer system. Therefore, one log line consists of a time stamp, which specifies the date (time) when a process takes/took place, and an event, which describes the process. The protocols stored in log files enable subsequently analysis of the actions in a system. This for example allows detecting erratic behavior. Basically, it is protocoled, who has done (or does) what, at which time on a system. For example, in log files of data bases every correctly completed transaction is stored. In case of a system crash, this allows restoring the previous state, before the system crashed.

Usually, by default log files are stored as simple text files. In opposite to a database format this has the advantage, that also in case the system crashes, it is easy to access the log data. Using a database format would raise the problem, that the log data is only reachable, if the data base, in which it is stored, is available. These days the log data management in large ICT networks is also easy. Solutions for log data management and security information event management are for example:

- Graylog[17],
- HP ArcSight Logger[18],
- AlienVault OSSIM[19].

A standard for transmitting log reports is syslog[20]. There exists a huge number of software components, based on different development environments, which can be used for writing and evaluating log data. Some examples are:

- Log4j 2[21] for Java[22],
- Enterprise Library[23] and .NET Logging Framework[24] for .NET Framework[25],
- Log4Delphi [26] for Delphi[27].

These software components are also called logger.

2.1.3 Intrusion Detection Systems

A decade ago the aim of cyber-criminal attacks against ICT networks was receiving recognition from like-minded people. In opposite to today's attacks the consequences have been very little, like downtimes and recovering and cleaning up compromised systems. Today's attacks target on stealing intellectual information and sabotaging systems.[28] These sophisticated and tailored attacks are called Advanced Persistent Threats (APT) and usually result in financial loss and tarnished reputation. Already in 2011, the global cost caused by cybercrimes was estimated to 1 trillion dollars every year and it is increasing.[1] Actual reports [2, 3, 4] point out another big problem - on average it takes more than 1 year until an APT is discovered. Furthermore, they show that 77% of companies already have been affected by APTs, also including cyber security firms and the remaining 23% are unaware that they could be.

The aim of IDSs is to detect invaders in ICT networks rapidly, so it is possible to react quickly and reduce the resulting damage. James Anderson has been in 1980 one of the first researchers, who pointed out the need of IDSs. [29] According to Anderson's work, software developers and administrators believe, that their systems are running in 'friendly' - save - environments. Accordingly, control mechanisms aim on preventing erratic behavior, because illegal access on such 'friendly' environments according to this paradigm only happens accidentally. Due to this fact it is easy for potential attackers, to undergo these control mechanisms and invade a computer network or system. This justifies using IDSs in addition to common security mechanisms. Even today, decades after Anderson published his work and the use of ICT networks strongly changed, the principle of IDSs is still valid. Moreover, it can be assumed, that the security of ICT networks has become worse, since their complexity and the networking of systems increased. This fact is also shown by the increasing severity of the consequences of todays cyber-criminal attacks.[30]

IDSs can roughly be classified as follows[31]:

- host-based IDS (HIDS),

- network-based IDS (NIDS),
- hybrid IDS.

Thereby, HIDSs are the oldest type. These kind of IDSs have to be installed on every monitored system, so called hosts. On the other hand NIDSs allow monitoring a whole network. Therefore, they also observe the network traffic between different systems. One advantage of HIDSs is, that they enable comprehensive monitoring of a single system and if the system got infiltrated, very specific statements about the attack can be made. But HIDSs get paralyzed if the monitored system is put out of action. In contrast, since NIDSs allow monitoring a whole network, if one system is put out of action, the monitoring of the rest of the network is not influenced by this issue. But in opposite to HIDSs, NIDSs do not allow comprehensive monitoring. One reason for this is that the bandwidth of the NIDSs could be overloaded. To make use of the advantages of both HIDSs and NIDSs and lower their disadvantages, usually hybrid approaches are applied. Therefore the data of both HIDSs and NIDSs is gathered in a central and closed management system.

2.1.4 Testing and Evaluation of ICT Security Mechanisms

One opportunity to test and configure security mechanisms like IDSs, is to include them into a running productive system. But this 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 the tested IDSs can detect an attack at all. Due to this fact, that 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 network base load. Therefore, highly realistic test environments are required, which allow to evaluate security mechanisms outside from running productive systems. Some examples for such test environments are:

- **Virtual Security Testbed (ViSe)**[32]: 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)**[33]: LARIAT was the first attempt, to invent a comprehensive test environment for IDSs.
- **Lincoln Laboratory Simulator (LLSIM)**[34]: LLSIM is a completely virtual further development of LARIAT, implemented in Java[22]. LLSIM offers a customizable test environment, in which hundreds of components of standard hardware can be simulated.

- **Testbed for Evaluating Intrusion Detection Systemns (TIDeS)**[35]: 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)**[36]: Among cyber-security scientists DETER is one of the leading test environments. It was invented with the collaboration of the National Science Foundation, the Department of Homeland Security, US, UC Berkely and McAffee Research.

The presented test environments all follow network centralized approaches. Therefore, they provide a good base for testing pure network traffic analysis mechanisms and for studying the behavior of a worm in a specific network. But most IDSs operate on a higher log-level-layer. Therefore, functionalities are needed, which also simulate human users' behavior. This is a major component in simulating the total liability of system. At the moment, mechanisms are missing which allow automatic evaluation of the most efficient configuration of IDSs. This makes an optimal usage of IDSs in specific network environments difficult.

CHAPTER 3 MATHEMATICAL FOUNDATIONS

The following chapter outlines the mathematical foundations, especially about Markov chains, which are required for the rest of the thesis. First, we summarize some important definitions and properties of Markov chains. Afterwards we deal with computer simulation of them. Since the Markov theory is a huge subject with a lot of application areas there exists a big mount of literature on it. The succeeding chapter is mainly based on [37, 38, 39, 40, 41].

3.1 Definitions and Properties of Markov Chains

A Markov chain, named after Andrey Markov[1], is a specific stochastic process-

Definition 3.1. Given a probability space $(\Omega, \mathcal{A}, \mathcal{P})$, a nonempty index set T and a measurable space $(S, \mathcal{S})$. A collection

$$\{X_t; t \in T\}$$

of S-valued random variables is named *stochastic process* with parameter area T and state space S.

For defining a Markov chain we also need the definition of conditional probability (cf. [42]).

Definition 3.2. Given a probability space $(\Omega, \mathcal{A}, \mathcal{P})$ and $A \in \mathcal{A}$. We define the *conditional probability* of a given A for any $B \in \mathcal{A}$

$$P(B|A) = \begin{cases} \frac{P(A \cap B)}{P(A)}, & \text{if } P(A) > 0, \\ 0, & \text{else.} \end{cases} \tag{3.1}$$

[1]Andrey Markov (1856-1922)

Now we can define a Markov chain as follows:

Definition 3.3. A stochastic process $\{X_t; t \in \mathbb{N}\}$ with countable state space S is called *Markov* chain, if it fulfills the *Markov property*, also known as the *memoryless property*: For all $n \in \mathbb{N}$, all $s_{i_0}, s_{i_1}, \ldots, s_{i_{n-1}} \in S$ and for all $s_i, s_j \in S$ with

$$P\left(X_0 = s_{i_0}, X_1 = s_{i_1}, \ldots, X_{n-1} = s_{i_{n-1}}, X_n = s_i\right) > 0$$

follows

$$\begin{aligned} &P\left(X_{n+1} = s_j | X_0 = s_{i_0}, X_1 = s_{i_1}, \ldots, X_{n-1} = s_{i_{n-1}}, X_n = s_i\right) \\ =&P\left(X_{n+1} = s_j | X_n = s_i\right). \end{aligned} \tag{3.2}$$

Markov chains, which are used for simulations are often characterized by the following property:

Definition 3.4. A Markov chain with countable state space S is named *homogeneous*, if the conditional distribution $P\left(X_{n+1} = s_j | X_n = s_i\right)$ is independent of n.

In the following we only consider Markov chains with a finite state space S with dimension $k \in \mathbb{N}$, where the state space S is given by the finite set $\{s_i\}_{i \in I}$ with $I = \{1, \ldots, k\}$. In the remaining chapter we only write S if no other sate space is considered.

Next we define stochastic matrices (cf. [42]):

Definition 3.5. A Matrix $P \in \mathbb{R}^{k \times k}$ is named *transition matrix* or *stochastic matrix*, if it satisfies

$$0 \leq p_{ij} \leq 1 \text{ for all } i, j \in \{1, \ldots k\} \tag{3.3}$$

and

$$\sum_{j=1}^{k} p_{ij} = 1 \text{ for all } i \in \{1, \ldots k\}. \tag{3.4}$$

Property 3.3 is needed, because probabilities are always nonnegative, and property 3.4 ensures that they sum to one. Hence, all p_{ij} satisfy $0 \leq p_{ij} \leq 1$ and each row of P represents a probability distribution.

The next results we receive from definitions 3.4 and 3.5. Homogeneous Markov chains satisfy for all $s_i, s_j \in S$ and all $n \in \mathbb{N}$

$$P\left(X_{n+1} = s_j | X_n = s_i\right) = P\left(X_1 = s_j | X_0 = s_i\right) =: p_{ij}.$$

In this context we call p_{ij} the *transition probability* and $P \in \mathbb{R}^{k \times k}$ (cf. equation ((3.5)), where k specifies the dimension of the state space, the *transition matrix* of the Markov chain.

$$P = \begin{pmatrix} p_{11} & \cdots & p_{1k+1} \\ \vdots & \ddots & \vdots \\ p_{k+11} & \cdots & p_{k+1k+1} \end{pmatrix} \tag{3.5}$$

Accordingly

$$p_{ij}^{(m)} := P(X_{n+m} = s_j | X_n = s_i) \tag{3.6}$$

defines the *m-step-transition probability* from state s_i to state s_j. That (3.6) is independent of n follows for the case $m = 1$ from Definition 3.4. The case $m \geq 2$ can be shown by induction and using the Chapman[2]-Kolmogorov[3] equation (cf. [43]) in the form as shown in (3.7).

$$p_{ij}^{(n+m)} = \sum_{s_l \in S} p_{il}^{(n)} p_{lj}^{(m)}. \tag{3.7}$$

The Chapman-Komogorov equation basically demonstrates that the probability that a Markov chain moves from state s_i to state s_j in $n+m$ steps is equal to the probability that it moves from state s_i to any state $s_l \in S$ in n steps and then it moves from state s_l in m steps to state s_j [41]. If $p_{ij}^{(0)} = \delta_{ij}$, where δ_{ij} denotes the Kronecker[4] delta, the equation also holds in the case $n = 0$ and $m = 0$.

In the following we only deal with homogeneous Markov chains. To simplify we won't mention this fact anymore.

Next we deal with the state a Markov chain starts with. Therefore we first define the initial distribution.

Definition 3.6. Given a Markov chain $\{X_t; t \in \mathbb{N}\}$ with countable state space $S = \{s_1, \dots, s_k\}$. The row vector given by

$$\begin{aligned} \mu^{(0)} &= (\mu_1^{(0)}, \mu_2^{(0)}, \dots \mu_k^{(0)}) \\ &= (P(X_0 = s_1), P(X_0 = s_2), \dots, P(X_0 = s_k)) \end{aligned}$$

is named *initial distribution* of the Markov chain.

Note, since $\mu^{(0)}$ represents a probability distribution, it has to satisfy conditions (3.8) and (3.9).

$$\mu_i^{(0)} \geq 0 \text{ for all } i = 1, \dots, k \tag{3.8}$$

[2]Sydney Chapman (1888-1970)
[3]Andrey Kolmogorov (1903-1987)
[4]Leopold Kronecker (1823-1891)

$$\sum_{i=1}^{k} \mu_i^{(0)} = 1. \tag{3.9}$$

According to Definition 3.6 $\mu^{(1)}, \mu^{(2)}, \ldots$ indicate the distribution of the considered Markov chain at the times $n = 1, 2, \ldots$ (cf. (3.10)).

$$\begin{aligned} \mu^{(n)} &= (\mu_1^{(n)}, \mu_2^{(n)}, \ldots \mu_k^{(n)}) \\ &= (P(X_n = s_1), P(X_n = s_2), \ldots, P(X_n = s_k)) \end{aligned} \tag{3.10}$$

The next result shows, that once one knows the initial distribution $\mu^{(0)}$ and the transition matrix P of a Markov chain, it is possible to calculate the distributions $\mu^{(n)}$ at any time $n = 1, 2, \ldots$.

Theorem 3.7. *Given a Markov chain $\{X_t; t \in \mathbb{N}\}$, with state space S, an initial distribution $\mu^{(0)}$ and a transition matrix P. For any $n \in \mathbb{N}$ we have*

$$\mu^{(n)} = \mu^{(0)} P^n, \tag{3.11}$$

where P^n denotes the n^{th} power of the transition matrix P.

To prove Theorem 3.7 we need the law of total probability (cf. [42]).

Theorem 3.8 (Law of Total Probability)**.** *Given a probability space $(\Omega, \mathcal{A}, \mathcal{P})$ and a sequence of pairwise disjoint sets $(B_i)_{i\in\mathbb{N}}$, with $B_i \in \Omega$, which satisfies $P\left(\biguplus_{i\in I} B_i\right) = 1$. Then for any $A \in \mathcal{A}$*

$$P(A) = \sum_{i\in I} P(A|B_i)P(B_i). \tag{3.12}$$

Proof. Because of the σ-additivity of P

$$P(A) = P\left(\biguplus_{i\in I}(A \cap B_i)\right) = \sum_{i\in I} P(A\cap B_i) \overset{(3.1)}{=} \sum_{i\in I} P(A|B_i)P(B_i).$$

[42] □

Now we can proof Theorem 3.7.

Proof of Theorem 3.7. To prove the theorem we use mathematical induction. First we deal with the case $n = 1$. With Theorem 3.8 we get for all $j = 1, \ldots, k$:

$$\begin{aligned} \mu_j^{(1)} &= P(X_1 = s_j) \\ &= \sum_{i=1}^{k} P(X_0 = s_i)P(X_1 = s_j|X_0 = s_i) \\ &= \sum_{i=1}^{k} \mu_i^{(0)} p_{ij} \\ &= (\mu^{(0)}P)_j, \end{aligned}$$

where $(\mu^{(0)}P)_j$ denotes the j-th element of the row vector $\mu^{(0)}P$. Thus, $\mu^{(1)} = \mu^{(0)}P$.

Now we use induction to prove the general case of (3.11). Fix any $m \in \mathbb{N}$ and suppose that (3.11) holds for $n = m$. Then we get for $n = m + 1$

$$\begin{aligned}\mu_j^{(m+1)} &= P(X_{m+1} = s_j)\\ &= \sum_{i=1}^{k} P(X_m = s_i)P(X_{m+1} = s_j|X_m = s_i)\\ &= \sum_{i=1}^{k} \mu_i^{(m)} p_{ij}\\ &= (\mu^{(m)}P)_j,\end{aligned}$$

so that $\mu^{(m+1)} = \mu^{(m)}P$. But by the induction hypothesis we get $\mu^{(m)} = \mu^{(0)}P^m$, which leads to

$$\mu^{(m+1)} = \mu(m)P = \mu^{(0)}P^mP = \mu^{(0)}P^{m+1}.$$

cf. [37] □

The remaining of the subsection deals with some major properties Markov chains can fulfill.

Definition 3.9. Given a Markov chain $\{X_t; t \in \mathbb{N}\}$ with state space S and transition matrix P. A state s_i *communicates* with state s_j, writing $s_i \rightarrow s_j$, if for at least one $m \in \mathbb{N}$ and one $n \in \mathbb{N}$, with $n > m$ holds:

$$P(X_{m+n} = s_j|X_n = s_i) > 0.$$

In other words, state s_i communicates with state s_j, if the n-step-transition probability, cf. (3.6), from state s_i to state s_j is strict positive for any $n \in \mathbb{N}$.

Sate s_i and s_j *intercommunicate*, writing $s_i \leftrightarrow s_j$, if $s_i \rightarrow s_j$ and $s_i \rightarrow s_j$.

Definition 3.9 leads directly to the definition of irreducibly.

Definition 3.10. A Markov chain $\{X_t; t \in \mathbb{N}\}$ with state space S and transition matrix P is said to be *irreducible*, if for all states $s_i, s_j \in S$, $s_i \leftrightarrow s_j$ holds, i.e. if for all states s_i and s_j there is a $n \in \mathbb{N}$ with $p_{ij}^{(n)} > 0$. Otherwise the Markov Chain is *reducible*.

If a Markov chain $\{X_t; t \in \mathbb{N}\}$ with state space S and transition matrix P is irreducible can be tested by calculating matrix A (cf. equation (3.13)) [44].

$$A = I + P + P^2 + \ldots + P^k \tag{3.13}$$

The Markov chain is irreducible if for all elements of A it is valid $a_{ij} = 0$, for all $i.j \in \{1, \ldots, k\}$. This is valid, because according to definition 3.10 $s_i \leftrightarrow s_j$

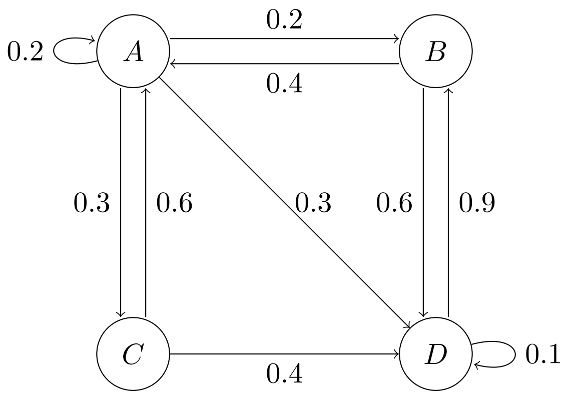

Figure 3.1: Graph of an irreducible Markov chain $\{X_t; t \in \mathbb{N}\}$ with state space $S = \{A, B, C, D\}$ and transition matrix P (cf. (3.14)).

is equal to there exists a $n \in \mathbb{N}$, with $p_{ij}^{(n)} > 0$. If such an n exists, there exists also a $n \in \mathbb{N}$, with $0 \leq n \leq k$ that fulfills $p_{ij}^{(n)} > 0$.

For a better understanding, example 3.11 shows an irreducible Markov chain and Example 3.12 shows a reducible Markov chain.

Example 3.11 (Irreducible Markov chain)**.** Consider a Markov chain $\{X_t; t \in \mathbb{N}\}$ with state space $S = \{A, B, C, D\}$ and transition matrix

$$P = \begin{pmatrix} 0.2 & 0.2 & 0.3 & 0.3 \\ 0.4 & 0 & 0 & 0.6 \\ 0.6 & 0 & 0 & 0.4 \\ 0 & 0.9 & 0 & 0.1 \end{pmatrix}. \tag{3.14}$$

The graph of the considered Markov chain in Figure 3.1 shows, that the Markov chain with transition matrix P is irreducible, since from each state $s_i \in D$, every state $s_j \in S$, also s_i itself, is reachable.

Example 3.12 (Reducible Markov chain)**.** Consider a Markov chain $\{X_t; t \in \mathbb{N}\}$ with state space $S = \{A, B, C, D\}$ and transition matrix

$$P = \begin{pmatrix} 0.2 & 0.8 & 0 & 0 \\ 0.5 & 0.5 & 0 & 0 \\ 0 & 0 & 0.3 & 0.7 \\ 0 & 0 & 0.6 & 0.4 \end{pmatrix}. \tag{3.15}$$

The graph of the considered Markov chain in Figure 3.2 shows, that the Markov chain with transition matrix P is reducible. If the chain starts in state A or B, the states C and D are unreachable. Vice versa, if the chain starts in state C or D, the states A and B are unreachable.

Furthermore, if the Markov chain starts in state A or B, the chain behaves like a Markov chain with state space $S = \{A, B\}$ and transition matrix

$$P = \begin{pmatrix} 0.2 & 0.8 \\ 0.5 & 0.5 \end{pmatrix}.$$

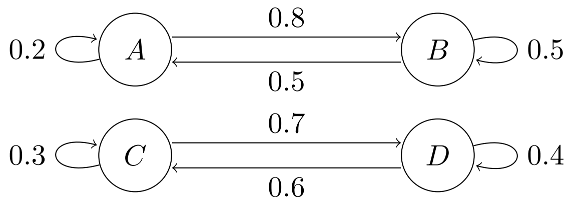

Figure 3.2: Graph of a reducible Markov chain $\{X_t; t \in \mathbb{N}\}$ with state space $S = \{A, B, C, D\}$ and transition matrix P (cf. (3.15)).

If it starts in state C or D, the chain behaves like a Markov chain with state space $S = \{C, D\}$ and transition matrix

$$P = \begin{pmatrix} 0.3 & 0.7 \\ 0.6 & 0.4 \end{pmatrix}.$$

This fact shows, that analyzing a reducible Markov chain can be reduced to the analysis of one or more irreducible Markov chains with state spaces, which are subsets of the state space of the original reducible Markov chain.

3.2 Computer Simulation of Markov Chains

In the following subsection we describe a method, how to perform computer simulation of a given Markov chain $\{X_t; t \in \mathbb{N}\}$, with sate space S, given initial distribution $\mu^{(0)}$ and transition matrix P. The remaining section is mainly based on [37].

To perform the computer simulation of a 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.

In the remaining section we show that then the initial value of the Markov chain can be calculated with $X_0 = \psi(U_0)$ and the values for $n \in \mathbb{N}\backslash\{0\}$ with $X_n = \phi(X_{n-1}, U_n)$.

First, we indicate how to generate the starting value X_0 of a considered Markov chain $\{X_t; t \in \mathbb{N}\}$. Therefore, we start with the construction of an initial function ψ.

Definition 3.13. The *initiation function* is a function $\psi : [0, 1] \to S$, which we use to generate the starting value X_0. We assume ψ fulfills:

(i) ψ is piecewise constant, i.e. $[0, 1]$ can be split into a finite number of subintervals, so that ψ is constant on each interval.

(ii) 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

$$\int_0^1 \mathbb{1}_{\{s\}}\left(\psi\left(x\right)\right) dx = \mu^{(0)}\left(s\right) \qquad \text{for all } s \in S. \tag{3.16}$$

Note, $\mathbb{1}_{\{s\}}(x)$ defines the so-called indicator function:

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

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 $s \in S$ (3.17) is valid.

$$P(x_0 = s) = P(\psi(U_0) = s) = \int_0^1 \mathbb{1}_{\{s\}}(\psi(x))dx \overset{(3.16)}{=} \mu^{(0)}(s). \tag{3.17}$$

Definition 3.14. We call an initiation function ψ *valid* for the Markov chain $\{X_t; t \in \mathbb{N}\}$, if equation (3.16) holds for all $s \in S$.

Now we can construct a valid initiation function. Therefore, let S be the state space and $\mu^{(0)}$ the initial distribution of a Markov chain, which is considered for simulation. We define:

$$\psi(x) = \begin{cases} s_1 & \text{for } x \in \left[0, \mu^{(0)}(s_1)\right) \\ s_2 & \text{for } x \in \left[\mu^{(0)}(s_1), \mu^{(0)}(s_1) + \mu^{(0)}(s_2)\right) \\ \vdots & \\ s_i & \text{for } x \in \left[\sum_{j=1}^{i-1} \mu^{(0)}(s_j), \sum_{j=1}^{i} \mu^{(0)}(s_j)\right) \\ \vdots & \\ s_k & \text{for } x \in \left[\sum_{j=1}^{k-1} \mu^{(0)}(s_j), 1\right]. \end{cases} \tag{3.18}$$

The next result verifies, that (3.18) satisfies the properties (i) and (ii) from Definition 3.13.

Lemma 3.15. *The function ψ, defined in (3.18), represents a valid initiation function for the Markov chain $\{X_t; t \in \mathbb{N}\}$.*

Proof. We have to prove, that ψ satisfies the properties (i) and (ii) from Definition 3.13. Since, (3.18) defines a piecewise constant function, property (i) is obvious. To prove, that ψ satisfies property (ii), we have to check that (3.16) holds:

$$\int_0^1 \mathbb{1}_{\{s_i\}}(\psi(x)) = \sum_{j=1}^{i} \mu^{(0)}(s_j) - \sum_{j=1}^{i-1} \mu^{(0)}(s_j) = \mu^{(0)}(s_i) \qquad \text{for i=1,...,k.}$$

Consequently ψ defines a valid initiation function. □

Finally, we can generate a starting value X_0 using the initiation function defined in (3.18) and the first random value U_0. Next, we describe how to generate X_{n+1} from X_n for any $n \in \mathbb{N}$. Then this procedure can be applied iteratively to simulate the Markov chain $\{X_t; t \in \mathbb{N}\}$. Therefore, we define an update function.

Definition 3.16. The *update function* ϕ is a function $\phi : S \times [0,1] \to S$, which we use to generate the value X_{n+1} from X_n and U_{n+1} for any $n \in \mathbb{N}$. We assume ϕ fulfills:

(i) for fixed $s_i \in S$, the function $\phi(s_i, x)$ is piecewise constant (when ϕ is considered as function of x) and

(ii) 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:

$$\int_0^1 \mathbb{1}_{\{s_j\}}(\phi(s_i, x))dx = p_{ij} \qquad \text{for all } s_i, s_j \in S. \tag{3.19}$$

If ϕ (denote $X_{n+1} = \phi(X_n, U_{n+1})$ satisfies (3.19), then:

$$\begin{aligned} P(X_{n+1} = s_j | X_n = s_i) &= P(\phi(s_i, U_{n+1}) = s_j | X_n = s_i) \\ &= P(\phi(s_i, U_{n+1}) = s_j) \\ &= \int_0^1 \mathbb{1}_{\{s_j\}}(\phi(s_i, x))dx \overset{(3.19)}{=} p_{ij}. \end{aligned} \tag{3.20}$$

In (3.20) $P(\phi(s_i, U_{n+1}) = s_j | X_n = s_i) = P(\phi(s_i, U_{n+1} = s_j))$ is valid, because U_{n+1} is independent of $(U_1, \ldots, U_n)$, and thus also of X_n. Due to the same argument the probability remains the same if we condition with the values $(X_0, \ldots, X_{n-1})$. Hence, the described procedure, using an update function as characterized in Definition 3.16, provides a correct simulation of the Markov chain.

Definition 3.17. We call an update function ϕ *valid* for the Markov chain $\{X_t; t \in \mathbb{N}\}$, if equation (3.19) holds for all $s_i, s_j \in S$.

Now, we can construct a valid update function similarly to a valid initiation function. We define for each $s_i \in S$:

$$\phi(s_i, x) = \begin{cases} s_1 & \text{for } x \in [0, p_{i1}) \\ s_2 & \text{for } x \in [p_{i1}, p_{i1} + p_{i2})) \\ \vdots & \\ s_j & \text{for } x \in \left[\sum_{l=1}^{j-1} p_{il}, \sum_{l=1}^{j} p_{il}\right) \\ \vdots & \\ s_k & \text{for } x \in \left[\sum_{l=1}^{k-1} p_{il}, 1\right]. \end{cases} \tag{3.21}$$

The next result verifies, that (3.21) satisfies the properties (i) and (ii) from Definition 3.16.

Lemma 3.18. *The function ϕ, defined in (3.21), represents a valid update function for the Markov chain $\{X_t; t \in \mathbb{N}\}$.*

Proof. We have to prove, that ϕ satisfies the properties (i) and (ii) from Definition (3.16). Since, (3.21) defines a piecewise constant function, property (i) is obvious. To prove, that ϕ satisfies propert (ii), we have to check that (3.19) holds:

$$\int_0^1 \mathbb{1}_{\{s_j\}}(\phi(s_i, x))dx = \sum_{l=1}^{j} p_{il} - \sum_{l=1}^{j-1} p_{il} = p_{ij}, \qquad \text{for all } s_i, s_j \in S.$$

Consequently ϕ defines a valid update function. □

Finally, we have found a procedure, which allows simulation of a homogeneous Markov chain $\{X_t; t \in \mathbb{N}\}$, with state space S, initial distribution $\mu^{(0)}$ and transition matrix P. Therefore, first a valid initiation function ψ and a valid update function ϕ are constructed (for instance as in (3.18) and (3.21)). Then, using a sequence $\{U_t; t \in \mathbb{N}\}$ of independent and identical distributed random numbers, uniformly distributed in the unit interval $[0, 1]$, we set:

$$\begin{aligned} X_0 &= \psi(U_0) \\ X_1 &= \phi(X_0, U_1) \\ X_2 &= \phi(X_1, U_2) \\ X_3 &= \phi(X_2, U_3) \\ &\vdots \end{aligned}$$

We finish the section with a simple example:

Example 3.19 (Weather)**.** In the following example we assume, that the weather tomorrow only depends on today's weather. Under this condition, the weather forecast can be subscribed by a Markov chain. To simplify, we assume that there are only two kinds of weather: sunshine and rain. Therefore we consider a Markov chain with state space $S = \{s_1 = \textit{sunshine}, s_2 = \textit{rain}\}$ and transition matrix

$$P = \begin{pmatrix} 0.75 & 0.25 \\ 0.25 & 0.75 \end{pmatrix}.$$

We assume, that the considered Markov chain starts on a rainy day. Hence, we get an initiation distribution $\mu^{(0)} = (0, 1)$. To simulate the Markov chain with the previously proposed procedure, we have to construct an initiation function and an update function. By (3.18) we get the initiation function

$$\psi(x) = s_1 \qquad \text{for all } x,$$

and by (3.21) we get the update function given by

$$\phi(s_1, x) = \begin{cases} s_1 & \text{for } x \in [0, 0.75) \\ s_2 & \text{for } x \in [0.75, 1] \end{cases}$$

and

$$\phi(s_2, x) = \begin{cases} s_1 & \text{for } x \in [0, 0.25) \\ s_2 & \text{for } x \in [0.25, 1]. \end{cases}$$

3.3 Additional Results from the Probability Theory

The following section summarizes some results from the probability theory, which will be used later. First we recall the *binomial distribution* and the *bernoulli*[5] *distribution* [45].

Definition 3.20 (binomial distribution)**.** The discrete probability distribution with the probability mass function

$$B(k|p,n) = \binom{n}{k} p^k (1-p)^{n-k}, \text{ with } k = 0, 1, \dots n \tag{3.22}$$

is named binomial distribution with the parameters $n \in \mathbb{N}$ - number of trials - and $p \in [0, 1]$ - success probability in each trial.

Definition 3.21 (bernoulli distribution)**.** The bernoulli distribution is the probability distribution of a random variable that takes the value 1 with the success probability $p \in [0, 1]$ and the value 0 with the failure probability $q = 1 - p$.

Remark 3.22. The bernoulli distribution is the special case of the binomial distribution with $n = 1$.

Next we define a *bernoulli process* [46].

Definition 3.23 (bernoulli process)**.** A bernoulli process is a time discrete stochastic process, which consists of a finite or infinite sequence of independent bernoulli distributed (with parameter $p \in [0, 1]$) random variables.

Furthermore we need *binomial tests* [47].

Definition 3.24 (binomial test)**.** A binomial test is a statistical hypothesis test, where the test statistic X is binomially distributed. There are three types of hypothesis which can be tested for the unknown probability p of a characteristic: two-sided, right-sided, left-sided. See also table 3.1.

[5] Jacob Bernoulli (1654-1705)

Test Type	H_0	H_1
two-sided	$p = p_0$	$p \neq p_0$
right-sided	$p = p_0$ or $p \leq p_0$	$p > p_0$
left-sided	$p = p_0$ or $p \geq p_0$	$p < p_0$

Table 3.1: Types of hypothesis

The test statistic X specifies how often the characteristic occurred in a random sampling of size $n \in \mathbb{N}$. Under the null hypothesis $H_0 : p = p_0$ the test statistic is binomially distributed with $B(p_0, n)$ (cf. (3.23)).

$$P(X = i) = B(i|p_0, n) = \binom{n}{i} p_0^i (1 - p_0)^{n-i} \tag{3.23}$$

In the following we only focus on left-sided binomial tests. We specify a significance-level $\alpha \in [0, 1]$. Then the critical value $c \in \mathbb{N}$, with $c < n$, is the smallest value, for which equation (3.24) is respected.

$$\sum_{i=0}^{c} B(i|p_0, n) \geq \alpha \tag{3.24}$$

This means that if the tested characteristic occurs at least c times in the sample of size n the null hypothesis H_0 is accepted. Otherwise it is rejected.

CHAPTER 4

MODEL FOR GENERATING SYNTHETIC LOG DATA

In the following chapter we define the theoretical model on which our novel approach for generating synthetic log data is based. Furthermore an evaluation of the model is done.

4.1 Theoretical Model

The following section describes the theoretical model behind our novel approach for generating highly realistic synthetic log data. Figure 4.1 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 system which is logged, a part of real log data is required. Analysis of this log data is done to identify characterizing properties. Based on the so generated data a model is build, which then can be used for generating realistic synthetic log data. Furthermore, iterative and interactive refinement of the analysis and the model allows modifying the complexity of the generated log data. This means that test data for quick to in-depth analysis and evaluation of different software applications, especial IDSs, 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 figure 4.1 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,

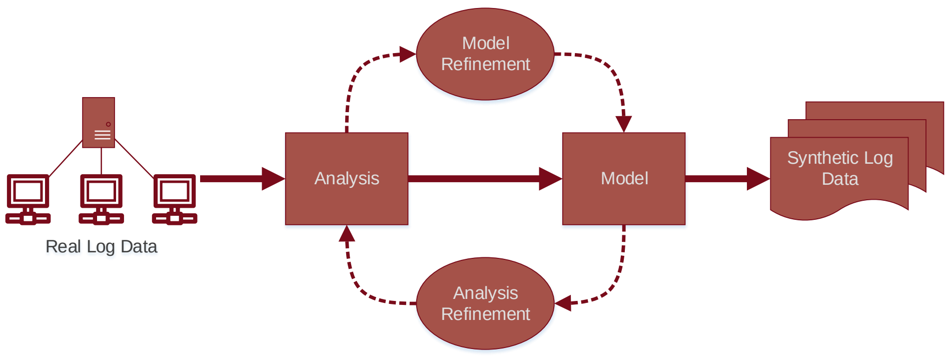

Figure 4.1: The concept of the proposed approach for building a log data model.

(iv) populating log lines.

During step (i) clusters are build 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 generated synthetic log 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.

4.2 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 [48]. The algorithm has been especially developed for detecting word clusters in log files [49]. Furthermore, there already exists an C [50] implementation - Simple Logfile Clustering Tool (SLCT) [51] - of the algorithm, which is open source and easy to adapt for our needs. The remaining section is partly based on [48, 52].

4.2.1 Foundations and Comparison to other Algorithms

In past decades a lot of research about cluster analysis has been done and various clustering algorithms have been invented [53, 54]. According to [55] a cluster analysis can be defined as follows:

The aim of a cluster analysis is to divide a set of $n \in \mathbb{N}$ objects into $m \in \mathbb{N}$

classes/cluster $C_1, \ldots, C_m$, which is called a classification (cf. equation (4.1)).

$$\mathcal{C} = \{C_1, \ldots, C_m\} \tag{4.1}$$

Furthermore every cluster includes at least one element and at most all n objects.

The classification should guarantee that all objects within one cluster are close/similar to each other. To determine which objects are close a distance function $d(x, y)$ is employed. Considering a set of points with $k \in \mathbb{N}$ attributes in the data space $\mathbb{R}^k$, to determine if two points $x, y \in \mathbb{R}^k$ are close, a distance function $d(x, y)$ can be employed. Many clustering algorithms use variants of L_p norms ($p = 1, 2, \ldots$) as distance function (cf. equation (4.2)).

$$d_p(x, y) = \sqrt[p]{\sum_{i=1}^{k} |x_i - y_i|^p} \tag{4.2}$$

Traditional clustering methods work well for data with numerical attributes in low-dimensional data spaces, with k below 10. Next we address two main problems, which occur while clustering log lines.

Firstly, we consider the log line `Connect to 192.168.1.1`. This log line can be represented by by the point $(Connect, to, 192.168.1.1)$ in a three-dimensional data space. But since the attributes of this data point are of categorical nature the distance function from equation (4.2) cannot be used for determining the similarity of the example log line to an other one [56]. Meanwhile various distance function, also for categorical attributes, exist. Some like the Jaccard coefficient are defined in [53, 57]. The Jaccard[1] coefficient $J(A, B)$ of two sets A, B (cf. equation (4.3)) is defined as the ratio between the size of the intersection and the size of the union of both sets. The Jaccard coefficient ranges between 0 and 1, whereby 1 means $A = B$ and 0 means that A and B are completely different.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{4.3}$$

Secondly, 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. Traditional clustering algorithms usually does not work well for high-dimensional data with $k > 15$ [53, 58]. With increasing dimensions every pair of points seems far away, which makes it impossible to detect any clusters. This problem occurs, because traditional clustering methods do not detect clusters, which are existing in subspaces of the original data space [58]. For example the points $(1, 45, 133, 177578, 2)$ and $(534, 51, 129, 3, 2134)$ do not form a cluster

[1]Paul Jaccard (1868-1944)

in the original data space, but in the subspace of the second and the third dimension they are very close.

There exist various algorithms for clustering high-dimensional data with categorical attributes. Examples are CLIQUE, MAFIA, PROCLUS [59] and CACTUS [60], which all try to avoid the above mentioned problems, which can occur using traditional clustering methods.

Both, the CLIQUE [58] and the MAFIA [61] are bottom-up algorithms. This means they make use of the monotonicity of the clustering criterion regarding to dimensionality to lower the search space. See also definition 4.1 [58].

Definition 4.1 (Monotonicity)**.** If a collection of points S is a cluster in a k-dimensional space, then S is also part of a cluster in any $(k-1)$-dimensional projections of this space.

Therefore the algorithms work as follows: Starting with identifying clusters in 1-dimensional subspaces and after identifying clusters $C_1, \dots, C_m$ in $(k-1)$-dimensional subspaces, from $C_1, \dots, C_m$ cluster candidates for the k-dimensional space are formed. These algorithms follow density based approaches for clustering (an example for a density based clustering algorithm is given in section 4.2.2), which means no distances between data points are measured. Instead dense regions are identified in the data space. Based on this dense regions clusters are formed. Therefore, these algorithms are very effective in discovering clusters in subspaces. The disadvantage of these algorithms is their low performance, since for producing a frequent m-itemset they first have to produce $2^m - 2$ subsets of this m-itemset. The MAFIA algorithm is an extension of CLIQUE, which improves the scalability and the efficiency.

In opposite to CLIQUE and MAFIA, PROCLUS [62] implements a top-down subspace clustering approach [59]. First an initial approximation of the clusters in the full data space is built. Whereby all dimensions are equally weighted. Then a weight is assigned to each dimension for each cluster. The new weights are used for regenerating clusters in the next iteration. The PROCLUS algorithm is based on the K-medoid method. This means K clusters in subspaces of the original data space are built. Here the problem is that it is impossible to predict precisely the number of clusters for log file data before the cluster analysis.

The CACTUS [60] algorithm is based on summarization. The CACTUS algorithm passes over the data set only twice and therefore is very fast. In the first step a data summary is built and in the second step, based on the data summary cluster candidates are generated. Finally the set of actual clusters is determined. But the algorithm tends to be chaining, which means that long strings of points are assigned to the same cluster [63].

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.

According to [48] log data has two characterizing properties. First, most of the occurring words in a log file are rather infrequent. Many of them even appear only once. Only a small part of the set of words, which occur in a log file can be considered as frequent. This means they appear at least once per every 1000 or 10000 lines. A similar phenomenon has been shown for World Wide Web data in [64]. Second, there are strong correlations between words, which are considered as frequent. Log lines often follow patterns, which consist of fixed and variable parts, e.g.,

`connection from` *%a* `port` *%b*,

where `connection`, `from` and `port` are the fixed parts. This words also would be frequents words in a log file. The placeholder *%a* replaces an IP-adress and *%b* replaces a port number. This variable parts would be infrequent words in a log file and the replaced words would only occur once or just a few times. This also shows that the position of a word in a log line matters. Therefore, `connection` would have a different meaning, if it does not occur at the first position of a log line.

While most of the other clustering algorithms for high-dimensional data are suitable for generic data SLCT also takes the above mentioned properties into account. Furthermore, SLCT implements a density based clustering approach, which tries to detect clusters in subspaces of the original data space. Similar to CACTUS, SLCT performs three steps. During a first parse over the data set, a data summary is built. Afterwards, while passing over the data set again cluster candidates are built. Finally, the clusters are selected from the list of candidates.

4.2.2 Defining the Log Line Clustering Algorithm

According to Vaarandi, every *data point* P in the *data space* D corresponds to one log line in a log file. The *dimension* $n \in \mathbb{N}$ 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 P has *categorical attributes* $i_1, \ldots, i_n$. Categorical attributes are equal to the words of the log line corresponding to the *data point* P. The *values* $v_1, \ldots, v_n$ of $i_1, \ldots, i_n$ 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 $i_1, \ldots, i_n$ are equal to the position of the word they correspond to, i.e. $i_1 = 1, i_2 = 2, \ldots, i_n = n$. As a result one data point $P \in D$ is a vector of strings of the form as shown in equation (4.4).

$$P = (x_1 = v_1, \ldots, x_n = v_n) \tag{4.4}$$

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 $l < k \leq n$ are set to null, i.e. they are empty.

Furthermore, let $J \subseteq \{1, 2, \ldots n\}$ be a subset of indices, a *region* S is defined as a subset of D ($S \subseteq D$), where all data points $P \in S$ have the same values v_j for all $j \in J$, c.f. equation (4.5).

$$S = \{P \in D | x_j = v_j, \forall j \in J\} \tag{4.5}$$

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

The clustering algorithm can be structured into three steps:

1. data summarization,
2. cluster candidates building,
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 $\{(1, \text{‘example’})\}$ and $\{(3, \text{‘example’})\}$ not necessarily contain the same data points (log lines) and the word ‘example’ may have different meanings on different positions. Since frequent words correspond to dense 1-region, 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, $J \subseteq \{1, 2, \ldots n\}$ is a subset of indices with cardinality $|J| = m$ and $m \in \{1, 2, \ldots, n\}$. If the processed log line can be assigned to m dense 1-regions with the m fixed attributes (i_j, v_j), $j \in J$, the generated cluster candidate is a region S with the set of fixed attributes $S_{fixedAttributes} = \{(i_j, v_j) | j \in J\}$. 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 $(1, \text{‘Connection’})$ and an other one with the fixed attribute $(2, \text{‘from’})$, the generated cluster candidate is the region S, with the set of fixed attributes $S_{fixedAttributes} = \{(1, \text{‘Connection’}), (2, \text{‘from’})\}$. Note, that at most one cluster candidate per line can be generated. Therefore, the support value does not specifies 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 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 candidate, ‘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 $n = 4$. 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 S 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 4.3.

To perform the proposed log line clustering, we adapted SLCT [51], an already existing C [50] 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 specifies, from how many log lines a cluster candidate has to be generated, so that it becomes a cluster in the end. The support threshold value N 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 [48, 52, 65]. 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 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*, *clusterDescriptions*`)` 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.

```
   Data: $logFile$, $supportThresholdValue$
   Result: $clusterDescriptions$
1  $clusterDescriptions \leftarrow$ runSLCT($logFile, supportThresholdValue$)
2  $outlier$ = storeOutliers($logFile, clusterDescriptions$)
3  while $length(Outlier) >= supportThresholdValue$ && $lengthIncreased(clusterDescriptions)$ do
4      $clusterDescriptions \leftarrow$ runSLCT($outlier, supportThresholdValue$)
5      $outlier$ = storeOutliers($outlier, clusterDescriptions$)
6  end
```

Algorithm 1: Creation of the cluster descriptions.

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 one more time 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.

4.3 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 4.4.

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* [66] with the same properties of a *dendrogram* [67], 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. equation(4.6)) for every cluster C_i, with $i \in \{1, \dots, n\}$, where $n \in \mathbb{N}$ is the number of generated clusters. The cluster value cv_i is defined as the number of fixed attributes a cluster consists of.

$$cv = (cv_1, \dots, cv_n) \tag{4.6}$$

For example the cv of the cluster `Connect from * port *` is 3, because it consists of the fixed attributes $\{(1, \text{`Connection'}), (2, \text{`from'}), (4, \text{`to'})\}$. All clusters a log line matches, 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 bases 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 is also no problem 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 3 characterizes, how the most accurate cluster is chosen. The function `getClusterByValue(`$cluster, clusterValue$`)` returns the cluster, corresponding to the previously chosen cluster value.

4.4 Arranging Clusters in the Generated Log File

The following step in the proposed model is arranging the clusters in the generated log file. We apply a Markov chain approach (cf. section 3.1), 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 log file by simulating a homogeneous Markov chain $\{X_t; t \in \mathbb{N}\}$, with state space S, transition matrix P and initial distribution $\mu^{(0)}$ by applying the approach

```
   Data: $logFile, clusterDescriptions$
   Result: $clusters$
1  for $1 \leq i \leq$ length(clusterDescriptions) do
2      $clusterValue_i$ = calculateClusterValue($clusterDescription_i$)
3  end
4  for $1 \leq i \leq$ length(logFile) do
5      for $1 \leq j \leq$ length(clusterDescriptions) do
6          if matches($logFile_i$,$clusterDescriptions_j$) then
7              $matchingClusters \leftarrow clusters_j$
8          end
9      end
10     $mostAccurateCluster$ =
       findMostAccurateCluster($matchingClusters$)
11     $mostAccurateCluster \leftarrow logFile_i$
12 end
```

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

presented in section 3.2.

First, we calculate the transition matrix P. In section 4.3 we already mentioned that we need distinct clusters. We achieve this by choosing the most accurate cluster. Hence, it is possible to estimate the probability at which one cluster follows another one. The number of transitions t_{ij} from cluster C_i to cluster C_j with $i, j \in \{1, \dots, n+1\}$ is stored in a matrix $T \in \mathbb{N}^{(n+1)\times(n+1)}$ (cf. equation (4.7)), which was defined in definition 3.5.

$$T = \begin{pmatrix} t_{11} & \cdots & t_{1n+1} \\ \vdots & \ddots & \vdots \\ t_{n+11} & \cdots & t_{n+1n+1} \end{pmatrix} \tag{4.7}$$

The last index here is $n+1$ (instead of n, the number of clusters), because the outliers are considered as an extra cluster. Calculating the matrix T can be done, while choosing the most accurate clusters.

The transition probabilities p_{ij} from cluster C_i to cluster C_j can be calculated, as shown in equation (4.8).

$$p_{ij} = \frac{t_{ij}}{\sum_{k=1}^{n+1} t_{ik}}, \qquad \text{for all } i, j \in \{1, \dots, n+1\} \tag{4.8}$$

The transition probabilities are then stored in a transition matrix $P \in \mathbb{R}^{(n+1)\times(n+1)}$ (cf. equation 4.9).

$$P = \begin{pmatrix} p_{11} & \cdots & p_{1n+1} \\ \vdots & \ddots & \vdots \\ p_{n+11} & \cdots & p_{n+1n+1} \end{pmatrix} \tag{4.9}$$

```
   Data: $matchingClusters, clusterValuesOfMatchingClusters$
   Result: $mostAccurateCluster$
1  $sortedValues$=sort($clusterValuesOfMatchingClusters$)
2  $index$=length($sortedValues$)
3  while $occurence(sortedValues_{index})! = 1$ do
4      $index$-=1
5      if $index < 0$ then
6          considered line is an outlier
7          return
8      end
9  end
10 $mostAccurateCluster$ =
   getClusterByValue($matchingClusters$,$sortedValues_{index}$)
```

Algorithm 3: Deciding which of the clusters a log line matches is the most accurate cluster.

In the next step we estimate the initial distribution $\mu^{(0)} \in \mathbb{R}^{n+1}$, which was defined in definition 3.6. 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 equation (4.10).

$$\mu_i^{(0)} = \frac{\sum_{l=1}^{n+1} t_{il}}{\sum_{k=1}^{n+1} \sum_{l=1}^{n+1} t_{kl}} \qquad \text{for all } i \in \{1, \ldots, n+1\} \tag{4.10}$$

Finally the clusters in the generated log file can be arranged by simulating the Markov chain $\{X_t; t \in \mathbb{N}\}$, with the state space $S = \{C_1, \ldots, C_{n+1}\}$, transition matrix P and initial distribution $\mu^{(0)}$. For simulating the Markov chain to arrange the clusters in the generated log file, we apply the approach proposed in section 3.2. Remember, 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 functuion ψ and
- an update funtion ϕ.

The initiation function $\psi : [0, 1] \rightarrow S$ was defined in definition 3.13. We use it to generate the starting value X_0. The update function $\phi : S \times [0, 1] \rightarrow S$ was defined in 3.16. We use it to generate the value X_{n+1} from X_n and U_{n+1} for any $n \in \mathbb{N}^{>0}$.

Eventually we get a procedure which allows simulation of the homogeneous Markov chain $\{X_t; t \in \mathbb{N}\}$, with state space $S = \{C_1, \ldots, C_{n+1}\}$, 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 in the unit interval $[0, 1]$, we obtain equation (4.11). The number of values to be generated can be specified by the user.

$$\begin{aligned} X_0 &= \psi(U_0) \\ X_i &= \phi(X_{i-1}, U_i) \qquad i \in \mathbb{N} \setminus \{0\} \end{aligned} \tag{4.11}$$

Assuming that the input log file models a irreducible Markov chain, the generated Markov chain has also to be irreducible (cf. definition 3.10). Remember, that 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 does not happen 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 equation (3.13).

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 this time stamps and log line content have to be generated.

4.5 Populating Log Lines

The following section deals with generating time stamps and log line content for the generated log file. In the proposed model we assume, that the time stamps are independent from the log line content, but they are depending on the cluster the log line belongs to. In other words, the interval between two consecutive log lines depends on the cluster the log lines belong 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 $j \in \{0, \dots, M-1\}$, 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 $j \in \{1, \dots, M-1\}$, calculated, cf. equation (4.12).

$$td_j = ts_j - ts_{j-1} \qquad j \in \{1, \dots, M-1\} \tag{4.12}$$

For every cluster C_i a sequence of time differences TD_i, with $i \in \{1, \dots, n+1\}$ 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) [45] based on the elements of TD_i. An EDF is defined as follows:

Definition 4.2. Let $X_1, \dots, X_n$ be elements of a sample. A function $F : \mathbb{R} \to [0, 1]$ defined as in equation (4.13) is named a *empirical distribution function.*

$$x \mapsto \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}_{(-\infty, x])}(X_i) \tag{4.13}$$

The distribution of elements of TD_i can be described by an EDF.

Next we define the quantile function Q [45], also called inverse cumulative distribution function, of the EDF F.

Definition 4.3. Let F be an EDF, then Q in equation (4.14) defines the *quantile function* of F.

$$Q(p) = F^{-1}(p) := \inf\{x \in \mathbb{R} | p \leq F(x)\},\ 0 < p \leq 1 \tag{4.14}$$

Now we can 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 equation (4.15), 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.

$$td_{rand}(U, TD_i) := \inf\{x \in \mathbb{N} | U \leq F(x)\} \tag{4.15}$$

The time stamp for every generated log line can be calculated as shown in equation (4.16), 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 log file. Note, that the first time stamp ts_1 has to be specified by the user.

$$ts_j = ts_{j-1} + td_{rand}(U, TD_i), \quad j = 2, \dots, L \tag{4.16}$$

The remaining of this section describes our approach for generating log line content. We provide three options which allow generating 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 bases 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 belonging 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 option can be chosen to generate a log file which only reproduces the line sequence of the considered real log file.

Next we introduce an option which allows 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 real 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 are stored also, 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 choosing one word out of the related list. This works exactly the same way as the generation of time differences td, which is shown in equation (4.15). If we consider the example cluster and the log lines

`Connect from 192.168.1.1 port 123` and

`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` and `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 log file resembles the one of the real log file. Furthermore IP addresses are only replaced by IP addresses, and since the generated log file can be any times longer than the real log file, 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 are stored in a list. Since reoccurring values are also stored, using a function as in equation (4.15) 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 eleven characters, and the second one a word with three characters. Therefore, the log line

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

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 are automatic pattern generation algorithm which try to find frequent patterns. Here the patterns should cover the words defining 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.

For each of the proposed options in case the cluster representing the outliers occurs one of the lines stored in the outliers file is drawn.

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

Table 4.1: This table summarizes the three common types of test data: synthetic, real and semi-synthetic. Also their advantages and disadvantages are pointed out. [68]

4.6 Evaluation

The following section deals with the evaluation of the proposed approach, for generating realistic synthetic log data. For evaluating and testing the introduced approach for generating synthetic log data we implemented the previous defined model as a Java [22] application. To perform the log line clustering as presented in section 4.2, we adopted SLCT [48], which as already mentioned provides a C [50] 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 right support threshold value are discussed. Finally we evaluate the Markov chain simulation and the wild card replacement.

4.6.1 Generating Semi-Synthetic Log Data for Testing the Proposed Model

It is impossible to adjust real log data in a way which allows comparing the results obtained by using input data of different complexity. Therefore we decided to generate semi-synthetic log files (cf. table 4.1) for testing the proposed model. Table 4.1 outlines the differences between synthetic, real and semi-synthetic test data and their advantages and disadvantages.

For generating semi-synthetic log data, we applied the approach presented in [68]. This method allows generating log files, of any size and different complexity. Virtual users perform specified actions on a web platform, running the MANTIS Bug Tracker System [69]. In the log files a web server, a database and a firewall are logged. 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 done in which order by which probability, the complexity of the

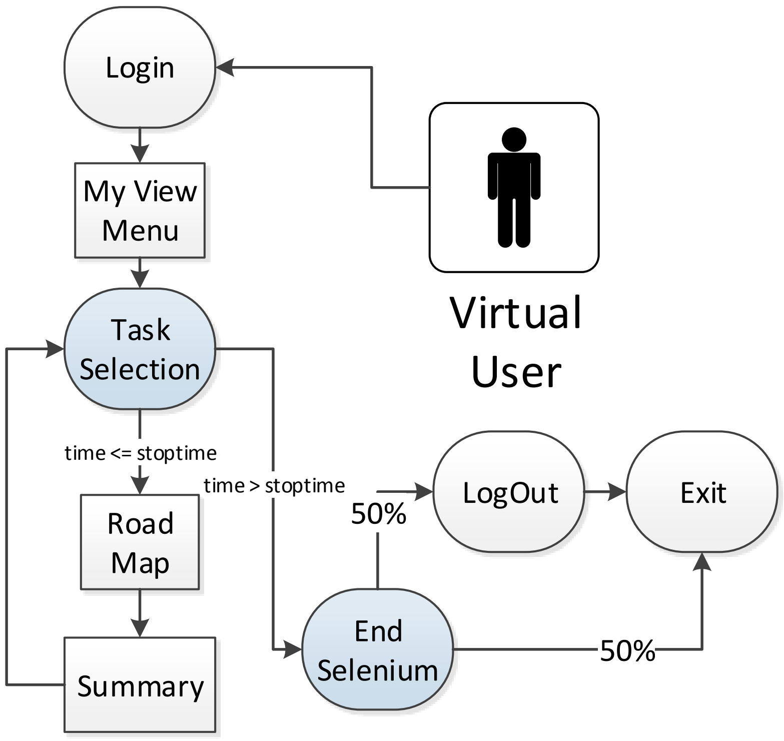

Figure 4.2: This figure shows configuration I, which we use for generating semi-synthetic log data. The complexity in this configuration is kept relatively simple. As long as the (passed) time is smaller than the recorded time (specifies how long the user actions are logged) a simulated user is accessing 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%.

generated log data can be adjusted easily. Also it is possible to configure the time intervals between two consecutive user actions.

For evaluating the proposed approach we generated 4 different log files applying the approach in [69]. In order to simulate different levels of complexity we implemented two configurations - configuration I (low complexity, cf. figure 4.2) and configuration II (high complexity, cf. figure 4.3). Furthermore we changed the number of virtual users operating on the system. For testing simpler cases only one virtual user is operating and for more complex ones four users are simulated. The properties of the considered log files are summarized in table 4.2. For generating the log files the user activity was logged for 10 hours. Table 4.2 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 virtual users) changing from configuration I to

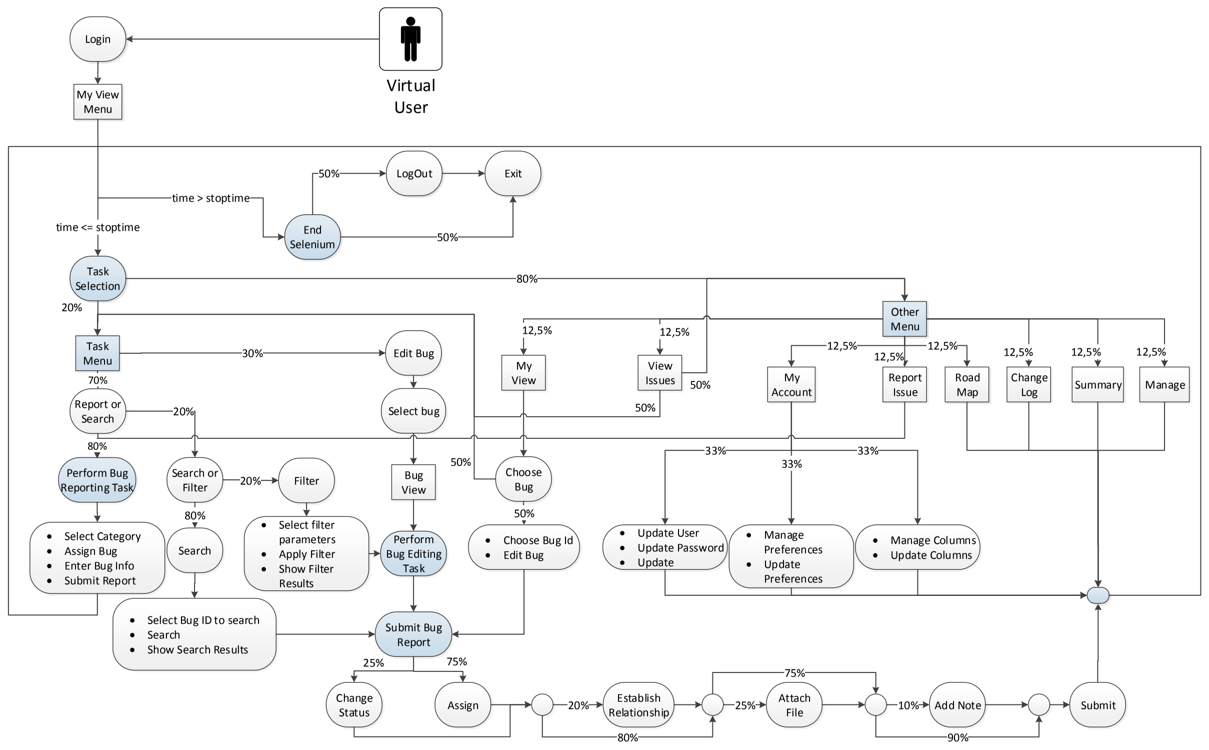

Figure 4.3: This figure shows configuration II, which we use for generating semi-synthetic log data. The complexity in this configuration is much higher than in configuration I (cf. Figure 4.2). The labels of the arrows define with which probability a virtual user performs an action. Again the time it takes a user to set an action is variable and the virtual user logs out and exits or exits immediately after the recorded time expires.

Data Set	*Simulated Users*	*Recorded Time (h)*	*Data Set Length (lines)*	*Used Configuration*
U1C1	1	10	484.239	Configuration I
U4C1	4	10	1.887.824	Configuration I
U1C2	1	10	413.106	Configuration II
U4C2	4	10	1.600.217	Configuration II

Table 4.2: Properties of the considered semi-synthetic log files.

configuration II generated around 15% less log lines. This happens because in configuration II there are more options for the virtual users to choose his next action, because in configuration I in configuration II there are more actions which raise a longer waiting time until a virtual user starts its next action.

4.6.2 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 4.2. 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 4.2. We then analyzed the *Mean Coverage Rate (MCR)*, the *Number of Outliers (NoO)* and the *Number of Clusters (NoC)*.

The Mean Coverage Rate (MCR)

For calculating the MCR, we define n as the length, i.e. number of lines, of the considered log file, l_i as the length of the i-th log line of the considered log file, and cv_i as the cluster value (cf. equation (4.6)) of the cluster the i-th log line has been assigned to. Then the MCR of a log file can be calculated as shown in equation (4.17), where $\frac{l_i}{cv_i}$, with $i \in \{1, \ldots, n\}$ specifies the coverage

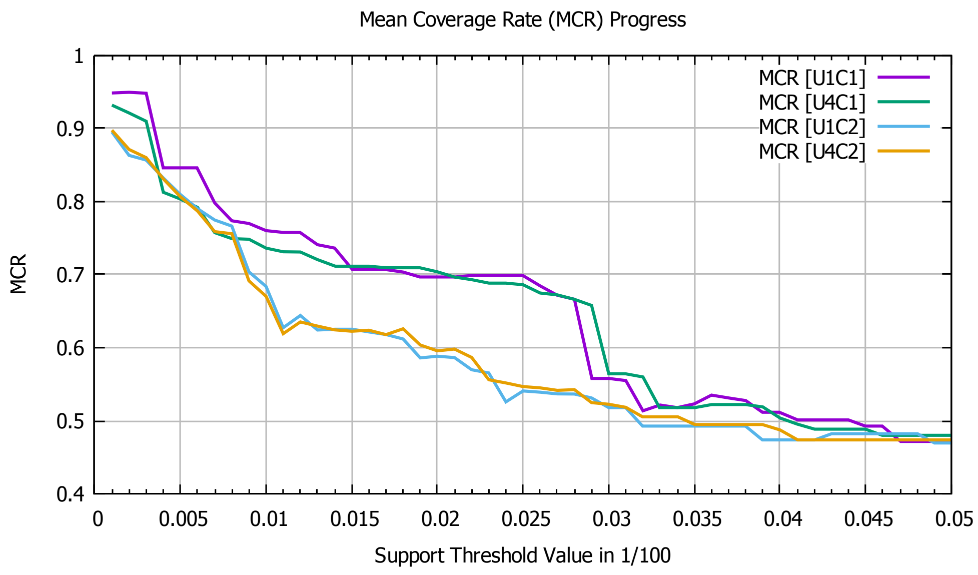

Figure 4.4: This figure shows the progress of the MCR for the log files described in table 4.2. Some of the values are summarized in table 4.3.

$N\ (\frac{1}{100})$	*U1C1*	*U4C1*	*U1C2*	*U4C2*
0.05	0.4721	0.4808	0.4705	0.4747
0.03	0.5583	0.5639	0.5184	0.5225
0.028	0.6662	0.6661	0.5365	0.5424
0.013	0.7409	0.7206	0.6242	0.6295
0.008	0.7736	0.7493	0.7664	0.7560
0.001	0.9486	0.9320	0.8953	0.8978

Table 4.3: This table summarizes some MCR values. See also figure 4.4.

rate for every log line.

$$MCR = \frac{1}{n}\sum_{i=1}^{n}\frac{l_i}{cv_i} \tag{4.17}$$

The progress of the MCR is shown in figure 4.4 and some of the interesting values are summarized in table 4.3. Figure 4.4 demonstrates that the MCR mainly depends on the configuration used to build the log files. It is independent from the number of users been simulated and also from the length of the generated 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 progress 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

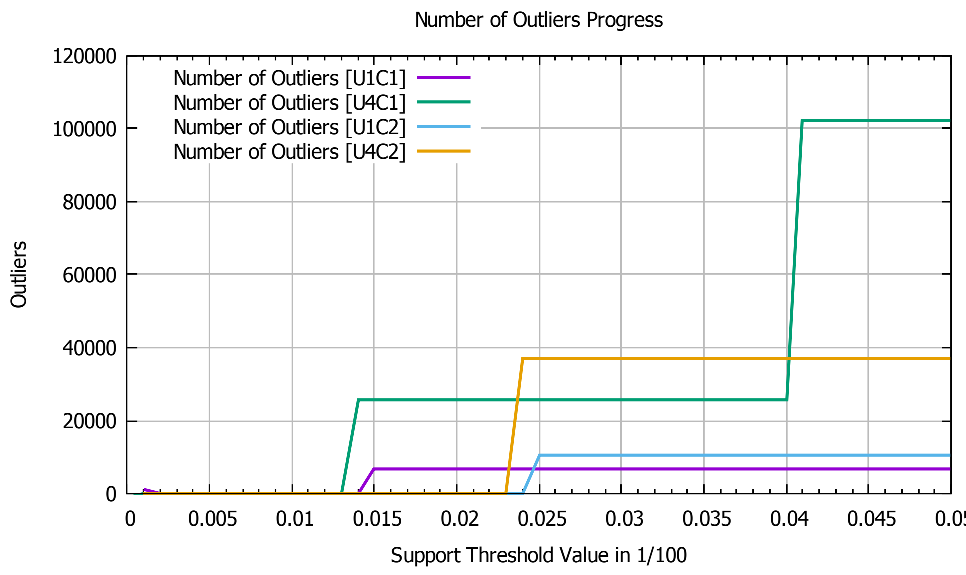

Figure 4.5: This figure shows the progress of the NoO for the log files described in table 4.2. Some of the values are summarized in table 4.4.

N	$U1C1$	$U4C1$	$U1C2$	$U4C2$
0.05	6860	102327	10312	37141
0.03	6860	25466	10312	37141
0.023	6860	25466	40	40
0.013	40	40	40	40
0.001	1134	20	0	0

Table 4.4: This table summarizes some NoO values. See also figure 4.5.

be recognized for support threshold values $N \in [0.008, 0.028]$.

The Number of Outliers (NoO)

The trend of the NoO is shown in figure 4.5 and some relevant values are summarized in table 4.4. Since the NoO is represented in total numbers, figure 4.5 suggests that the NoO progress depends on either the configuration and the number of simulated virtual users, which refers to the log file length. Also the graphs of the NoO progress are more constant, than the ones of the MCR progress. The NoO of the files in which four virtual users have been logged 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 figure 4.6, where $\frac{NoO}{n}$, i.e. the percentage of outliers, is plotted. The NoO for the log files, where configuration II has been used are 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 $N \leq 0.013$.

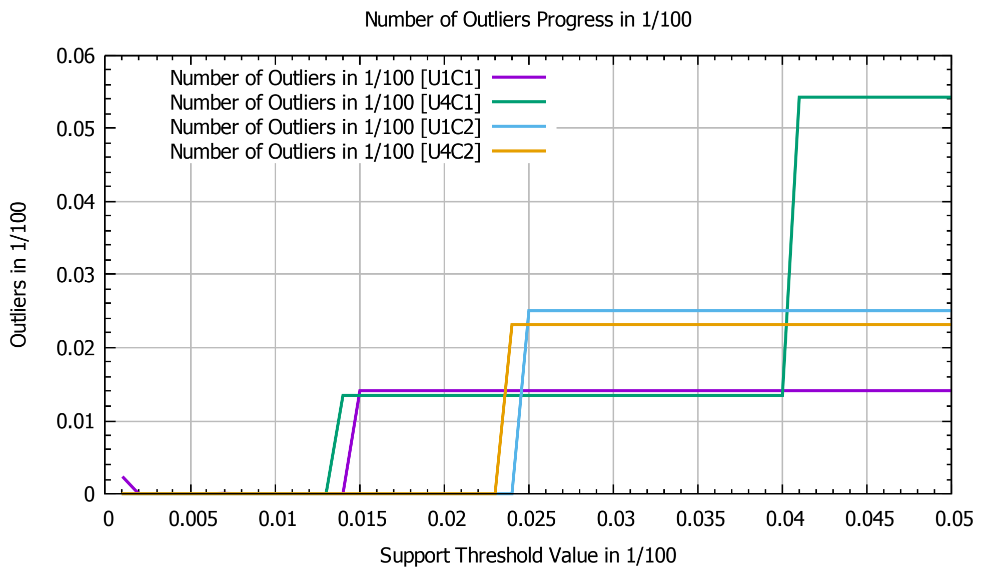

Figure 4.6: This figure shows the progress of the NoO in $\frac{1}{100}$ for the log files described in table 4.2.

N	$U1C1$	$U4C1$	$U1C2$	$U4C2$
0.05	15	15	12	12
0.028	36	38	23	25
0.013	48	51	51	58
0.008	64	67	122	119
0.003	179	178	183	184
0.001	193	213	245	239

Table 4.5: This table summarizes some NoC values. See also figure 4.7.

The Number of Clusters (NoC)

The progress of the NoC is shown in figure 4.7 and some relevant values are summarized in table 4.5. As previously mentioned the NoC mainly depends on the used configuration. Therefore the graphs in figure 4.7 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. Also NoC does not increases very fast. For $N < 0.011$ the NoC of all log files increases faster. Also the NoC in configuration II gets bigger than the NoC in 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 4.3 we mentioned that the clusters generated with SLCT can be arranged in a kind of graph theoretical tree. When decreasing the support threshold value

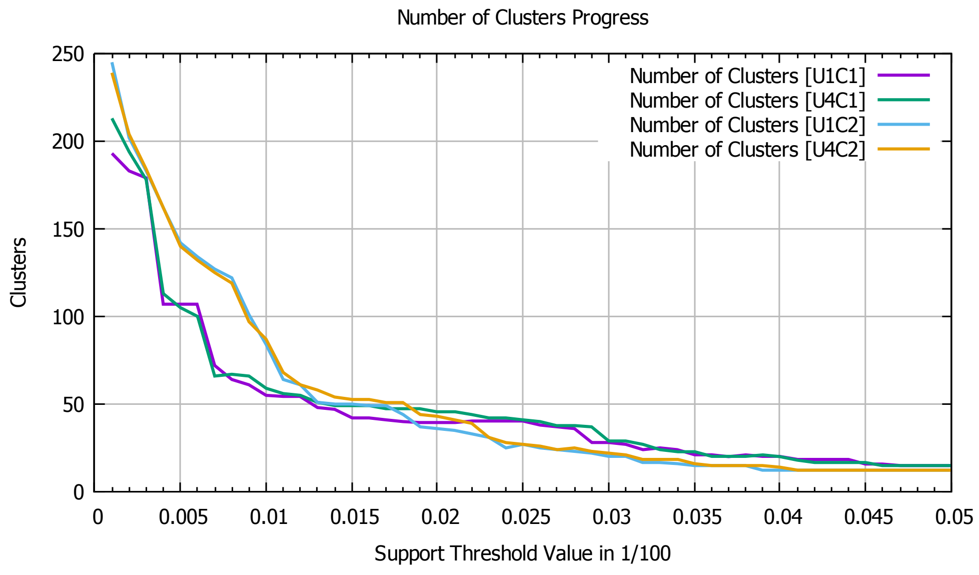

Figure 4.7: This figure shows the progress of the NoC for the log files described in table 4.2. Some of the values are summarized in table 4.5.

Log File	$N \leq$
U1C1	0.018
U4C1	0.020
U1C2	0.009
U4C2	0.008

Table 4.6: Support threshold value N with $MCR \geq 0.7$ per log file.

N more specific clusters are generated which often have the same roots like clusters already existing in the previous iterations. When N becomes smaller SLCT starts earlier building more specific clusters for simpler log files, such as the ones obtained with configuration I.

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 descriptions should a large percentage of the log lines

(ii) there should be a low number of outliers.

To address objective (i) a criteria could be a specific threshold value for the MCR, such as $MCR \geq 0.7$. Table 4.6 shows, for which N this assumption is fulfilled.

To address objective (ii) the NoO should be considered. Since the NoO depends on the log file length, it should be looked at the fraction of the NoO

Log File	$N \leq$
U1C1	0.014
U4C1	0.013
U1C2	0.024
U4C2	0.023

Table 4.7: Support threshold value N with $\frac{NoO}{n} \leq 0.01$ 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	24541	0.7206	0.000021	51
U1C2	0.009	3717	0.7037	0.000048	101
U4C2	0.008	12801	0.7560	0.000012	119

Table 4.8: Options for N according to the assumptions $MCR \geq 0.7$ and $\frac{NoO}{n} \leq 0.01$ per log file.

and log file length which corresponds to the percentage of outliers (cf. figure 4.6). Again a threshold value for the rate of outliers $\frac{NoO}{n}$ can be chosen. For example $\frac{NoO}{n} \leq 0.01$ can be assumed, which means less than 1% outliers. Table 4.7 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 table 4.6 and table 4.7. The results are shown in table 4.8.

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

4.6.3 Evaluating the Markov Chain Approach

In this section we evaluate the output of the Markov chain simulation we applied for generating synthetic log files. On the one hand we want to show that the transitions between consecutive clusters reflects the sequence of the log lines in the original log file, 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.

Evaluating the Transitions

Since we use a Markov chain simulation for generating a synthetic log file, the transition probabilities of the original log file and the generated log file 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 test log file (cf. table 4.2) a synthetic log file, using the support threshold value given in table 4.8. For analyzing the transitions in the generated log file only the cluster of each generated log line is stored.

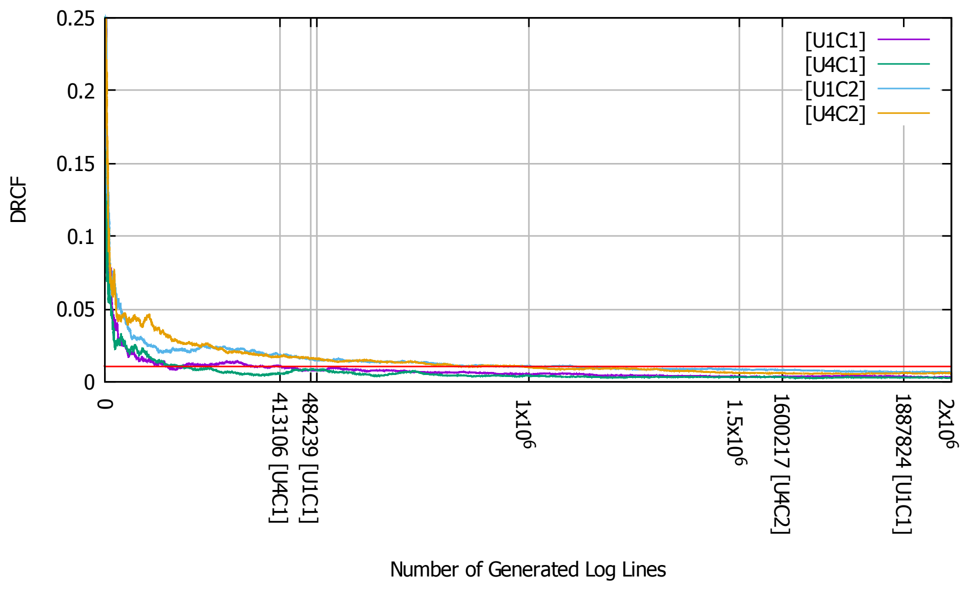

Figure 4.8: The figure shows the progress of the $DRCF$. The red line marks the threshold value $DRCF = 0.01$. Furthermore the lengths of the original log files are marked.

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 equation (4.18), where $\mathbb{1}_{\{C_i\}}$ is the indicator function, which is 1, if the j-th generated line l_j is an element of cluster C_i and 0 if not.

$$CRF(C_i, m) = \frac{1}{m} \sum_{j=1}^{m} \mathbb{1}_{\{C_i\}}(l_j), \text{ for all } i \in \{1, \dots, NoC\} \tag{4.18}$$

In figure 4.8 we consider the progress of the *difference of the relative cluster frequencies* $DRCF$ between the original and the generated log file. The $DRCF$ of a log file after generating m log lines is calculated as shown in equation (4.19), where $relFreq$ returns the relative frequency of a cluster in the original log file (cf. equation (4.20), where $n \in \mathbb{N}$ specifies the length of the original log file).

$$DRCF(m) = \frac{1}{NoC} \sum_{i=1}^{NoC} \left| relFreq(C_i^{original}) - CRF(C_i^{generated}, m) \right| \tag{4.19}$$

$$relFreq(C_i) = CRF(C_i, n), \text{ for all } i \in \{1, \dots, NoC\} \tag{4.20}$$

Figure 4.8 shows the progress of the $DRCF$ for the four 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 4.8), this is explained, 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 the generated log file converge towards the transition probabilities of the original log file, 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 the original log file.

Since the outliers are considered as a cluster during the generation step (cf. section 4.4), the relative frequency of the outliers in the generated log file must be similar to the relative frequency of the outliers in the real log file.

In figure 4.9 we illustrate the difference between the transitions of the original and the generated log file in case of configuration U1C1; both files have the same length (484.239 lines). We calculate T^{diif} (cf. equation (4.21)) the difference of the transition matrices $T^{original}$ and $T^{generated}$ (cf. equation (4.7)). We normalize the difference over the log file length so that the cluster size does not effect the value, i.e. we consider the difference between the relative frequency of the transitions.

$$t_{ij}^{diff} = \frac{\left| t_{ij}^{original} - t_{ij}^{generated} \right|}{n}, \text{ for all } i, j \in \{1, \ldots, NoC\} \tag{4.21}$$

In figure 4.9 the darker the field of a transition is, the lower is the difference between the transitions in the original and in the generated log file. 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 $DRCF$.

Evaluating the Wild Card Replacement

In the following section we evaluate the wild card replacement. The wild cards are replaced by using the probability distribution which describes the relative frequency of the words which occur in the real log file 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 the real log file.

To ensure that we do not create any log lines completely different from the lines of the real log file, we run the clustering algorithm again on the generated log files, with the same support threshold value N we used before for generating them (cf. table 4.8). Also the generated log files have the same length as the real ones. Afterwards we compare the clusters we obtain from the real and the generated log files. The results of this analysis are summarized in table 4.9. If the generated log files do not have the same length as the real ones, the support threshold value must be modified. Otherwise, if the generated log file is longer than the real log file, a larger number of clusters,

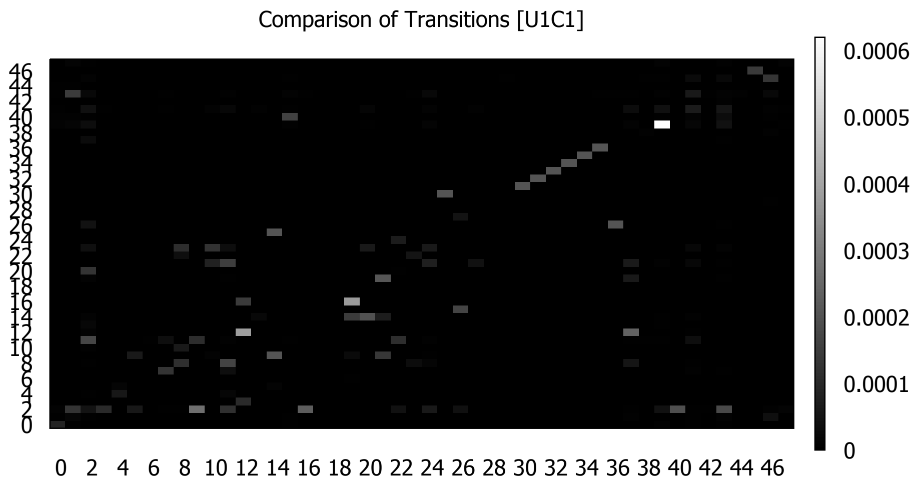

Figure 4.9: The figure shows the differences of the relative frequencies of the transitions between the original and the generated log file with the configuration U1C1.

Log File	*C. real*	*C. gen*	*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

Table 4.9: Again the generated log files have the same length of the real log files. The table compares the number of clusters in the real log file and in the generated one. Furthermore it is summarized the number of clusters occurring in both log files. Furthermore it is shown how many of the different clusters from the generated log file are sub- or supclusters of the clusters of the real log file. The column named *match* indicates how many of the clusters from the generated log file describe lines of the real log file.

which are more specific than the ones of the real log file can be expected. To avoid this for example if the generated log file is twice as long as the real log file, the support threshold value chosen for the analysis must be twice as big as the one used for generating the log file. If the generated log file is shorter than the real one, it is the other way around.

Column 2 and column 3 of table 4.9 compare the number of clusters found in the real and in the generated log file. For all configurations more cluster have been found in the generated log file. The 4th column shows how many clusters are found in both the real and the generated log file. Between 54% and 64% of the clusters are equal. Column 5 shows how many of the different clusters found in the generated log files are subclusters of the real log file. A cluster is considered as a subcluster if it is more specific than another cluster. None of the clusters found in generated log files, which are different and no subcluster, are a supcluster. A supcluster is a more generic cluster.

Since we allowed SLCT generating overlapping clusters also generic clusters are found. Therefore it was predictable that there would be no new generic clusters generated. The last column points out that every cluster of the generated log file describes lines of the real log file. Table 4.9 shows that for configuration I there exist generated clusters which are different to the clusters of the real log file and they are neither subclusters nor supclusters. But since all clusters describe lines of the real log lines, we can be sure that we have not generated a group of log lines significantly different from the real log file, 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 the real log file. Moreover, the rate of outliers occurring in the generated log files is the same as in the real log files.

CHAPTER 5 ILLUSTRATIVE APPLICATION

In this chapter we show an example of application in which we use our generated log data for testing and evaluating the intrusion detection system (IDS) AECID (Automated Event Correlation for Incident Detection) [5]. First we describe the AECID algorithm and its functionalities. Afterwards we demonstrate that log data generated with our novel approach is suitable for testing and evaluating AECID. Finally we assess AECIDS detection's capability AECID, also while simulating an attack.

5.1 Automated Event Correlation for Incident Detection - AECID

Many IDSs are based on blacklist approaches. This means they only consider actions and behavior as malicious, if it matches to known attack patterns or signatures of malware traces. In opposite AECID is a self-learning IDS which implements a white-list approach. This means that the algorithm learns the normal system behavior and can afterwards recognize anomalous behavior. AECID processes log files, and is independent from knowledge about the semantics and the syntax of the log lines. While processing log data AECID builds a *system model* M, comprising the following main building blocks [70, 71]:

- *Search-Patterns*(P): Patterns are random substrings of the processed 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.

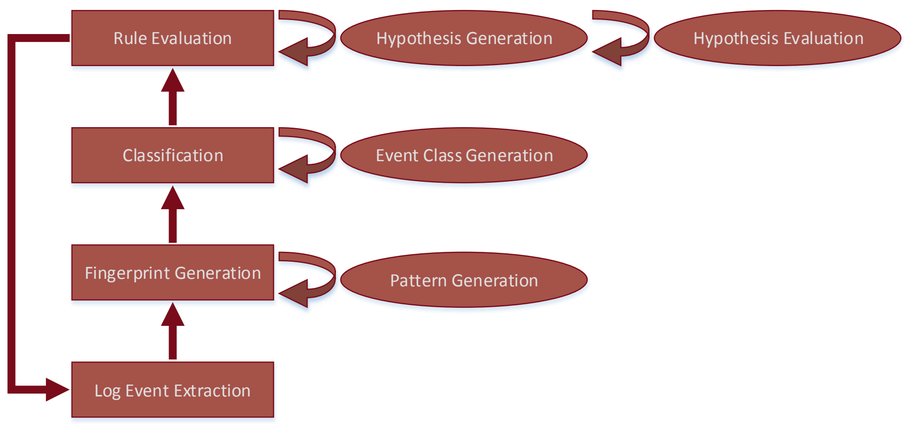

Figure 5.1: This figure describes the concept that AECID implements.

- *Rules* (R): A rule is a hypothesis which has been proven as stable. This means the hypothesis has held in a significant time of evaluations. Rules are used for detecting anomalies in the log data.

The system model M (cf. equation (5.1)) is therefore defined by the set of known patterns $\mathbb{P}$, the set of known event classes $\mathbb{C}$[1], the set of known hypothesis $\mathbb{H}$ and the set of known rules $\mathbb{R}$[2].

$$M = (\mathbb{P}, \mathbb{C}, \mathbb{H}, \mathbb{R}) \tag{5.1}$$

Figure 5.1 shows the concept of the AECID algorithm. In the following we describe the single parts and tasks performed by AECID as described in [70, 71].

5.1.1 Log-Event Extraction

The first step of the algorithm is extracting log-events from the input file (log file). In the described approach a single log line is called a *log atom* L_a. One L_a is defined by the sequence of symbols s it consists of, as shown in equation (5.2), where $n \in \mathbb{N}$ defines the log line length. Note: here is only considered the log line content without the time stamp.

$$L_a = s_1 s_2 \dots s_n \tag{5.2}$$

A log event L_e (cf. equation (5.3)) is defined as the tuple of a L_a and the time stamp t, which specifies when the log line which triggered L_a was

[1]In the following $\mathbb{C}$ corresponds to the set of event classes, not to the set of complex numbers

[2]In the following $\mathbb{R}$ corresponds to the set of rules, not to the set of real numbers

	p_1	p_2	p_3	p_4	p_5	p_6	p_7
Patterns $P \in \mathbb{P}'$	`Query#`	`Conn`	`datab`	`1SELE`	`192.16`	`orma`	`apa`
Fingerprint $\vec{F}$	1	0	1	1	0	1	0

Table 5.1: This table shows the fingerprint $\vec{F}$ of the log line '`database mysql-normal #011#01173640 Query#011SELECT *`' if the set of patterns $\mathbb{P}'$ is considered.

processed.

$$L_e = (L_a, t) \tag{5.3}$$

From this point on the log events L_e are processed one after each another, sorted by their time stamp t.

5.1.2 Fingerprint Generation

During the next step a fingerprint for each log atom is generated by using the set of search patterns $\mathbb{P}$. A search pattern P is defined as a substring of a log atom, with a length of 1 to l symbols s, where l defines the length (number of symbols) of the log atom from which the pattern has been obtained (cf. equation (5.4)).

$$P = s_{1+i} \dots s_{m+i}, \text{ where } 0 \leq i \text{ and } m + i \leq l \tag{5.4}$$

The procedure to generate a pattern P will be explained later.

The fingerprint $\vec{F}$ is a vectorization of the log atom L_a. $\vec{F}$ consists of $|\mathbb{P}|$ elements (cf. equation (5.5)), where every p_i corresponds to an already existing pattern ($p_1 \mapsto P_1, \dots, p_{|(\mathbb{P})|} \mapsto P_{|\mathbb{P}|}$). The element p_i is 1 if the pattern P_1 is a substring of L_a and 0 otherwise.

$$\vec{F} = (p_1, \dots, p_{|\mathbb{P}|}), \text{ with } p_i \in \{0, 1\} \tag{5.5}$$

After the fingerprint generation the algorithm only uses $\vec{F}$ for the following steps of the procedure. An example for a fingerprint is given in table 5.1.

5.1.3 Log Atom Classification

After generating a fingerprint $\vec{F}$ the log atom L_a gets classified. Note that the classification is time stamp independent L_a is classified and not L_e. For the classification the set of already known event classes $\mathbb{C}$ is used. The event classes matching a L_a are stored in $\mathbb{C}_{L_a}$ (cf. equation (5.6)).

$$\mathbb{C}_{L_a} = \{C \in \mathbb{C} | L_a \in C\} \tag{5.6}$$

The log atoms which do not belong to any class are discarded and not considered for further analysis.

	p_1	p_2	p_3	p_4	p_5	p_6	p_7
Patterns $P \in \mathbb{P}'$	`Query#`	`Conn`	`datab`	`1SELE`	`192.16`	`orma`	`apa`
Fingerprint $\vec{F}$	1	0	1	1	0	1	0
$\vec{C}_m$	1	0	0	1	0	1	0
$\vec{C}_v$	1	0	0	1	0	1	0

Table 5.2: This table shows a class that classifies the log line '`database mysql-normal #011#01173640 Query#011SELECT *`' if the set of patterns $\mathbb{P}'$ is considered.

Next we define an event class $C \in \mathbb{C}$. One event class C consists of two binary vectors, a mask vector $\vec{C}_m$ and a value vector $\vec{C}_v$, whereby both have the same dimension (cf. equation (5.7)).

$$C = (\vec{C}_m, \vec{C}_v) \tag{5.7}$$

The mask $\vec{C}_m$ defines which patterns $P \in \mathbb{P}$ are relevant for the class C (cf. equation (5.8)).

$$\vec{C}_m = (p_1, \dots, p_{|\mathbb{P}|}), \text{ where } p_i = \begin{cases} 1, & \text{if } P_i \in \mathbb{P} \text{ is relevant} \\ 0, & \text{if } P_i \in \mathbb{P} \text{ is irrelevant} \end{cases} \tag{5.8}$$

The value $\vec{C}_v$ specifies if pattern P_i is enforced or prohibited in the fingerprint $\vec{F}$ of the log atom L_a (cf. equation (5.9)). Note that the value vector $\vec{C}_v$ only determines which patterns are relevant for the class.

$$\begin{aligned} &\vec{C}_v = (p_1, \dots, p_{|\mathbb{P}|}), \text{ where} \\ &p_i = \begin{cases} 1, & \text{if } P_i \in \mathbb{P} \text{ is enforced} \\ 0, & \text{if } P_i \in \mathbb{P} \text{ is prohibited or irrelevant} \end{cases} \end{aligned} \tag{5.9}$$

A fingerprint $\vec{F}$ is classified by a class C if the condition in equation (5.10) holds.

$$\vec{C}_v = \vec{F} \wedge \vec{C}_m \tag{5.10}$$

Table 5.2 shows an example of a class which classifies a specific fingerprint and table 5.3 an example of a class which does not. How an event class is generated will be discussed later in this section.

For every class $C \in \mathbb{C}_{L_a}$, an event E^C (cf. equation (5.11)) is triggered. An event E^C provides the information that a log atom L_a was classified by C at time t.

$$E^C = (t, C) \tag{5.11}$$

From that point on the AECID considers the events E^C.

	p_1	p_2	p_3	p_4	p_5	p_6	p_7
Patterns $P \in \mathbb{P}'$	`Query#`	`Conn`	`datab`	`1SELE`	`192.16`	`orma`	`apa`
Fingerprint $\vec{F}$	1	0	1	1	0	1	0
$\vec{C}_m$	1	0	1	1	0	1	0
$\vec{C}_v$	0	0	0	1	0	1	0

Table 5.3: This table shows a class that does not classify the log line '`database mysql-normal #011#01173640 Query#011SELECT *`' if the set of patterns $\mathbb{P}'$ is considered.

Occurence	*Evaluation Result*
$E^{C_{cond}} \wedge \neg E^{C_{impl}}$	$E^{C_{impl}}$ did not occur and the time window t_w has passed. The hypothesis evaluates to false.
$E^{C_{cond}} \wedge E^{C_{impl}}$	Both events occurred before t_w passed. The hypothesis evaluates to true.
$\neg E^{C_{cond}}$	$E^{C_{cond}}$ did not occur. No evaluation result is returned.

Table 5.4: This table summarizes the possible results of an evaluation of a hypothesis.

5.1.4 Hypothesis Evaluation

A hypothesis $H \in \mathbb{H}$ consists of a conditional class C_{cond}, an implied class C_{impl} and a time window t_w, which can be either greater or smaller than zero. (cf. equation(5.12)).

$$(C_{cond}, C_{impl}, t_w) \tag{5.12}$$

A hypothesis evaluates if the correlation $E^{C_{cond}} \rightarrow E^{C_{impl}}$ holds in t_w. For a continuous evaluation process, for every hypothesis $H \in \mathbb{H}$, the relevant events are stored in a queue Q^H. The cases which the evaluation of a hypothesis H based on Q^H can lead to are summarized in table 5.4.

The results of a hypothesis evaluation e are stored as shown in equation (5.13), where res specifies the result of the evaluation (true or false), H the evaluated hypothesis and pos the position of the evaluation in the stream of evaluations S^H of the hypothesis H. There is one stream S^H for every hypothesis $H \in \mathbb{H}$

$$e = (res, H, pos) \tag{5.13}$$

The newest evaluation always gets the value $pos = 0$ and the pos of older evaluations is increased by 1.

We define a *slot* Sl_k, with $k \in \mathbb{N}$, which works as a filter and returns the last k evaluations, when applied on S^H (cf. equation (5.14)).

$$Sl_k(S^H) = \{e | e \in S^H \wedge pos < k\} \tag{5.14}$$

The result value of a evaluation $res(e)$ is 1 if the result is true and 0 if the result is false. Therefore the result values form a binary stream and the continuous evaluation process of a hypothesis can be interpreted as a bernoulli process (cf. definition 3.23). Hence a binomial test (cf. definition 3.24) is applied for testing the stability of a hypothesis. The binomial test in equation (5.15) tests the stream of evaluations S^H against a predefined stochastic distribution p_0. Therefore $\{e_t^H\}$ is the set of evaluations of the hypothesis H that evaluated to true. In other words, a hypothesis is considered stable if the probability that it evaluates to true is greater or equal to p_0.

$$isStable(H) = \sum_{i=0}^{|\{e_t^H \in Sl_k\}|} B(i|p_0, k) \geq \alpha \tag{5.15}$$

If a hypothesis is considered stable it becomes a rule $R \in \mathbb{R}$ (cf. equation (5.16)), which is then used for detecting anomalies.

$$\begin{aligned} &\mathbb{R} = \{H | isStable(H) \geq \alpha\} \\ &\mathbb{R} \subseteq \mathbb{H} \end{aligned} \tag{5.16}$$

The generation of hypothesis/rules will be discussed later in the following

5.1.5 Anomaly Detection

For the anomaly detection the rules $R \in \mathbb{R}$ are evaluated. The procedure of the rule evaluation is similar to the evaluation of hypothesis. But in the case of a rule evaluation p_0 is not a predefined stochastic distribution anymore. It is calculated as shown in equation (5.17) (where t_{stab} is the time when the hypothesis evaluates stable), which calculates the ratio between the number of true evaluations and the number of all considered evaluations, when the slot Sl_k is applied to S^R.

$$p_0 = \frac{|\{e_{t_{stab}}^H \in Sl_k(S^H)\}|}{k} \tag{5.17}$$

AECID detects anomalies by evaluating the rules on three slots $Sl_{k_1}, Sl_{k_2}, Sl_{k_3}$ of different size $k_1, k_2, k_3 \in \mathbb{N}$ with $k_1 < k_2 < k_3$[3]. Furthermore a significance-level α_A specifies a threshold under which an anomaly is triggered. The evaluation of a rule is shown in equation (5.18).

$$isAnomalous(R) = \bigvee_{j=1}^{3} \left(\sum_{i=0}^{|\{e_t^R \in Sl_{kj}\}|} B(i|p_0, k_j) \leq 1 - \alpha_A \right) \tag{5.18}$$

If the evaluation is false for one of the slots an anomaly A (cf. equation (5.19)) is triggered, where $p = 1 - \sum_{i=0}^{|\{e_t^R \in Sl_{kj}\}|} B(i|p_0, k_j)$ is the probability that A is an anomaly, R is the rule that triggered it and t specifies the time at which the anomaly was detected.

$$A = (p, R, t) \tag{5.19}$$

[3]The biggest slot Sl_{k3} is also used for evaluating hypothesis.

5.1.6 Search Pattern Generation

The search patterns $P \in \mathbb{P}$ build the base for the vectorization performed by generating a fingerprint $\vec{F}$ for every processed log atom L_a. Therefore a set of patterns $\mathbb{P}$ which properly covers every log atom L_a is sufficient. New patterns are generated from currently processed log atoms. To achieve a good coverage more patterns should be generated from uncovered and weakly covered log atoms. Therefore a token bucket algorithm as presented in algorithm 4 is applied. The most important input parameter is the pattern cost τ_P. The pattern cost τ_P consists of a predefined pattern base cost τ'_P and a pattern balancing cost τ''_P, which is calculated as shown in equation (5.20), where the *matchingCount* specifies how many of the already existing patterns $P \in \mathbb{P}$ match the currently processed log atom L_a.

$$\tau''_P = 2^{matchingCount} \tag{5.20}$$

The algorithm increases the *bucketSum* by 1 for every processed log atom L_a. Every time the *bucketSum* is greater than the pattern cost τ_P, a new pattern P is generated from L_a. If the pattern candidate already exists in $\mathbb{P}$, it is discarded and the *bucketSum* is not reduced by τ_P.

The pattern balancing cost τ''_P guarantees that more patterns are generated from uncovered and weakly covered log atoms. On the one hand this is important, since the less often occurring log atoms are more interesting. On the other hand it is also important to generate new patterns from well covered log atoms. Since the event classes $C \in \mathbb{C}$ are based on the patterns $P \in \mathbb{P}$, this prevents from generating too generic event classes.

```
Data: $L_a$, $\tau'_P$, $\tau''_P$
Result: $P$
1  $bucketSum$++
2  $matchingCount \leftarrow$ sum_matching_search_patterns($L_a$)
3  $\tau_P \leftarrow \tau'_P + \tau''_P$
4  if $bucketSum \geq \tau_P$ then
5      $candidate$ = get_random_substring()
6      if $\nexists P \in \mathbb{P} \equiv substring$ then
7          $P \leftarrow substring$
8          $bucketSum = bucketSum - \tau_P$
9      end
10 end
```

Algorithm 4: Creation of a new search pattern P [71]

5.1.7 Event Class Generation

To generate event classes C a token bucket algorithm (as shown in algorithm 5) is applied. Again it is important to balance the event class generation procedure because every log event L_e, which cannot be classified by any class $C \in \mathbb{C}$, carries no information for the anomaly detection process. Therefore the class cost τ_C consists of a predefined class base cost τ'_C and a predefined τ''_C, which is multiplied with the *matchingCount*, which specifies how many of the already existing event classes $C \in \mathbb{C}$ are matching the currently processed log event L_e.

The classification is performed on the fingerprint $\vec{F}$ of a log event L_e, which is a binary vector. To guarantee event classes with an accurate entropy, the following three parameters influence the class generation procedure:

(i) *Enforced search patterns* (μ_e): This parameter defines a minimum number of patterns to be enforced in a new event class.

(ii) *Percentage of matching search patterns that will be enforced* (ϕ_e): This number specifies the percentage of the patterns $P \in \mathbb{P}$ matching L_e, which will be enforced in a new event class, i.e. specifies the number of patterns which are required.

(iii) *Percentage of not matching search patterns that will be prohibited* (ϕ_p): Similar to ϕ_e this number specifies the percentage of the patterns $P \in \mathbb{P}$ not matching L_e, which will be prohibited in a new event class.

The algorithm prevents generating a new class if the number of patterns matching L_e (*matchingBits*) is lower than μ_e. Furthermore it forces at least μ_e patterns to be enforced if the $number\_enforced\_bits$ is smaller than μ_e.

If an event class candidate already exists in the set of event classes $\mathbb{C}$ this candidate is rejected and the *bucketSum* is not reduced by τ_C.

5.1.8 Hypothesis and Rule Generation

The generation of hypothesis H is based on the event classes characterizing the currently processed log atom $\mathbb{C}_{L_a}$ (cf. equation (5.6)). Again the same token bucket algorithm as for the pattern and class generation is applied. The hypothesis cost τ_H consists of a predefined hypothesis base cost τ'_H and a predefined hypothesis balance cost τ''_H. Also the algorithm prevents generating already in the set of hypothesis $\mathbb{H}$ existing hypothesis. The chosen correlated event classes and the time window t_w are random.

The existing hypothesis $H \in \mathbb{H}$ are periodically tested on their stability. If *isStable*(H) (cf. equation (5.15)) evaluates to true for a hypothesis $H \in \mathbb{H}$, the hypothesis is considered as a rule R and moved to the set of rules $\mathbb{R} \subseteq \mathbb{H}$.

The stability test is performed on the largest slot Sl_{k_3} used by AECID for detecting anomalies. Indeed an accurate number of evaluations e of a hypothesis are needed to decide about its stability.

```
    **Data**: $\vec{F}$, $\mu_e$, $\phi_p$, $\phi_e$, $\tau'_C$, $\tau''_C$
    **Result**: $\vec{C_v}$, $\vec{C_m}$
 1  $bucketSum{+}{+}$
 2  $matchingCount \leftarrow$ numberOfClassifyingEventClasses($\vec{F}$)
 3  $\tau_C \leftarrow \tau'_C + (matchingCount * \tau''_C)$
 4  if $bucketSum \geq \tau_C$ then
 5      $matching\_bits \leftarrow \{p_i \mid p_i \in \vec{F} \land p_i = 1\}$
 6      $number\_enforced\_bits \leftarrow \phi_e * |matching\_bits|$
 7      $not\_matching\_bits \leftarrow \{p_i \mid p_i \in \vec{F} \land p_i = 0\}$
 8      $number\_prohibited\_bits \leftarrow \phi_p * |not\_matching\_bits|$
 9      if $|matching\_bits| < \mu_e$ then
10          abort
11      end
12      if $number\_enforced\_bits < \mu_e$ then
13          $number\_enforced\_bits \leftarrow \mu_e$
14      end
15      enforceNRandomPatterns($number\_enforced\_bits$, $\vec{C_m}$, $\vec{C_v}$, $\vec{F}$)
16      prohibitNRandomPatterns($number\_prohibited\_bits$, $\vec{C_m}$, $\vec{C_v}$, $\vec{F}$)
17      if $\nexists C \in \mathbb{C} \equiv (\vec{C_m}, \vec{C_v})$ then
18          $C \leftarrow (\vec{C_m}, \vec{C_v})$
19          $bucketSum = bucketSum$ - $\tau_C$
20      end
21  end
```

Algorithm 5: Creation of a new class C [71].

5.2 Evaluation of the AECID System

In the following section we adapt real and generated log files with the configuration U4C2 (cf. section 4.6.1) for evaluating the AECID system.

5.2.1 Is the Generated Log Data Suitable to Evaluate AECID?

In the following section we verify that the log data we generated with our novel approach is suitable to test and evaluate AECID. AECID can run on the generated log data since it is independent from the syntax and the semantics of its input log data. Moreover we intend to show that the generated log data can be used to evaluate and test AECID in a specific user environment, which is given by the real log data. We first ran AECID on the real log file with the configuration U4C2 (cf. table 4.2) and then on the log file we generated based on the real log file with the support threshold value N as given in table 4.8 (N = 12801). Since AECID depends on the log file length, both log files consist of 1.600.217 log lines (cf. table 4.2).

Parameter	*Value*
Pattern base cost τ'_P	1
Event class base cost τ'_C	1
Hypothesis base cost τ'_H	30
Event class balancing cost τ''_E	30
Hypothesis balancing cost τ''_H	300
Minimum pattern length	3
Maximum pattern length	12
Minimum enforced patterns μ_e	3
Percentage of enforced patterns ϕ_e	40%
Percentage of prohibited patterns ϕ_p	50%
Stochastic distribution p_0	$0,95$
Stability significance level α	$0,1$
Anomaly significance level α_A	$0,9999$
Slot size k_1	10
Slot size k_2	100
Slot size k_3	1000

Table 5.5: This table summarizes the AECID basic configuration given in [71]. The parameters are described in section 5.1.

To evaluate if the generated log data is suitable for testing AECID under conditions of the network environment given by the real log data, we use for AECID the basic configuration given in [71]. Table 5.5 summarizes the parameters and their values in this configuration.

For deciding if the log file generated with our novel approach 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 equation (5.21); it is the ratio between the Average Number of Enforced Patterns $ANEP$ in the event classes $\mathbb{C}$ and the percentage of enforced patterns ϕ_e in every class $C \in \mathbb{C}$.

$$ALC = \frac{ANEP}{\phi_e} \tag{5.21}$$

The ALC specifies how many patterns $P \in \mathbb{P}$ match the log atoms L_a on average, that obtained 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 equation (5.22) [71]. The FPR is the ratio between the number of anomalous rule evaluations if no anomaly

	real ALC	*gen ALC*	*real FPR*	*gen FPR*
Mean	$17,5831$	$18,5233$	$0,0526$	$0,0546$
Median	$17,7767$	$18,3686$	$0,0403$	$0,0407$
Minimum	$15,1145$	$16,7241$	$0,0025$	$0,0088$
Maximum	$19,9802$	$21,6642$	$0,1329$	$0,1922$

Table 5.6: This table shows the results for the ALC and the FPR, when running AECID with the basic configuration (cf. table 5.5) on the real and the generated log file based on the configuration u4c2.

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.

$$FPR = \frac{FP}{FP + TN} \tag{5.22}$$

Since we consider both the real and the generated log file 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 5.6 shows the results of the analysis of the ALC and the FPR, when running AECID with the basic configuration as shown in table 5.5 on the real and the generated log file. Since AECID depends on random numbers, 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 is on average for the real log file $17,5$ patterns and on the generated one 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 the generated log file are slightly larger than the ALC values obtained with the real one. 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 the real and the ALC of the generated log file is less than 1 pattern. The table also shows that according to the ALC the algorithm performs better with the generated log file. This can be explained by the fact that the generated log file is based on more deterministic conditions.

Since the FPR is a ratio it is given in percentage. The average FPR obtained with the generated log file is just $0,2\%$ higher than the FPR obtained with the real log file. 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 shows that the dependency of AECID on the random numbers is relatively strong. But since AECID implements a self-learning approach and the tested log files only map 10 hours 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 the real and the generated log file. This proves that it is possible to effectively test and evaluate AECID's performance in a network

Parameter	*Value*
Pattern base cost τ'_P	1
Event class base cost τ'_C	1, 30
Hypothesis base cost τ'_H	1, 30
Event class balancing cost τ''_E	30, 300
Hypothesis balancing cost τ''_H	300
Minimum pattern length	3
Maximum pattern length	12
Minimum enforced patterns μ_e	3
Percentage of enforced patterns ϕ_e	40%, 50%, 60%
Percentage of prohibited patterns ϕ_p	30%, 40%, 50%
Stochastic distribution p_0	$0,95$
Stability significance level α	$0,1$
Anomaly significance level α_A	$0,9999$
Slot size k_1	10
Slot size k_2	100
Slot size k_3	1000

Table 5.7: This table summarizes the different AECID configurations we used for finding the optimal one.

environment with the characteristics of the real log file, by using log data generated with our approach.

5.2.2 Finding the Optimal Configuration

The following section discusses the best AECID configuration for a given network environment characterized by the log file generated with our novel approach, based on the configuration U4C2. Therefore we properly change significant input parameters summarized in table 5.5. The seed value for generating the random numbers is fixed, this means that the random numbers generated in every run are the same. This allows generating comparable results, which are independent from the seed. Hence it is possible to find the optimal configuration just based on the input parameters.

The values we used for finding the best configuration are shown in table 5.7. The choice is influenced by the values used in the evaluation section of [71]. Since we ran AECID on an anomaly free data set, we want to find the configuration with the lowest FPR. After running AECID with all possible combinations of configuration given by table 5.7, the configuration shown in table 5.8 turned out to be the optimal configuration according to the FPR. In the following we refer to this configuration as the optimal configuration .

The FPR we obtained using the optimal configuration was $0,46\%$. In comparison the mean of the FPR over all configurations was $3,49\%$ and the maximum was $21,02\%$. Also the ALC obtained with the optimal configura-

Parameter	*Value*
Pattern base cost τ'_P	1
Event class base cost τ'_C	30
Hypothesis base cost τ'_H	1
Event class balancing cost τ''_E	300
Hypothesis balancing cost τ''_H	300
Minimum pattern length	3
Maximum pattern length	12
Minimum enforced patterns μ_e	3
Percentage of enforced patterns ϕ_e	60%
Percentage of prohibited patterns ϕ_p	40%
Stochastic distribution p_0	$0,95$
Stability significance level α	$0,1$
Anomaly significance level α_A	$0,9999$
Slot size k_1	10
Slot size k_2	100
Slot size k_3	1000

Table 5.8: This table describes the optimal configuration of AECID for a low FPR, if it is applied on the generated log file based on the configuration u4c2.

tion was with $20,0752$ patterns higher than the mean over all configurations $19,7442$ patterns. In the optimal configuration the event class base cost τ'_C, the event balancing cost τ''_C and the percentage of enforced patterns ϕ_e is higher than in the basic configuration. On the opposite the hypothesis base cost τ'_H is lower and the percentage of prohibited patterns ϕ_p is equal.

In figure 5.2 the trend of number of search patterns, event classes and stable rules against the number of lines when applying the optimal configuration is plotted. The graphs show that the number of patterns and rules become stable. The number of event classes increases during the whole execution, but the more lines are processed by AECID the slower is the growth of the number of event classes. With longer log files we expect the number of event classes to become stable as well. The trends shown in figure 5.2 become stable because the token bucket algorithm balances the generation process.

5.2.3 Evaluating the Attack Detection Capability of AECID

In the following section we evaluate AECID while an anomaly is simulated in the network, which generates the logs. To simulate an attacker that tries to access the data base without being detected, we manually disabled the logging function of the data base server. We launched the attack twice, once for thirty seconds and once for two minutes, to show that AECID can detect attacks of short and long duration. Between the two attacks we had a break of one hour. The training phase, i.e. the time before the anomaly was injected, was around

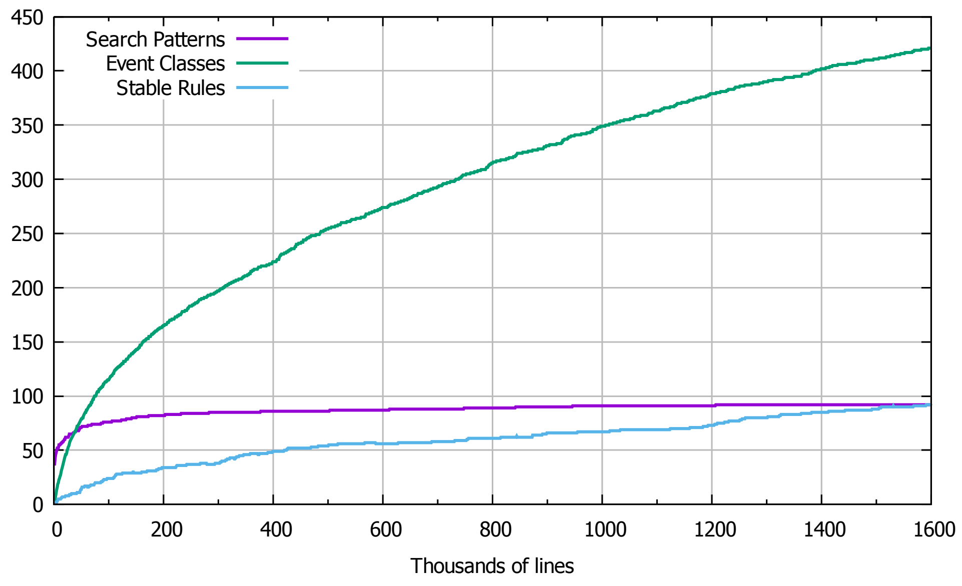

Figure 5.2: This figure shows the trend of number of search patterns, number of event classes and number of stable rules against the number of processed log lines.

8 hours and 30 minutes.

For this evaluation we used the generated log file with the configuration U4C2 and ran AECID with the optimal configuration shown in table 5.8. In the following we focus on the results of one rule to illustrate the output is provided by AECID. Figure 5.3 shows the anomaly probability graph of the rule we are looking at. All the three slots Sl_{k_i} detected the two anomalies we injected. The plots also show that the larger the slot size is the longer the rule evaluates to false. This means that the rule also triggers false positives. However, the graphs show that these false positives are caused by the injected anomalies.

In table 5.9 the FPR and the TPR of the rule are summarized. A true positive TP is an anomalous rule evaluation during the anomaly is injected. The TPR (cf. equation (5.23)) is the ratio between the true positives and the sum of the TP and the false negatives FN, which are the not anomalous rule evaluations during the anomaly is injected [71].

$$TPR = \frac{TP}{TP + FN} \tag{5.23}$$

Table 5.9 shows the results already indicated by figure 5.3. For all slots the TPR is 100%. Conversely, the FPR gets lager, the larger the slot size is.

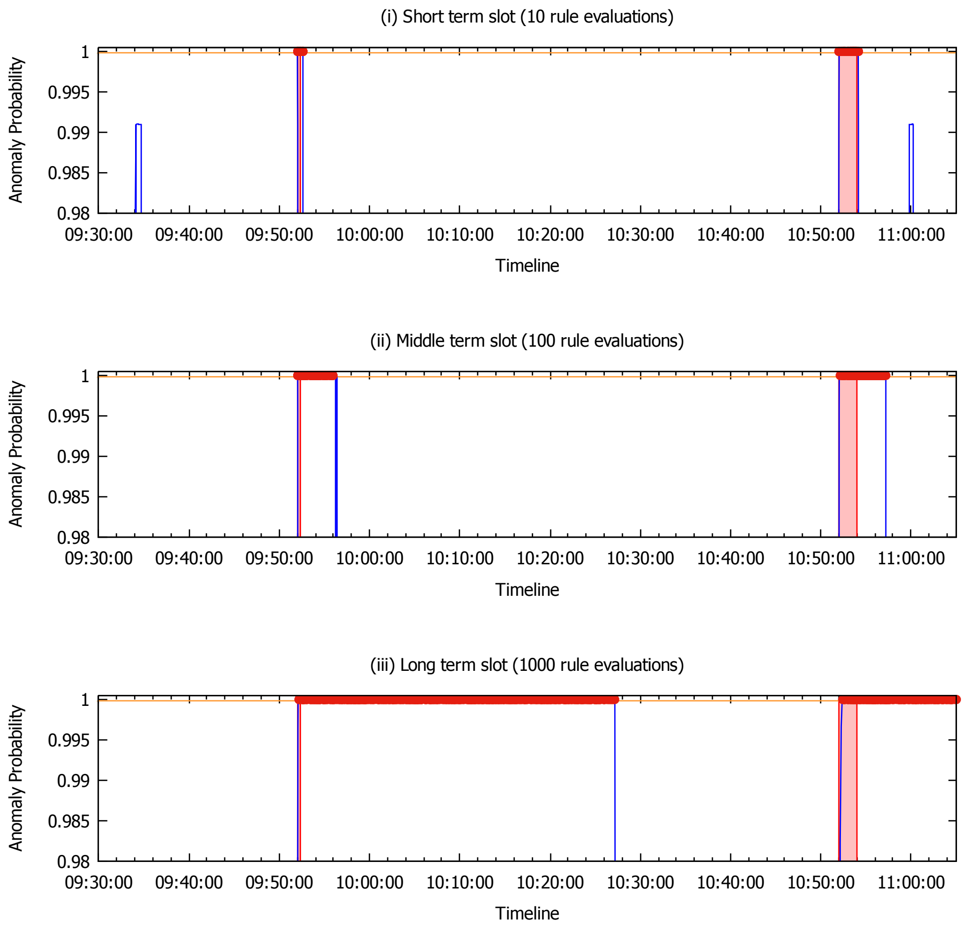

Figure 5.3: This figure shows the anomaly probability graph of one rule AECID uses for detecting anomalies. Figure (i) shows the results for the slot Sl_{k_1}, (ii) for slot Sl_{k_2} and (iii) for slot Sl_{k_3}. The red blocks characterize injected anomalies and red dots characterize detected anomalies. The orange line symbolizes the anomalous threshold α_A

$Slot$	FPR	TPR
Sl_{k_1}	$0,0093$	1
Sl_{k_2}	$0,1009$	1
Sl_{k_3}	$0,7257$	1

Table 5.9: This table summarizes the TPR and the FPR in the respective slot.

CHAPTER 6

CONCLUSION

This chapter concludes this thesis. Section 6.1 summarizes the performed work and presents the key findings of the research activity. Finally section 6.2 gives an outlook on future work.

6.1 Summary and Key Findings

In this thesis we presented a novel approach for generating synthetic network log data. The approach takes as input a small set6 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. This model allows the generation of log files of any size based on the properties of a specific network environment. To verify the effectiveness of the so generated log data we introduced novel metrics such as the mean coverage rate MCR and the difference of the relative cluster frequencies $DRCF$ (cf. sections 4.6.2 and 4.6.3). To prove the similarity between the real and the generated log file we execute the clustering algorithm on the generated log data and compared the clusters obtained with the real and those obtained with the generated log data.

Since the aim of the thesis was to design a model for generating synthetic log data for the evaluation of intrusion detection systems (IDS), we used AECID as an illustrative example of an application for the generated log data. First we proved that the generated log data is suitable for testing and evaluating AECID. Therefore we executed AECID with the real and the generated log data and compared the average line coverage ALC and the false positive rate FPR (cf. section 5.2.1) obtained by AECID.

We discussed then the optimal configuration of AECID. Using this configuration we executed AECID on a log file including a manually injected anomaly, and we showed how well AECID could detect the anomalous behavior. This verifies that our approach can not only be used for testing and evaluating IDSs, it can also be applied for determining IDSs' optimal configuration.

6.2 Future Work

The proposed model is composed by several modules, which can be flexibly be 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 occurrence of a log line can depend not only on the single previous log line, but the privious n lines can be considered.

The introduced approach can be part of a IDS testbed, aiming at helping to compare IDSs in specific network environments. As already mentioned also the optimal configuration for log-based IDSs can be identified. Moreover the proposed model can be utilized to generate test data for several other applications, processing log data.

BIBLIOGRAPHY

[1] C. Tankard, “Advanced persistent threats and how to monitor and deter them,” *Network security*, vol. 2011, no. 8, pp. 16–19, 2011.

[2] AIRBUS Defence and Space. http://www.cybersecurity-airbusds.com/en_US/cybersecurity/advancedpersistentthreataptcheck/, 2015. Accessed: 2015-07-06.

[3] Mandiant, “Threat report 2014. m-trends. beyond the breach.,” tech. rep., Mandiant, 2014.

[4] P. Chen, L. Desmet, and C. Huygens, “A study on advanced persistent threats,” in *Communications and Multimedia Security*, pp. 63–72, Springer, 2014.

[5] I. Friedberg, F. Skopik, and R. Fiedler, “Cyber situational awareness through network anomaly detection: state of the art and new approaches,” 2015.

[6] AIT Austrian Institute of Technology GmbH, “Research service: ICT-Security.” `http://www.ait.ac.at/research-services/research-services-digital-safety-security/ict-security/`. Accessed: 2015-07-03.

[7] S. Jajodia, P. Liu, V. Swarup, and C. Wang, *Cyber situational awareness*, vol. 14. Springer, 2010.

[8] U. Franke and J. Brynielsson, “Cyber situational awareness–a systematic review of the literature,” *Computers & Security*, vol. 46, pp. 18–31, 2014.

[9] T. Mather, S. Kumaraswamy, and S. Latif, *Cloud security and privacy: an enterprise perspective on risks and compliance.* ” O’Reilly Media, Inc.”, 2009.

[10] S. Goel, Y. Hong, V. Papakonstantinou, and D. Kloza, *Smart Grid Security*. Springer, 2015.

[11] D.-G. Feng, Y. Zhang, and Y.-Q. Zhang, "Survey of information security risk assessment," *Journal-China Institute of Communications*, vol. 25, no. 7, pp. 10–18, 2004.

[12] A. Elçi, *Theory and Practice of Cryptography Solutions for Secure Information Systems*. IGI Global, 2013.

[13] S. Axelsson, "Intrusion detection systems: A survey and taxonomy," tech. rep., Technical report Chalmers University of Technology, Goteborg, Sweden, 2000.

[14] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, p. 15, 2009.

[15] C. Lutzky and M.-H. Teichmann, "Logfiles in der marktforschung: Gestaltungsoptionen für analysezwecke," *Jahrbuch der Absatz-und Verbrauchsforschung*, vol. 48, no. 3, pp. 295–317, 2002.

[16] R. Kimball and R. Merz, *The data webhouse toolkit*. Wiley, 2000.

[17] Graylog, Inc., "Graylog v1.1.4." `https://www.graylog.org/`. Accessed: 2015-07-06.

[18] Hewlett-Packard Development Company, L.P., "ArcSight Logger 6.0." `http://www8.hp.com/us/en/software-solutions/arcsight-logger-log-management/index.html`. Accessed: 2015-07-06.

[19] AlienVault, Inc, "AlienVault OSSIM v5.0." `https://www.alienvault.com/products/ossim`. Accessed: 2015-07-06.

[20] R. Gerhards, "Neuere entwicklungen in der syslog-protokollierung," *Datenschutz und Datensicherheit-DuD*, vol. 33, no. 12, pp. 723–727, 2009.

[21] Apache Software Foundation, "Log4j 2 2.3." `http://logging.apache.org/log4j/2.x/`. Accessed: 2015-07-06.

[22] Sun Microsystems, "Java 8 update 45." `http://www.oracle.com/technetwork/java/index.html`. Accessed: 2015-07-06.

[23] Microsoft, "Enterprise library 6." `https://msdn.microsoft.com/library/cc467894.aspx`. Accessed: 2015-07-06.

[24] Lorne Brinkman, ".net logging framework." `http://www.theobjectguy.com/DotNetLog/`. Accessed: 2015-07-06.

[25] Microsoft, ".NET Framework 4.5.51209." `https://msdn.microsoft.com/de-de/vstudio/aa496123`. Accessed: 2015-07-06.

[26] Log4Delphi Project, "Log4Delphi 0.8." `http://log4delphi.sourceforge.net//`. Accessed: 2015-07-06.

[27] Embarcadero Technologies, "Delphi XE8." `http://www.embarcadero.com/`. Accessed: 2015-07-06.

[28] Symantec Corporation, "Internet security threat report 2014," *2013 Trends*, vol. 19, 2014.

[29] J. P. Anderson, "Computer security threat monitoring and surveillance," tech. rep., Technical report, James P. Anderson Company, Fort Washington, Pennsylvania, 1980.

[30] L. Marinos, "ENISA threat landscape 2014: Overview of current and emerging cyber-threats," 2014.

[31] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung, "Intrusion detection system: A comprehensive review," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 16–24, 2013.

[32] M. Richmond, "Vise: A virtual security testbed," *University of California, Santa Barbara, Tech. Rep*, 2005.

[33] 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. IEEE*, vol. 6, pp. 6–2671, IEEE, 2002.

[34] 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*, pp. 243–250, IEEE, 2003.

[35] G. Singaraju, L. Teo, and Y. Zheng, "A testbed for quantitative assessment of intrusion detection systems using fuzzy logic," in *Information Assurance Workshop, 2004. Proceedings. Second IEEE International*, pp. 79–93, IEEE, 2004.

[36] T. Benzel, "The science of cyber security experimentation: the deter project," in *Proceedings of the 27th Annual Computer Security Applications Conference*, pp. 137–148, ACM, 2011.

[37] O. Häggström, *Finite Markov chains and algorithmic applications*, vol. 52. Cambridge University Press, 2002.

[38] P. Bremaud, *Markov chains: Gibbs fields, Monte Carlo simulation, and queues*, vol. 31. Springer Science & Business Media, 1999.

[39] J. R. Norris, *Markov chains.* No. 2, Cambridge university press, 1998.

[40] C. M. Grinstead and J. L. Snell, *Introduction to probability.* American Mathematical Soc., 1997.

[41] R. Serfozo, *Basics of applied stochastic processes.* Springer Science & Business Media, 2009.

[42] A. Klenke, *Wahrscheinlichkeitstheorie*, vol. 1. Springer, 2006.

[43] U. Krengel, *Einführung in die Wahrscheinlichkeitstheorie und Statistik: Für Studium, Berufspraxis und Lehramt.* vieweg studium; Aufbaukurs Mathematik, Vieweg+Teubner Verlag, 2005.

[44] J. Gravner, "Lecture notes for introductory probability," January 2014.

[45] N. Kusolitsch, *Maß-und Wahrscheinlichkeitstheorie: Eine Einführung.* Springer-Verlag, 2011.

[46] C. H. Hesse, *Angewandte Wahrscheinlichkeitstheorie: eine fundierte Einführung mit über 500 realitätsnahen Beispielen und Aufgaben.* Springer-Verlag, 2013.

[47] N. Henze, *Stochastik für Einsteiger: eine Einführung in die faszinierende Welt des Zufalls.* Springer-Verlag, 2006.

[48] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *IP Operations Management, 2003. (IPOM 2003). 3rd IEEE Workshop on*, pp. 119–126, Oct 2003.

[49] J. Stearley, "Towards informatic analysis of syslogs," in *Cluster Computing, 2004 IEEE International Conference on*, pp. 309–318, Sept 2004.

[50] D. M. Ritchie, B. W. Kernighan, and M. E. Lesk, *The C programming language.* Bell Laboratories, 1975.

[51] R. Vaarandi, "SLCT version 0.05." `http://ristov.users.sourceforge.net/slct/`, 2003–2007.

[52] R. Vaarandi, "Mining event logs with slct and loghound," in *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, pp. 1071–1074, April 2008.

[53] P. Berkhin, "A survey of clustering data mining techniques," in *Grouping multidimensional data*, pp. 25–71, Springer, 2006.

[54] R. Xu, D. Wunsch, *et al.*, “Survey of clustering algorithms,” *Neural Networks, IEEE Transactions on*, vol. 16, no. 3, pp. 645–678, 2005.

[55] J. Hartung and B. Elpelt, *Multivariate Statistik: Lehr-und Handbuch der angewandten Statistik*. Oldenbourg Verlag, 2007.

[56] S. Guha, R. Rastogi, and K. Shim, “Rock: A robust clustering algorithm for categorical attributes,” in *Data Engineering, 1999. Proceedings., 15th International Conference on*, pp. 512–521, IEEE, 1999.

[57] A. Huang, “Similarity measures for text document clustering,” in *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*, pp. 49–56, 2008.

[58] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan, *Automatic subspace clustering of high dimensional data for data mining applications*, vol. 27. ACM, 1998.

[59] L. Parsons, E. Haque, and H. Liu, “Subspace clustering for high dimensional data: a review,” *ACM SIGKDD Explorations Newsletter*, vol. 6, no. 1, pp. 90–105, 2004.

[60] V. Ganti, J. Gehrke, and R. Ramakrishnan, “Cactus—clustering categorical data using summaries,” in *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 73–83, ACM, 1999.

[61] S. Goil, H. Nagesh, and 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*, pp. 443–452, 1999.

[62] C. C. Aggarwal, J. L. Wolf, P. S. Yu, C. Procopiuc, and J. S. Park, “Fast algorithms for projected clustering,” in *ACM SIGMoD Record*, vol. 28, pp. 61–72, ACM, 1999.

[63] D. J. Hand, H. Mannila, and P. Smyth, *Principles of data mining*. MIT press, 2001.

[64] J. Zobel, S. Heinz, and H. E. Williams, “In-memory hash tables for accumulating text vocabularies,” *Information Processing Letters*, vol. 80, no. 6, pp. 271–277, 2001.

[65] R. Vaarandi and K. Podiņš, “Network ids alert classification with frequent itemset mining and data clustering,” in *Network and Service Management (CNSM), 2010 International Conference on*, pp. 451–456, IEEE, 2010.

[66] R. Balakrishnan and K. Ranganathan, *A Textbook of Graph Theory*. Universitext (Berlin. Print), Springer New York, 2012.

[67] B. Everitt, S. Landau, M. Leese, and D. Stahl, *Cluster Analysis.* Wiley series in probability and statistics, Wiley, 2011.

[68] F. Skopik, G. Settanni, R. Fiedler, and I. Friedberg, "Semi-synthetic data set generation for security software evaluation," in *Privacy, Security and Trust (PST), 2014 Twelfth Annual International Conference on*, pp. 156–163, IEEE, 2014.

[69] P. Mantis, "Bug tracker development group. mantis bug tracker homepage," *World Wide Web, http://www. mantisbt. org*, 2015.

[70] F. Skopik, I. Friedberg, and R. Fiedler, "Dealing with advanced persistent threats in smart grid ict networks," in *Innovative Smart Grid Technologies Conference (ISGT), 2014 IEEE PES*, pp. 1–5, IEEE, 2014.

[71] I. Friedberg, F. Skopik, G. Settanni, and R. Fiedler, "Combating advanced persistent threats: from network event correlation to incident detection," *Computers & Security*, vol. 48, pp. 35–57, 2015.