TU WIEN Informatics

Resource-Efficient Log Analysis to Enable Online Anomaly Detection in Cyber Security

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Dipl.-Ing. Markus Wurzenberger
Matrikelnummer 00926105

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner

Diese Dissertation haben begutachtet:

Stefan Rass	Christopher Kruegel

Wien, 21. Jänner 2021

Markus Wurzenberger

TU WIEN Informatics

Resource-Efficient Log Analysis to Enable Online Anomaly Detection in Cyber Security

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Markus Wurzenberger
Registration Number 00926105

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner

The dissertation has been reviewed by:

Stefan Rass	Christopher Kruegel

Vienna, 21st January, 2021

Markus Wurzenberger

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

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Markus Wurzenberger

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.

Wien, 21. Jänner 2021

Markus Wurzenberger

Acknowledgements

First and foremost I would like to thank my supervisor Wolfgang Kastner for his comprehensive assistance in both technical and organizational aspects, and extensive feedback on my research. I would also like to thank Florian Skopik, who accepted me into his Cyber Security research group at the Austrian Institute of Technology (AIT) and provided me with daily advice and support. I would also like to mention my closest colleagues Giuseppe Settanni, Max Landauer and Georg Höld for their strong cooperation and excellent support over the years. Without their help it would not have been possible to achieve all the scientific achievements provided in my thesis.

Furthermore, I want to thank all my other coworkers for providing such a pleasant and positive work environment. This has never made it difficult to come to work and has led to good friendships.

I am also grateful to the Austrian Research Promotion Agency (FFG) and the European Commission (EC) for supporting my PhD research financially through the projects BAESE (FFG 852301), synERGY (FFG 855457), ECOSSIAN (EC FP7 607577), and GUARD (EC H2020 833456). These funding allowed me to publish my research results in scientific journals and on various conferences.

Finally, I want to thank my family and my friends who supported me through my whole life. Without them I would not be where I am today. My last words belong to my beloved girlfriend Rita, who has always been patient and supportive, as well as reminding me, especially when necessary, that there is a life beyond my research activities.

Kurzfassung

Die schiere Anzahl an unterschiedlichen Angriffsvektoren und die großen Mengen an Daten, die von Computersystemen produziert werden, machen es unmöglich, Netzwerkinfrastrukturen mit traditionellen Sicherheitsmaßnahmen wie Antivirus, Firewalls und signaturbasierten Intrusion-Detection-Systemen (IDS) abzusichern, welche meist nur die Erkennung bekannter Angriffe ermöglichen. Darüber hinaus erschweren Ende-zu-Ende-Verschlüsselung, Virtualisierung und Containerisierung die Überwachung und Analyse von Netzwerkverkehr. Daher untersucht diese Arbeit die Möglichkeit anhand von Anomalie-basierter Intrusion Detection textuelle Logdaten wie System Logs, Audit Logs, Web Logs und Application Logs zu analysieren. Die Dissertation identifiziert als Forschungslücken der Analyse von un- bzw. semi-strukturierten Logdaten unter anderem die fehlende Online-Analysefunktion und die oft ineffiziente Informationsverlust-behaftete Anomalieerkennung. Daher wird ein neuartiger inkrementeller Clustering-Ansatz vorgestellt, der durch hochleistungsfähige bioinformatische Anwendungen motiviert ist und die Online-Analyse großer Mengen an Logzeilen ermöglicht. Weiters wird ein zeichenbasierter Template-Generator entwickelt, der das Problem der Berechnung von Multi-line-Alignments für beliebige Strings löst und detaillierte Cluster-Beschreibungen liefert. Dies ermöglicht die Erstellung aussagekräftiger Templates für Logzeilen und kompensiert die Nachteile Token-basierter Templates, wie zum Beispiel fehlerhafte Verarbeitung von ähnlichen aber nicht identen Tokens und Nutzung von Wildcards zur Beschreibung großer Teile von Logzeilen. Aktuelle Logparser wenden Listen von regulären Ausdrücken oder Signaturen an und benötigen daher große Mengen an Ressourcen, um Logzeilen zu verarbeiten. Folglich werden einige Teile der Zeilen während des Parsens ausgelassen, was zu Informationsverlusten bei Anomalieerkennung führt. Um diese zu kompensieren und ein Ressourcen-sparendes detailliertes Echtzeit-Parsing zu ermöglichen, stellt diese Arbeit einen Parsergenerator vor, der baumartige Parser erzeugt, die die Komplexität des Parsens ohne Informationsverlust effizient reduzieren. Schließlich wird das Potenzial der entwickelten Algorithmen in drei Anwendungsfällen demonstriert. Zuerst wird ein Ansatz zur Zeitreihenanalyse vorgestellt, der den inkrementellen Clustering-Ansatz in Kombination mit Cluster-Evolution nutzt, um Frequenzanomalien zu erkennen. Danach wird ein Anomalie-Erkennungssystem beschrieben, das den baumartigen Parsergenerator anwendet, um Echtzeit Anomalieerkennung unter Verwendung minimaler Ressourcen zu ermöglichen. Abschließend wird ein neuartiges Konzept beschrieben, das automatische Bewertung, Vergleich, und Optimierung von IDS und deren Konfigurationen ermöglicht.

Abstract

The sheer number of different attack vectors and large amount of data produced by computer systems make it impossible to secure network infrastructures using traditional security measures such as anti-viruses, firewalls, and signature-based intrusion detection systems (IDS) that mostly allow detection of known attacks. Additionally, end-to-end encryption, virtualization and containerization make monitoring and analyzing network traffic non-trivial. Therefore, this thesis investigates the potential of anomaly-based intrusion detection that monitors textual log data, such as system logs, audit logs (syscalls), web logs (e.g., access logs), and application logs. The thesis identifies research gaps in state of the art log-based anomaly detection, including missing online analysis features and efficient log line parsing without loss of information, when analyzing un- and semi-structured log data. Furthermore, we propose a novel incremental clustering approach motivated by high-performance bio informatics tools that enables online analysis of large amounts of log lines. Moreover, we introduce a character-based template generator that solves the problem of computing multi-line alignments for arbitrary strings and provides detailed cluster descriptions. This enables the creation of meaningful log line templates that overcome the disadvantages of token-based templates, including handling of similar but not equal strings, and covering large parts of log lines with wildcards. State of the art parsers apply lists of regular expressions or signatures. Hence, they require large amounts of resources to process log lines and consequently remove large parts of log messages during parsing procedure, which leads to loss of information in the anomaly detection process. To overcome this weakness and enable detailed online log parsing requiring just a minimum amount of resources, the thesis proposes a parser generator that creates tree-like parsers, which effectively reduce complexity of parsing without information loss. Finally, we demonstrate the potential of the developed algorithms in three application cases. The first one introduces a time series analysis approach that uses the incremental clustering approach in combination with cluster evolution to detect frequency anomalies. Next, we describe a log-based anomaly detection system that applies the tree-like parser generator to enable online intrusion detection with a minimum amount of resources. Eventually, we propose a novel concept that enables automatic evaluation, comparison, and optimization of IDS and their configurations with respect to a specific network infrastructure.

Contents

CHAPTER 1

Introduction

1.1 Motivation

We observe a growing attack surface that cyber criminals actively exploit. In recent years, new forms of cyber attacks have emerged with an unprecedented sophistication level. Besides exigent and tailored Advanced Persistent Threats (APT) [Tan11], all business sectors suffer from attacks using techniques such as ransomware, dropper, backdoor, and credential stealer [Fir20]. Consequences of cyber attacks reach from financial loss, over sabotaged systems to stolen intellectual properties.

Additionally, systems have grown to a size and complexity in which their mode of operation is barely understandable any more, especially for chronically understaffed security teams. The combination of ever increasing exploitation of zero day vulnerabilities, malware auto-generated from tool kits with varying signatures, and the still problematic lack of user awareness is alarming. Consequently, signature-based intrusion detection systems (IDS), which look for the presence of known malware or malicious behavior studied in labs, do not seem fit for future challenges. Hence, new, flexibly adaptable forms of IDS, which require minimal maintenance and human intervention, and rather learn themselves what is considered normal in an infrastructure, are a promising means to tackle today's serious security situation [WSSF18].

The ongoing digitization is the main reason for the continuously growing amount of data produced by today's IT networks. However, the large amount of data, the mix of technologies and the emerging number of possible operation modes and system configurations are the reason why analyzing this data is a challenging big data problem that accounts for novel smart detection algorithms that can be deployed in any network environment and are capable of processing large amounts of data efficiently to enable online analysis [WSS18].

1.2 Problem statement and goal

Current IDS still mainly implement signature-based approaches and thus only allow to detect known attacks. However, due to the diversity of technologies and the sheer endless number of configuration parameters, networks are inherently different and produce highly diverse data. Consequently, they require a large number of signatures to prohibit all malicious behavior. Especially poorly documented software, as well as services and programs with small market shares lack support from vendors and security companies. There are no signatures available to feed IDS. Moreover, initial deployment and keeping signatures up-to-date are cumbersome tasks, due to highly frequent updates that lead to changes in collected data. A solution to this problem, are self-learning anomaly-based detection algorithms that use artificial intelligence and machine learning to learn a networks normal behavior and detect deviations from this baseline and consequently reveal anomalies that potentially relate to attacks.

Furthermore, state of the art IDS mostly analyze network traffic, i.e. pcaps from network taps and NetFlows. However, because of the emergence of end-to-end encryption and tunneling technologies, which avoid deep packet inspection, as well as virtualization and containerization, which lead to deployment of machines on the same hypervisor, network activity is hard to monitor. Log data, such as system logs, audit logs (syscalls), web logs (e.g., access logs), as well as application logs are an excellent source to either inspect system behavior on host side, in a container or on a central log storage .

Log data is the lowest common denominator of data that any piece of software can produce to inform about its operational state. Thus, log data is a key information source for many different applications such as intrusion detection, fault diagnosis, performance evaluation, predictive maintenance and network behavior analysis. Nowadays, all these techniques are applied in virtually any type of system, being Web-based systems, enterprise IT, Cyber Physical Systems (CPS), Industry 4.0, or Internet of Things (IoT). However, despite its broad application, there is no common standard for the structure and appearance of log data. Consequently, it is rather difficult to make data automatically accessible for further analysis with no or minimal manual effort. This leads to the following research problems [WLSK19]:

1. There exist many algorithms that implement anomaly detection. However, they are mostly only applicable to numerical or structured data, which does not apply to textual log data that is in best case semi-structured.

2. Many anomaly detection algorithms are not applicable for online analysis, because they either require knowledge of all data before analysis, because they do not implement single-pass procedures, or suffer from high complexity, i.e., implementations lack performance.

3. The generation of log parsers to make information accessible for anomaly detection algorithms inherits problems from signature generation and thus their definition

is time consuming and requires large amounts of resources. Furthermore, parsers that base on regular expressions are inefficient and due to the large amount of collected log lines and their complexity of $O(n)$, where n is the number of event types occurring in the collected data, do not allow online analysis, especially when resources are limited[1].

4. Furthermore, many parsers only allow to parse specific parts of a log line, for example, time stamps, host names and specific parts of log messages such as IP addresses, while neglecting the remaining information. This prohibits detection algorithms from consuming all information that a network's log data provides.

Thus, the goal of the thesis is:

> Development of log analysis algorithms that enable **fast processing** of log data and consequently implement **online anomaly detection** that requires a **minimum amount of resources** and **minimal effort for configuration and deployment** to carry out intrusion detection based on **self-learning**.

The problem statement and the overall goal imply the following research questions:

1. *RQ1:* How can semi-/unsupervised machine learning techniques be applied to semi-/unstructured textual log data to enable online anomaly detection?

2. *RQ2:* To what extent is it possible to describe the content of log data during normal system operation?

3. *RQ3:* How can log line parsing be optimized to enable online processing of log lines with minimal information loss when analyzing large amounts of data with limited resources?

4. *RQ4:* How can online log analysis algorithms be applied?

1.3 Contributions

Major parts of the thesis have been published in journal and magazine articles, conference papers, and book chapters. The following publications contributed to the remaining thesis:

[1]Depending on the complexity of the data, i.e. number of different event types occurring in the data, as well as length of the log lines, and level of granularity of parsers, the length of the list of signatures/regular expressions and the length of the single signatures/regular expressions prevent online parsing, specifically if available resources are limited.

1. **Wurzenberger M.**, Höld G., Landauer M., Skopik F., Kastner W. (2020): Creating Character-based Templates for Log Data to Enable Security Event Classification. 15th ACM ASIA Conference on Computer and Communications Security (ACM Asia CCS), October 05-09, 2020, Taipei, Taiwan. ACM. [WHL+20]

2. **Wurzenberger M.**, Landauer M., Skopik F., Kastner W. (2019): AECID-PG: A Tree-Based Log Parser Generator To Enable Log Analysis. 4th IEEE/IFIP International Workshop on Analytics for Network and Service Management (AnNet 2019) in conjunction with the IFIP/IEEE International Symposium on Integrated Network Management (IM), April 8, 2019, Washington D.C., USA. IEEE. [WLSK19]

3. Landauer M., **Wurzenberger M.**, Skopik F., Settanni G., Filzmoser P. (2018): Dynamic Log File Analysis: An Unsupervised Cluster Evolution Approach for Anomaly Detection. Elsevier Computers & Security Journal, Volume 79. November 2018, pp. 94-116. Elsevier. [LWS+18a]

4. Landauer M., **Wurzenberger M.**, Skopik F., Settanni G., Filzmoser P. (2018): Time Series Analysis: Unsupervised Anomaly Detection Beyond Outlier Detection. 14th International Conference on Information Security Practice and Experience (ISPEC), September 25-27, 2018, Tokyo, Japan. Springer LNCS.[LWS+18b]

5. **Wurzenberger M.**, Skopik F., Settanni G. (2018): Big Data for Cyber Security. In Encyclopedia of Big Data Technologies. Sakr, Sherif and Zomaya, Albert (Eds.) Springer International Publishing, 2019, Online ISBN: 978-3-319-63962-8. [WSS18]

6. **Wurzenberger M.**, Skopik F., Settanni G., Fiedler R. (2018): AECID: A Self-learning Anomaly Detection Approach Based on Light-weight Log Parser Models. 4th International Conference on Information Systems Security and Privacy (ICISSP 2018), January 22-24, 2018, Funchal, Madeira - Portugal. INSTICC. [WSSF18]

7. Friedberg I., **Wurzenberger M.**, Balushi A., Kang B. (2017): From Monitoring, Logging, and Network Analysis to Threat Intelligence Extraction. pp. 69-127. In Collaborative Cyber Threat Intelligence: Detecting and Responding to Advanced Cyber Attacks at the National Level. Editor: Florian Skopik Taylor & Francis, CRC Press, 2017, ISBN-10: 1138031828, ISBN-13: 978-1138031821. [FWABK17]

8. **Wurzenberger M.**, Skopik F., Landauer M., Greitbauer P., Fiedler R., Kastner W. (2017): Incremental Clustering for Semi-Supervised Anomaly Detection applied on Log Data. 12th International Conference on Availability, Reliability and Security (ARES), August 29 - September 01, 2017, Reggio Calabria, Italy. ACM. [WSL+17]

9. **Wurzenberger M.**, Skopik F., Fiedler R., Kastner W. (2017): Applying High-Performance Bioinformatics Tools for Outlier Detection in Log Data. 3rd IEEE International Conference on Cybernetics (CYBCONF), June 21-23, 2017, Exeter, UK. IEEE. [WSFK17]

10. **Wurzenberger M.**, Skopik F., Fiedler R., Kastner W. (2016): Discovering Insider Threats from Log Data with High-Performance Bioinformatics Tools. 8th ACM CCS International Workshop on Managing Insider Security Threats (MIST 2016) colocated with the 23rd ACM Conference on Computer and Communications Security (CCS), October 24-28, 2016, Vienna, Austria. ACM. [WSFK16]

11. **Wurzenberger M.**, Skopik F. (2016): The BAESE Testbed – Analytic Evaluation of IT Security Tools in Specified Network Environments. ERCIM News, Number 107, October 2016, pp. 51-52. ERCIM – The European Research Consortium for Informatics and Mathematics.[WS16]

12. **Wurzenberger M.**, Skopik F., Settanni G., Scherrer W. (2016): Complex Log File Synthesis for Rapid Sandbox-Benchmarking of Security- and Computer Network Analysis Tools. Elsevier Information Systems (IS), Volume 60, Aug./Sept. 2016, pp. 13-33. Elsevier. [WSSS16]

13. **Wurzenberger M.**, Skopik F., Settanni G., Fiedler R. (2015): Beyond Gut Instincts: Understanding, Rating and Comparing Self-Learning IDSs (Poster and Extended Abstract). International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA 2015), June 8-9, 2015, London, UK. C-MRIC. [WSSF15]

Publications 1, 2, 5, 6, and 8-13 all solely base on my, Markus Wurzenberger's, ideas. Additionally, I have been the main contributor to concepts, models, and texts of these publications. The research for publications 3 and 4 has been carried out in close collaboration with Max Landauer. Also the concepts, models, and texts for these two publications, have been developed in close collaboration of Max Landauer and me. Publication 7 is a book chapter to which I, Markus Wurzenberger, have been the sole contributor of the parts on log data, as well as on intrusion and anomaly detection. No other parts of this publication have been reused in the present thesis.

The thesis extends the state of the art by the following contributions:

1. **C1 - Log data clustering using high-performance bioinformatics tools** [WSFK16, WSFK17]**:** Clustering is a well-established method for grouping similar log lines and detecting outliers. However, most clustering approaches apply distance-based approaches, which suffer from poor runtime and require large amounts of resources to store distance matrices. Therefore, they are not able to process large amounts of data. Hence, the thesis proposes an approach for log line clustering that makes high-performance bioinformatics clustering tools applicable to computer log data. Bio-informatics algorithms implement incremental single-pass procedures that are capable of processing large amounts of data requiring an acceptable amount of resources. They apply distance-based clustering instead of density-based and thus do not rely on distance matrices. While log data and genetic sequences share similar properties, they posses significantly different encoding. Therefore, the

proposed bio-clustering approach introduces a procedure for re-coding log data to the alphabet of amino acids. Additionally, it demonstrates how to benefit from the functionalities that by default make use of genetic interrelationships of amino acids. Finally, the bio-clustering approach enables processing large amounts of log data in short time and provide a clustering approach that does not require any information on the input data in advance.

2. **C2 - Incremental clustering for online anomaly detection** [WSL+17], [LWS+18b, LWS+18a]**:** State of the art clustering algorithms only enable forensic analysis, because they usually have to pass over data more than once and are not able to process data online, i.e., in real time. The incremental clustering approach builds on the ideas of the bio-clustering algorithm. It implements a similar single-pass procedure, i.e., each log line has to be processed only once, but does not require re-coding. As a result, in opposite to the bio-clustering, it enables online log analysis, i.e. it allows detection of anomalies when they occur and not only forensic analysis. Therefore, it implements several smart filters to reduce the number of distance calculations and ensures high performance while simultaneously demanding a minimum amount of resources. Furthermore, the thesis demonstrates how the approach enables outlier detection and time series analysis (TSA) for anomaly detection.

3. **C3 - Character-based template generation** [WHL+20]**:** The application areas of log line templates are manifold and include providing information on the content of clustered log lines, parser and whitelist rule generation by recognizing static and variable parts of log lines, log line filtering and even counting. As a result many security applications such as security information and event management (SIEM) and IDS benefit from log line templates. However, defining templates manually is a cumbersome task, thus template generators are essential. State of the art template generators provide only token-based approaches, which have difficulties with optional log line parts, require pre-defined separators that depend on the processed data and omit highly similar but not equal strings. Character-based templates mitigate these disadvantages and additionally maintain a higher coverage of log lines, and thus provide more detailed information about the content of log lines, for example, within a cluster and enable analysis of strictly variable parts. However, for the efficient generation of character-based templates, computing multi-line alignments is essential, which still is an unsolved problem. This thesis proposes an incremental single-pass approach for generating character-based templates that significantly reduces the complexity for computing multi-line alignments.

4. **C4 - Tree-based log line parser generator** [WLSK19, WSSF18]**:** Log line parsing is a key element of log analysis. Log parsers dissect log lines, assign event types, recognize static and variable parts of log lines, and thus make information stored in log lines accessible for further analysis. Considering the growing amount and diversity of log data, defining and updating parsers is a tedious task. Moreover,

most parsers do not take all information stored in a log line into account, but neglect information towards the end of lines or parse only specific information, such as IP addresses to reduce the runtime of parsing [HZZL17]. Furthermore, most parser generators apply regular expression to parse log lines, which is highly inefficient and thus prevents online log analysis locally on hosts with low resources. Therefore, the thesis proposes a parser generator approach that allows to create tree-like parsers that enable high-performance and detailed log line parsing requiring a minimum amount of resources, because of reducing the complexity of parsing by implementing a tree-like parser structure.

Besides the four main contributions C1-C4, in course of the thesis three applications have been developed, which apply the thesis's main contributions:

1. **A1 - Time series analysis** [LWS+18b, LWS+18a]**:** The thesis proposes a TSA approach that makes use of the incremental clustering approach C1/C2. The TSA uses cluster evolution to match clusters of different time windows. Based on these, it specifies an ARIMA model that enables detection of anomalies, which reflect through changes in properties such as change of cluster size over time.

2. **A2 - Self-learning anomaly detection** [WSSF18]**:** The tree-like parser generator C3 contributes to the anomaly detection system AECID (Automatic Event Correlation for Incident Detection). AECID employs the parser generator to generate a tree-like parser that enables fast log line parsing and allows to efficiently access information in log lines for further analysis.

3. **A3 - IDS testbed** [WSSS16, WS16, WSSF15]**:** BAESE (Benchmarking and Analytic Evaluation of IDSs in Specified Environments) is an innovative and lightweight testbed concept. For generating realistic semi-synthetic test data sets based on small snippets of real log data it either can use a combination of clustering C1/C2 and template generation C3 or a parser generator C4. Finally, it applies a feedback loop to assess, compare and optimize IDS.

1.4 Organization and methodology of the thesis

Figure 1.1 depicts the connections between the four main contributions C1-C4 (rectangles with rounded corners) and the three applications A1-A3 (diamonds). First, chapter ❷ summarizes relevant state of the art and background of the thesis. It includes the topics log data, intrusion detection systems, anomaly detection, which is the detection method the thesis focuses on. Furthermore, it covers clustering, which is a broad topic in the thesis, and template and parser generation. Next, Ch. ❸ introduces the bio-clustering approach (C1) that makes high-performance bioinformatics algorithms applicable to computer log data. Subsequently, Ch. ❹ proposes the incremental-clustering procedure (C2) that builds upon the bio-clustering and eliminates the dependency on the re-coding required

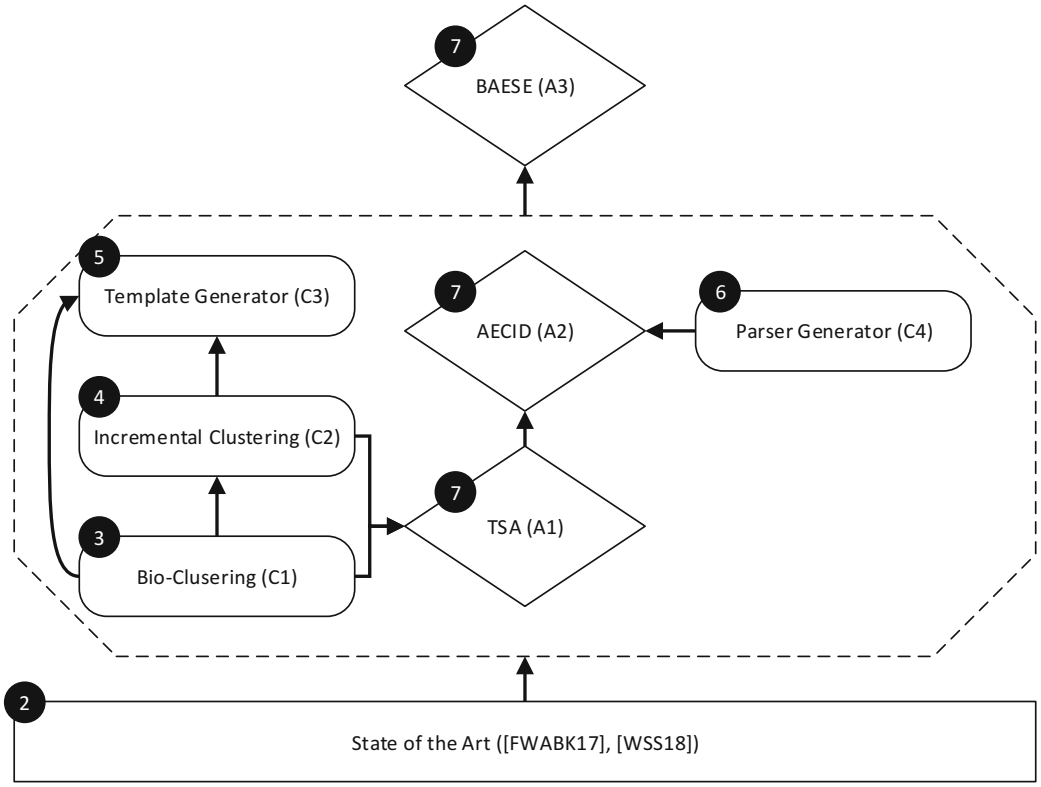


Figure 1.1: Organization of the thesis: Chapter ⓧ, Contribution (Cx) and Application (Ax).

to make log data readable for bioinformatics algorithms. Furthermore, it enables online processing of log data and thus makes real time anomaly detection possible. However, both the bio-clustering and the incremental clustering are missing the functionality to provide meaningful descriptions of the content of the log lines within a cluster as well as templates that enable, for example, rule and parser generation for support of further analysis. Chapter ❺ provides a character-based template generator approach (C3) that solves the problem of efficiently computing multi-line sequence alignments. Furthermore, it out-performs token-based template generators with respect to detail of description and log line coverage. Nevertheless, log line parsing using lists of template-based regular expressions is inefficient, due to high complexity, why often less accurate parsers are used that omit major parts of log lines. Therefore, the thesis proposes a novel parser generator that creates tree-like parsers (C4), which reduce the complexity of parsing and thus enable only log analysis demanding a minimum amount of resources. Chapter ❼ provides three applications that build upon the previously proposed approaches. The first one implements TSA (A1) that makes use of bio-clustering and incremental clustering,

as well as cluster evaluation to enable detection of frequency anomalies. Next, the chapter introduces the log-based anomaly detection system AECID (A2) that uses the developed parser generator to autonomously learn log parsers. Finally, the BAESE testbed (A3) applies the clustering approaches and the template generator as well as the parser generator to create semi-synthetic log data for testing IDS. Furthermore, the BAESE testbed provides an approach for evaluating, comparing and optimizing anomaly detection approaches, such as the TSA approach and AECID. Chapter ❽ concludes the thesis.

The process of conducting the thesis started with a literature research to define the state of the art and identify research gaps in the area of log data analysis, specifically anomaly detection in log data. The results of this procedure have been the research questions and the goal postulated in Sec. 1.2, as well as the background and state of the art summarized in Ch. 2. To develop the approaches proposed in Chapters 3, 4, 5, and 6, we started with outlining the research gaps and clearly specifying the planned contributions. After this, we designed a concept, described the detailed model of the approach, implemented a research prototype and evaluated the approach. Finally, we searched for applications that can be supported by the developed approaches and demonstrated the relevance of the thesis's results (see Ch. 7).

CHAPTER 2

Related work and background

The topic of the thesis involves a number of different technical aspects. Therefore, the related work covers the following topics:

- *Log data* is the data source for log analysis. It protocols a computer system's current state in human readable textual format.
- *Intrusion detection systems (IDS)* are used to detect attacks and adversaries in computer systems and networks.
- *Anomaly detection* is a method applied for intrusion detection that bases on white-listing and applies machine learning techniques.
- *Clustering* is a method from machine learning that groups data entities with similar properties. It can be applied for anomaly detection in form of outlier detection and time series analysis.
- *Template and parser generators* are required to create meaningful descriptions for log line clusters and to generate parsers that are needed to classify log events and dissect log lines for further analysis.

The structure of the related work originates from the different contributions of the thesis and the applications it introduces (see Ch. 1 and Fig. 1.1). First, the chapter summarizes background on log data, IDS and specifically anomaly detection, which is the primary use case of the thesis. The remaining chapter focuses on the background and related work of the actual contributions of the thesis, which includes clustering, as well as template and parser generation. Major parts of the remaining chapter have already been published in [WSS18] and [FWABK17].

2.1 Log data

Data logging has a widespread application area in the ICT sector. Log data is an important source for system monitoring [HA93], which comprises acquisition of data and knowledge. Furthermore, log data is investigated in course of digital forensics [Rag13], which is applied, for example, after an attack was detected to investigate its origin and find out information about the attacker and the purpose of the attack. Database logs can be used to back up and restore database content in case of a system crash or a destruction caused by an unauthorized access violation [FHH+11, FKS+12, WSSS16]. Firewall logs, if implemented firewall rules are working properly, allow to detect malicious activities such as multiple unsuccessful attempts to overcome the firewall, which are indicating an attacker that tries to intrude a network. Also suspicious outgoing connections might be an indication that, for example, a malware is used to launch an attack.

Log data contains automatically generated traces about all processes of services and components of an ICT network. Thus, it protocols all events occurring in such networks. Log data is usually represented in human-readable text format. This makes it easy to access the provided information. Other data sources, such as network packets, require time-, computational- and resource-intensive preprocessing before analysis, due to encryption. Thus, log data is a valuable source for cyber security analysis tools, such as IDS. The detail-level of information provided by log data depends on the configuration of the logging [Ger09]. One drawback is that usually only logs of lower severity levels, i.e. warning and error logs, are stored and used for security analysis. But to carry out extensive analysis, verbose logging on informational or debugging severity level is required, which, however, often is not the case. Reasons are that it is a resource consuming task and produces large amounts of data that have to be stored [CSP12].

By default, log files are usually stored as 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 too. A single log line usually consists of a time stamp and a protocolled event. The time stamp holds the information when the logged event occurred. Depending on the configuration of the logging it provides information about the day, the month and the year, as well as the time when the log line was produced. The event describes a process taking place in an ICT network, a network connection, or any other action carried out by a user or a program.

Because of the growing digitalization in recent years, the amount of produced log data is increasing exponentially. Thus, not only large scale networks account for suitable log management solutions that can handle this large amount of data. Log management [CSP12] comprises collecting logs, storing log data, analyzing log data, as well as searching and reporting log data.

Logging frameworks, which aid the implementation of proper logging support in new software products, exist for most programming platforms. It requires three components to establish logging: a logger, a formatter and a handler. The logger is responsible for

collecting the information that should be logged. The configuration of the logger defines at which level of detail information is stored in log files. This so-called severity level can be defined for each logger. Based on the configuration of the severity level, the logging framework decides whether a log message is stored in a log file or not. After the logger forwarded the log information to the logging framework, the formatter takes the provided object, which is normally represented as a binary and converts it into a string. Finally, the handler, which listens for log messages at or above the defined severity level, displays the resulting log line in a console, writes it to a file, or forwards it to another application [CSP12, YPZ12].

The common standard to store and transmit log messages is syslog [Ger09]. The key advantage of the syslog protocol is that a wide range of devices and network components support it. Its primary use is to send log messages to a centralized location, a so-called syslog server. Centralized logging makes log management much easier and simplifies correlating information collected at different locations of a network. Therefore, logging is an effective source to detect malware, malicious system behavior and invaders, because log data provides detailed information about an ICT network and its processes.

While the information that log data carries is essential for detecting security incidents and invaders, it might also provide an attacker knowledge about an ICT network. Especially in light of advanced persistent threats (APT) [Tan11], the (stealthy) passive phase can be used to learn about the logging mechanisms in place to manipulate it later on and to hide the active phase of the attack from the log. Therefore, it is indispensable to protect the logging from manipulation. A simple solution is offered by digital signatures that can be added to new log entries; however, resource constraints (space and computational power, such as in sensor networks) may practically prohibit such a solution. Centralized logging solutions have advantages regarding security compared to decentralized logging solutions: (i) Only one node has to be secured. (ii) If an attacker, for example, takes over a Web server, he can just manipulate log lines locally, but not the centrally stored version.

Among its advantages, the syslog standard also comprises some drawbacks. First of all, syslog does not define a standard how the messages have to be formatted. The only requirement is that it starts with a timestamp that is followed by the hostname. The rest of the log line can be chosen freely. This results in inconsistency, since every developer can define his own log message format. Furthermore, syslog by default uses the User Datagram Protocol (UDP) [Pos80] to transmit log messages over the network [Okm09]. Since UDP is connectionless, log messages can be lost because of network congestion and packet loss. Rsyslog [Ger10] extends the syslog standard and uses the more reliable Transmission Control Protocol (TCP) for transmitting log lines [GL12].

Log data is a suitable source to detect attacks online, because detailed information about the events occurring in an ICT network are available in easy accessible text format. Therefore, no preprocessing to decode the information stored in log data is required, which enables processing the data online. To get detailed information from network traffic, information stored in network packets has to be analyzed, which usually

requires computational preprocessing and therefore makes this kind of data unsuitable for online attack detection. Hence, for this purpose readable header information of network traces, which is stored in text format, can be analyzed instead of encrypted parts. In network traffic data, only attacks that violate rules or normal system behavior on the communication layer are visible. In opposite to log data, network traffic often is not controlled internally, which makes for example insider threats invisible for security tools monitoring network traffic data. While log data provides detailed information about events occurring in an ICT network, including also information about which connections have been established and why, network packet data include also the content that have been sent. But, this information might be encrypted and therefore not accessible. Another criteria, which data is analyzed, is the availability of the data. While, for example, an Internet provider can monitor the network traffic occurring in his network infrastructure, he has no access to log data produced by clients using the Internet. On the other hand, for an organization the bandwidth of their network intrusion detection systems (NIDS) might be insufficient to monitor the whole network traffic and therefore the log data produced on their client machines might be more valuable.

2.2 Intrusion detection systems

Like other security tools, intrusion detection systems (IDS) aim to achieve a higher level of security in ICT networks. Their primary goal is to timely and rapidly detect invaders, so that it is possible to react quickly and reduce caused damage. In opposite to Intrusion Prevention Systems (IPS) those are able to actively defend a network against an attack, IDS are passive security mechanisms, which only allow to detect attacks without setting any counter measurements automatically [WM12]. The remaining thesis only discusses IDS [WSSF18].

In 1980, James Anderson was one of the first researchers, who indicated the need of IDS and contradicted the common assumption of software developers and administrators that their computer systems are running in 'friendly', i.e. secure, environments [Jam80]. While companies started to invest large amounts of resources into cyber security, the awareness of their employees is still rather low, because most attackers still use phishing, skimming and infected devices such as USB sticks to exploit human negligence to get unauthorized access to companies' ICT networks. Thus, since Anderson published his report on IDS a lot of research in this area has been done until today [SM08, LRLLT13, GTDVMFV09, WSS18, WSSF18].

IDS can be roughly categorized as host-based IDS (HIDS), network-based IDS (NIDS) and hybrid or cross-layer IDS [Vac13, SM08, WSS18, WSSF18]:

- **HIDS** are the initial form of IDS and have been invented for the purpose of securing military mainframe computers. Similarly to simple security solutions such as anti-viruses, this sort of IDS has to be installed on each system (host) in a network that should be monitored. While HIDS deliver specific low-level information about an

HIDS	NIDS	Hybrid/Cross-layer IDS
• Installed on every system (host) • Allow comprehensive monitoring of a host • Detailed information about an attack • Source: Log data, registry, file system, etc. • When host is compromised also the HIDS is	• Monitor and analyze the network traffic of a whole network • Might create a bottleneck • Single sensor-node is sufficient for network • Drawbacks: overloaded bandwidth, encryption of payload • Source: Network traffic	• Combine many different methods • Provide a management framework that combines HIDS and NIDS • Reduce drawbacks and make use of advantages • Aim at maximizing the available information

Table 2.1: Comparison of HIDS, NIDS, and Hybrid/Cross-layer IDS.

attack and allow comprehensive monitoring of a single host, they can be potentially disabled by, for example, a Denial of Service (DoS) attack, because if a system is once compromised also the HIDS is.

- **NIDS** monitor and analyze network traffic of a whole network. The optimal application of NIDS would be monitoring inbound as well as outbound traffic. However, this might create a bottleneck and slow down the network. For monitoring a whole network with an NIDS, already one single sensor-node is sufficient and the functionality of this sensor is not affected, if one system of the network is compromised. A major drawback of NIDS is that if the NIDS's bandwidth is overloaded, a complete monitoring cannot be guaranteed, because some network packets and therefore possibly essential information might be dropped.

- **Cross-layer and hybrid IDS** combine different methods. Hybrid IDS usually provide a management framework that combines HIDS and NIDS to reduce their drawbacks and make use of their advantages. Cross-layer IDS aim to maximize the available information and therefore raise the detection capability to an optimum and minimize the false alarm rate at the same time. Therefore, various data sources, such as log data and network traffic data, can be used for intrusion detection. Especially light-weight solutions that work resource-efficient are required. A high data throughput is important to enable online analysis and evaluation of the collected information and thus timely detection of attacks and invaders.

Table 2.1 compares HIDS, NIDS, and Hybrid/Cross-layer IDS [WSS18]:

There exist generally three methods that are used in IDS: (i) signature-based detection (SD), (ii) stateful protocol analysis (SPA), and (iii) anomaly-based detection (AD)

Signature-based (knowledge)	**Anomaly-based (behavior)**	**Statefulf protocol analysis (specification)**
• Predefined signatures and patterns • Simplest method • Effective for detecting known attacks • Ineffective against unknown/evasions of attacks • Hard to keep up to date • Time consuming maintenance	• Use baseline of normal system behavior • Effective for detecting unknown/new attacks • Less dependent on monitored system • Often unavailable during building behavior profiles • High false positive rate (FPR)	• Trace protocol states • Distinguish unexpected sequences of commands • Unable to inspect attacks looking like benign protocol behavior • Adopt vendor developed profiles

Table 2.2: Comparison of SD, AD, and SPA.

[LRLLT13].

- **SD** uses predefined signatures and patterns to detect attackers. This method is simple and effective for detecting known attacks. The drawbacks of this method are: It is ineffective against unknown attacks or unknown variants of an attack, which allows attackers to evade IDS based on SD easily. Furthermore, since the attack landscape is rapidly changing, it is difficult to keep signatures up-to-date and thus maintenance is time consuming [WM12].
- **SPA** uses predetermined profiles that define benign protocol activity. Occurring events are compared against these profiles to decide if protocols are used correctly or not. IDS based on SPA track the state of network, transport, and application protocols. They use vendor-developed universal profiles, and therefore rely on their support [SM07].
- **AD** approaches learn a baseline of normal system behavior, a so-called ground truth. Against this ground truth all occurring events are compared to detect anomalous system behavior. In opposite to SD and SPA based IDS, AD based approaches allow detection of previously unknown attacks. A drawback of AD based IDS is a usually high false positive rate [GTDVMFV09].

Table 2.2 compares the three different IDS methods. SD and SPA can only detect previously known attack patterns using signatures and rules that describe malicious events and thus are also called blacklist approaches. AD approaches are more flexible and are also able to detect novel and previously unknown attacks. They permit only

normal system behavior and therefore are also called whitelist approaches. While blacklist approaches are usually easier to deploy, they depend on the support of vendors. They mostly cannot be applied in legacy systems and systems with small market shares, those are often poorly documented and lack of vendor support [WSS18, WSSF18].

2.3 Anomaly detection

The rapidly changing cyber threat landscape accounts for flexible, self-learning and self-adaptive IDS approaches. One solution are self-learning AD based approaches [CBK09]. These self-learning solutions usually learn during a training phase a baseline of normal system behavior that then serves as ground truth to detect anomalies that expose attacks and especially invaders. Generally, there are three ways how self-learning AD can be realized [GU16, WSS18, WSSF18]:

1. **Unsupervised**: This method does not require any labeled data and is able to learn to distinguish normal from malicious system behavior during the training phase. Based on the findings, it classifies any other given data during the detection phase.

2. **Semi-supervised**: This method is applied when the training set only contains anomaly-free data and is therefore also called 'one-class' classification.

3. **Supervised**: This method requires a fully labeled training set containing both normal and malicious data.

These three methods do not require active human intervention during the learning process. While unsupervised self-learning is entirely independent from human influence, for the other two methods the user has to ensure that the training data is anomaly free or correctly labeled. Consequently, the previously described methods can be categorized as unsupported self-learning approaches [WSS18, WSSF18].

Using completely unsupported self-learning raises some challenges. While providing training data for unsupervised self-learning is rather easy and does not require any preprocessing of the data, this approach might learn malicious system behavior as normal. Semi-supervised approaches try to avoid this problem by using anomaly-free training data. Applying these methods raises the problem of obtaining training data that guarantees to be anomaly-free. Retrieving such a dataset from a running productive system is usually difficult, because organizations are often not aware of malicious activities in their ICT network. Also for self-learning based anomaly detection it is true that the more information is provided during the training phase, the more accurate the system works later while an ICT network is monitored. Thus, supervised self-learning approaches provide the most detailed ground truth. But providing suitable training data sets for specific networks is time and resource consuming.

To avoid the mentioned drawbacks, supported self-learning approaches can be applied, where also system administrators can influence the training phase. This means for example, when an event is occurring for the first time and therefore is not part of the normal system behavior and as a consequence is classified anomalous, the administrator can decide if the event comprises an anomaly or if it is a false alarm and the event should be considered as normal system behavior in the future. Furthermore, self-learning can be used to constantly adapt the baseline, which describes the normal behavior and keep it up-to-date, when, for example, new devices are added to or removed from a network, or when the used software changes because of system updates.

Aside from all the advantages of AD based detection methods, they have also been shown to be vulnerable to a certain type of attack [SLJ+15]: When using carefully manipulated input samples, i.e., adversarial examples, a threat actor can circumvent detection or cause erroneous predictions (high-confidence misclassification). AD based methods are especially affected by poisoning attacks, where attackers try to manipulate the training phase in a way that during the detection phase their attacks are accepted as benign system/network behavior [KL10].

There exist many machine learning [WFHP16] algorithms to implement self-learning AD that can be applied for that task. Methods that are used for machine learning in cyber security are, for example [BG16, WSS18]:

- **Clustering:** Clustering enables grouping of unlabeled data. It is often applied to detect outliers and forms the foundation for generating log parsers that define a system's normal behavior [XW05, HZH+16, LSWR20].
- **Artificial neural networks (ANN):** Input data activates neurons (nodes) of an artificial network, inspired by the human brain. The nodes of the first layer pass their output to the nodes of the next layer, until the output of the last layer of the artificial network classifies the monitored ICT networks' current state [Can98].
- **Bayesian networks:** Bayesian networks define graphical models that encode the probabilistic relationships between variables of interest and can predict consequences of actions [H+98].
- **Decision trees:** Decision trees have a tree-like structure, which comprises paths that lead to a classification based on the values of different features [SL91].
- **Hidden markov models (HMM):** A Markov chain connects states through transition probabilities. HMM aim at determining hidden (unobservable) parameters from observed parameters [BE67].
- **Support vector machines (SVM):** SVM construct hyperplanes in a high- or infinite-dimensional space, which then can be used for classification and regression. Thus, similar to clustering, SVM can, for example, be applied for outlier detection [SC08].

2.4 Clustering

Clustering [XW05, Ber06, LSWR20] is a data analysis method from machine learning that is used in many different areas. The goal of clustering is to group data objects in clusters $C_1, C_2, \ldots C_n$, with $n \in \mathbb{N}$ the number of clusters, where objects within a cluster are more similar to each other than to elements of other clusters. In the remaining thesis, we refer to the set of clusters $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ as clustermap.

There exist many different techniques to carry out clustering on computer log data: distance-based (e.g., [DC15]), density-based (e.g., [Vaa03]), neural networks (e.g., [KFE14]), partitioning (e.g., [MZHM09]), longest common substring (e.g., [TLP11]), binary/source code analysis (e.g., [ZZH17]), genetic algorithms (e.g., [MPB+18]), frequent itemset mining (e.g., [Vaa04]), statistical modeling (e.g., [KIM+14]), graph community extraction (e.g., [LTPM17]), and heuristics (e.g., [JHHF08]).

The remaining thesis focuses on distance-based and density-based clustering approaches. When working with numerical or categorical data, properties of data entities are described by single values or a vector, which makes comparison easy. For numerical values and vectors, distances such as the Euclidean distance Eq. (2.1) can be applied. Categorical data can be transformed into binary vectors using one-hot encoding [HH10], which enables the application of the cosine similarity Eq. (2.2) [Hua08] to compute the similarity of two data entities.

$$d(a,b) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2} \tag{2.1}$$

$$s(a,b) = \cos(a,b) = \frac{\vec{a} \cdot \vec{b}}{\|a\|\|b\|} \tag{2.2}$$

When applying comparison-based algorithms to log lines, we distinguish between character-based and token-based approaches. In case of character-based comparison, distances, such as the Levenshtein distance [Lev66] that counts the number of edits which are required to transform one string into another one, are applied. Character-based approaches compare strings symbol by symbol. On the other hand, token-based approaches, such as SLCT [Vaa03], first split the log lines into a list of tokens, i.e., they split each string that represents a log line into substrings at predefined delimiters, such as white-spaces, semi-colons or brackets. Afterwards, the comparison is carried out. This method is often applied in density-based algorithms. While, character-based approaches provide a more accurate comparison, they are usually worse in runtime performance in opposite to token-based algorithms, because of the high computational complexity of character-based string metrics. While token-based approaches recognize tokens that differ only in a single character as completely different, although they might have the same meaning,

character-based approaches are more sensitive. Hence, they consider tokens as similar that only differ in a small number of characters, if they can be, for example, spelled differently, such as words like `php-admin`, `PHP-Admin` and `phpadmin`.

2.5 Template and parser generators

The following section was partly published in [WLSK19] and [WHL+20]. Template and parser generators are strongly related technologies. Templates are patterns that describe log line event types using their static parts and replacing variable parts with wildcards. Log line parsers apply such patterns to classify log lines and dissect them to make their content accessible for further analysis. Due to the fact that dealing with massive amounts of and fast changing log messages, it is important to automatize the process of generating log data parser.

Currently most template and parser generator algorithms implement token-based approaches. Furthermore, they often base on clustering or could be applied for clustering, which strongly connects those two topics. One of the first algorithms designed for log clustering using templates was SLCT [Vaa03]. The algorithm thereby pursues a density-based, clustering, i.e., frequent words on certain positions in the log lines are considered as fixed, while infrequent words are considered as variables. LogHound [VP15] uses a similar approach, but is resilient to shifts of word positions. Another density-based algorithm is CAPRI [ZMP+13], which uses bit-wise token matching and also considers the type of characters that make up the tokens for generating frequent patterns [WLSK19].

Other than density-based approaches, distance-based approaches group similar log lines and extract signatures from the resulting clusters. Thereby, the distance-function is often based on the number of matching tokens in every two log messages that are compared. The signatures may then be generated by different approaches, e.g., merging the log messages using string alignment [HDX+16, NJCY14], building parser trees based on the number of tokens in the log lines [HZZL17], and replacing the words that diverge in the grouped sets of log lines with wildcards that represent variable nodes [Shi16, Miz13]. There also exist approaches that additionally rely on domain knowledge and require manual effort. In particular, it is necessary to define grammars that describe the usual structure of log lines for differentiating between variable and fixed tokens [TL10, TBG+11, WLSK19].

Another method for generating log signatures is partitioning. Thereby, the groups of log lines are iteratively divided into subgroups by splitting at appropriate token positions. Usually, heuristics are used to determine the position of the split, e.g., searching for the token position that contains the most constants [GCTMK11]. Other approaches based on this principle are IPLoM [MZHM09], which in a final step also searches for relationships between tokens referred to as bijections, and POP [HZH+17], which computes the longest common subsequence (LCS) between the strings when merging log events [WLSK19].

Finally, several recent approaches use neural networks for signature extraction. Since log messages are usually designed to be human-readable and thus related to natural

language, it stands to reason to adapt existing concepts such as conditional random fields [KFE14] or long short-term memory (LSTM) [TMP17] for separating log messages into static and variable parts. However, one problem of these approaches is that they are supervised, i.e., require labeled training data. One solution for this problem is posed by RNN auto-encoders, which offer a possibility of unsupervised learning using neural networks [MP17].

The foundations for character-based templates are string metrics that allow to compare two strings character-wise. Some well-known examples for such string metrics are the Levenshtein distance [Lev66], the Needleman-Wunsch algorithm [NW70], the Smith-Waterman algorithm [SW+81], and different versions of the Jaro distance [Jar89]. For example, by reverting the procedure of the Levenshtein distance and leveraging the computed score-matrix, it is possible to calculate an alignment [JM09]. Other algorithms, such as the Needleman-Wunsch and Smith-Waterman, provide an alignment at once, but suffer from a higher computational complexity due to a more complex scoring function. However, all these algorithms are only able to provide pairwise sequence alignments. For generating templates for a group of log lines, it requires algorithms that are able to efficiently calculate mulit-line alignments. However, using common character-based string metrics, the computational complexity to achieve a mulit-line alignment cannot be lower than $O(n^m)$, where n is the length of the shortest considered log line and m is the number of considered lines [WHL+20].

In the area of bioinformatics, there exist several highly efficient algorithms, such as MAFFT [KKTM05], M-Coffee [WOHN06] and PROMALS [PG07], that allow to compute so-called multiple sequence alignments [Not07]. These algorithms mostly base on previously mentioned methods for calculating pairwise sequence alignments. Due to the fact that they use scoring systems and heuristics that make use of evolutionary relationships between amino acids, they can only be applied to strings that represent biological sequences such as DNA and RNA, and not to any other type of string [Not07]. Therefore, efficiently generating a template for a group of similar strings remains an unsolved problem. Furthermore, it is not expedient to calculate the optimal alignment for a group of strings, because it would be computationally too expensive. Hence, it is only feasible to approximate the optimal template [WHL+20].

CHAPTER 3

Application of high-performance bioinformatics tools to enable computer log data clustering

This chapter proposes an approach that applies high-performance algorithms from bioinformatics for clustering textual computer log data. In the following, we provide a proof of concept for this *bio-clustering* approach that improves log data analysis in various aspects. We demonstrate that bio-informatics methods are applicable for the purpose of clustering log data and therefore motivate the further development of these approach. Since this chapter contains preliminary research that is extended in later chapters, as evaluation it provides a proof of concept in the form of an illustrative example to motivate further research in this directions. Major parts of this chapter have been published in [WSFK16], [WSFK17], and [WSFK21].

Clustering is a very effective technique to describe a computer systems and networks system behavior by grouping similar log lines in clusters. Furthermore, it allows to periodically review rare events (outliers) and checking frequent events by comparing cluster sizes over time (e.g., trends in the number of requests to certain resources). Hence, clustering supports organizations to have a thorough understanding about what is going on in their network infrastructure, to review log data and to find anomalous events in log data. Existing tools are basically suitable to cover all these requirements, but they still show some essential deficits. Most of them, such as SLCT [Vaa03] implement token-based matching of log entries. Hence, they identify terms and words that can be spelled differently and often only differ in one character, such as `php-admin` and `phpadmin`, or similar URLs, as entirely different. Thus, the implementation of character-based matching with comparable runtime performance to token-based matching is necessary. Furthermore, existing traditional tools applied for log line clustering are usually not able to process large log files online and therefore are only applicable for forensic analysis,

but not for online anomaly detection, which enables detection of attacks, when they take place.

In the domain of bioinformatics, various methods have been developed to analyze and study the similarity of biologic sequences (DNA, RNA, amino acids), group similar sequences and extract common properties [WZTS06]. The algorithms to implement these features need to fulfill some strong requirements:

- *Adequate digital representation:* Biologic sequences must be represented as data streams in an appropriate format, i.e., no information must be lost, but the format should be as simple as possible.
- *Dealing with natural variations:* The dependency between a segment of a sequence and a certain biologic function (implemented by this segment) is sometimes not strict (or obvious). This means natural variations need to be accounted for and a certain degree of fuzziness in the input data accepted.
- *Dealing with artificial inaccuracies:* The process of recording long and complex biologic sequences causes inevitable inaccuracies and small errors. The negative influence of those (artificially introduced) variations in the following analysis phase should, however, be kept to an absolute minimum.
- *Dealing with massive data volumes:* Since biologic data sequences are (even to represent simple functions) very complex, algorithms need to deal with large amounts of data usually by (i) being scheduled in parallel and (ii) accepting certain inaccuracies caused by this non-sequential processing.

In general, all these requirements also apply to modern log data analysis as (i) data needs to be processed extremely fast (this means depending on the application approximately in real time); (ii) the process needs to accept certain inaccuracies and errors that occur due to conversion errors from varying character encodings, and slight differences in configurations and output across software versions; and if possible (iii) data analysis needs to be scheduled in parallel in order to scale. Furthermore, these tools aim at processing character sequences without taking into account their semantic meanings.

Consequently, if mentioned tools are not applied to biologic sequences but to re-coded (converted) digital sequences, such as log data (or even malware code), all of the special properties of these algorithms can be harnessed directly, without the need to design and implement complex tools again. The remaining chapter provides the following contributions:

(i) A method for re-coding log data into the alphabet used for representing canonical amino acid sequences, which enables the application of high-performance bioinformatics tools in the domain of log data analysis.

(ii) A detailed concept for the application of clustering algorithms from the domain of bioinformatics to log data to define a baseline of normal system behavior, which enables outlier detection and time series analysis to discover anomalous and erratic behavior, as well as trends and variations of behavior over time in computer networks.

(iii) An investigation of the applicability and feasibility of the proposed approach in a real setting by simulating a scenario of an attack by an insider threat, as well as an analysis and interpretation of the evaluation results.

The remaining chapter structures as follows: Section 3.1 introduces the concept of applying bioinformatics tools for log line clustering. Next, Sec. 3.2 describes how log lines can be re-coded into the alphabet of amino acids, and Sec. 3.3 depicts the clustering process. Section 3.4 provides an evaluation of the proposed approach in course of a proof of concept. Finally, Sec. 3.5 provides an outlook, describing required future work.

3.1 Concept for applying bioinformatics clustering tools to computer log data

The following section defines the concept that enables the application of high-performance bioinformatics tools for clustering computer log data. The proposed modular approach comprises several steps from re-coding log data to the alphabet used for describing amino acid sequences, towards interpretation and analysis of the output for anomaly detection:

(i) collect log data,

(ii) homogenize log data,

(iii) re-code and format log data,

(iv) compare log lines pairwise regarding similarity,

(v) cluster log lines,

(vi) re-translate data,

(vii) detect outliers and analyse time series.

Figure 3.1 visualizes the concept, the proposed bio-clustering approach bases on. It can be roughly split into three blocks which are sequentially repeated. Block I covers the process of re-coding log data into a format, which can be processed by bioinformatics tools. First, in step (i) log data is collected from different sources of the considered network. When analyzing log data from different sources usually the data shows some differences in the format. For example, main properties such as time stamps are represented in different formats. Therefore, step (ii) is required to homogenize the data. A uniform time stamp

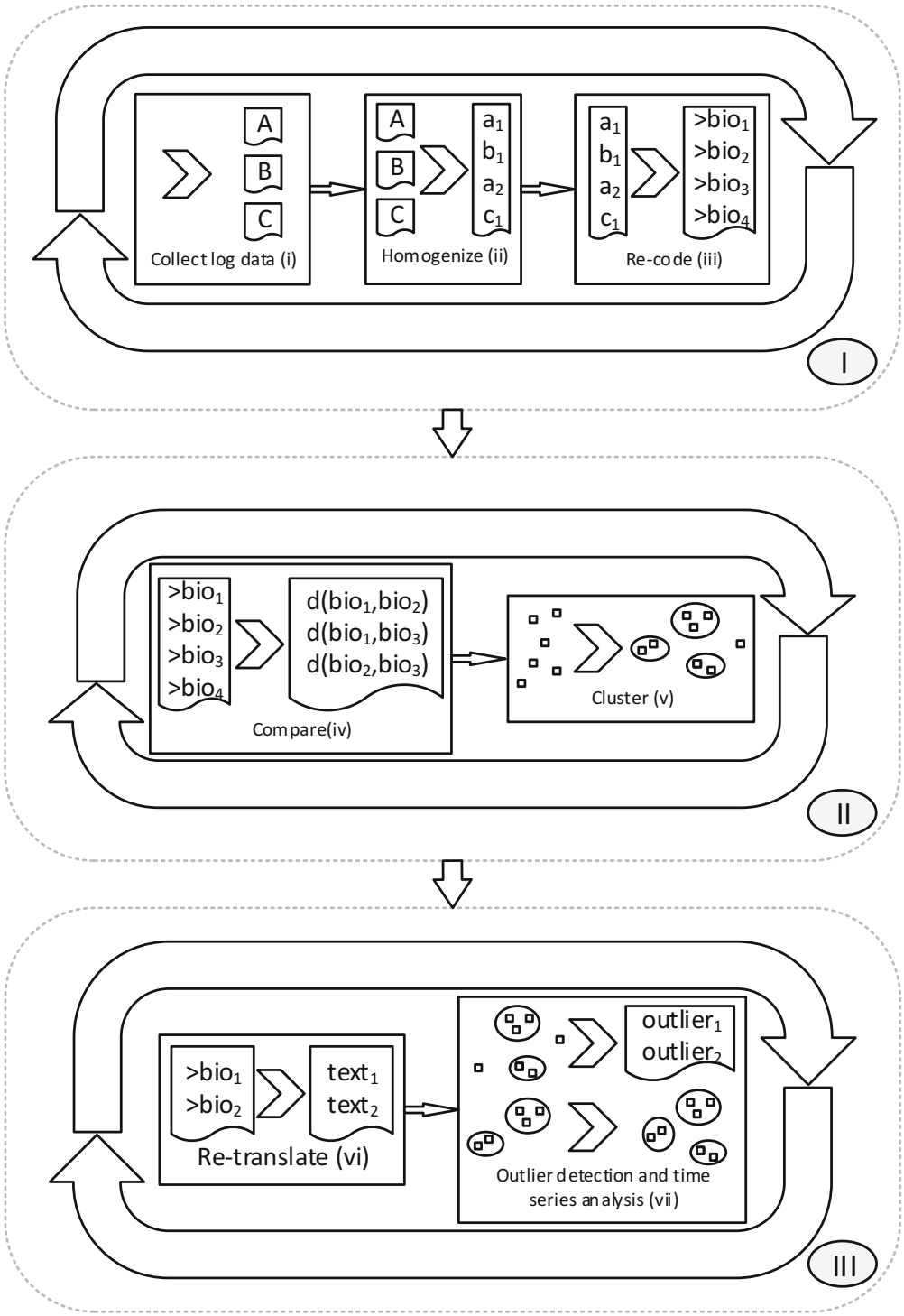


Figure 3.1: Concept to enable the application of high-performance bioinformatics tools in log data analysis [WSFK17, WSFK21].

format is important to order log lines chronologically when combining data from different sources. In step (iii), the homogenized data is re-coded from UTF-8 (256 code units) [Yer03] to the alphabet describing the canonical amino acids (20 code units) [Tay86].

In block II, bioinformatics tools are applied to the re-coded data. During step (iv) the re-coded log lines are compared and a distance between all pairs of lines is calculated. Therefore, a sequence alignment algorithm for amino acid sequences is applied to the data. Based on the calculated distance, the log lines are then clustered in step (v).

Block III implements the security analysis component of the proposed approach. First, in step (vi), a reverse look up function is used to re-translate the log lines from the alphabet describing canonical amino acids to UTF-8 code, which is readable for human users. Finally, in step (vii), an outlier detection and time series analysis are carried out to detect on the one hand rare events and on the other hand changes in the system behavior. Both can be caused by cyber attacks or invaders, as well as mis-configuration and erratic system behavior. The following sections describe the proposed model in details.

3.2 Re-coding log data from UTF-8 to amino acid alphabet

Log data from ICT systems is usually modeled in human-readable textual form. Therefore, to be able to apply tools from the domain of bioinformatics, step (iii), re-coding log data to the alphabet used for representing amino acids and converting it into a suitable format, has to be carried out.

A textual log atom L_{text} is a basic unit of logging information, e.g., one line for line-based logging, or one XML-element, which consists of a series of symbols s – typically letters and numbers (Eq. 3.1). The alphabet used to represent log data consists usually of UTF-8 encoded characters (256 different code units) [Yer03] of 8 bit size. In the following, A_{UTF-8} refers to this alphabet.

$$L_{text} = \langle s_1 s_2 s_3 \dots s_n \rangle \text{ where } s_i \in A_{UTF-8} \tag{3.1}$$

But data represented in this format is unsuitable as input to bioinformatics tools. Those tools require as input biologic sequences encoded with symbols of the alphabet A_{bio} (Eq. 3.2), defined for amino acid or DNA sequences. This alphabet consists of only 20 symbols, which represent the 20 canonical amino acids [Tay86].

$$A_{bio} = \{\text{A,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,V,W,Y}\} \tag{3.2}$$

A re-coding function takes as input a stream encoded as UTF-8 data and transforms it into a representation L_{bio} (Eq. 3.3) that is processable by bioinformatics tools.

$$L_{bio} = \langle s_1 s_2 s_3 \dots s_m \rangle \text{ where } s_j \in A_{bio} \tag{3.3}$$

Number:	0	1	2	3	4	5	6	7	8	9
Symbol:	A	C	D	E	F	G	H	I	K	L
Number:	10	11	12	13	14	15	16	17	18	19
Symbol:	M	N	P	Q	R	S	T	V	W	Y

Table 3.1: Bio alphabet symbol mapping [WSFK17, WSFK21].

In the simplest case, this transformation is a straightforward bijective mapping, where one A_{UTF-8} code unit is represented by two symbols from A_{bio}. However, for data where certain larger blocks appear frequently, those blocks (e.g., server names or IP addresses) could be replaced by a single symbol. This would effectively allow compression of data. Even further information loss could be – depending on the application and use case – acceptable. For instance, frequently appearing symbol blocks could be replaced through applying a more intelligent, but just one way mapping, for example, not a whole IP address, but just the last byte or the address's cross sum could be translated to A_{bio}. Another example are paths (from Web server logs), where each component of a path could be translated through hashing into single symbols of A_{bio}. Furthermore, symbols can be grouped by type, so that, for example, all separators such as '/', ';' or spaces can be replaced by one specific element of A_{bio}.

Even more complex re-coding schemes are possible, e.g., after identifying dynamic and static parts of log lines, more symbols could be spent on the variable parts of log lines (those with higher information entropy) and less symbols (or no symbols at all) on the rather static parts.

The following describes one simple - but effective - method for re-coding log data into A_{bio} in details. In order to re-code L_{text} into L_{bio}, a simple and straightforward solution is to convert each $s_i \in L_{text}$ into two[1] corresponding $s_j \in L_{bio}$, symbol by symbol (without any loss of information). For this purpose, each symbol in L_{text} (i.e., the single characters a log line consists of) is converted to its numerical representation in UTF-8. The result of this operation is L_{utf} (Eq. (3.4), where a_i represents one code unit).

$$L_{utf} = \langle a_1, a_2, a_3 \dots a_n \rangle, \text{ where } a_i \in \{0, \dots, 255\} \tag{3.4}$$

In a second step, each numerical value $a_i \in L_{utf}$ is converted into two symbols of the alphabet A_{bio}. Since the size of this alphabet is 20, a straightforward solution (and in order to use the whole possible input range) is to divide each $a_i \in L_{utf}$ by 20 and additionally keep the remainder of this division. Eventually, both results s_1 (the result of the integer division) and s_2 (the remainder of the division) are mapped via a conversion table (see Tab. 3.1) to A_{bio}. Concatenating all these symbols in a single stream, effectively produces L_{bio} – the input to alignment and clustering tools from the domain of bioinformatics.

[1] Since the size of the alphabet of L_{text} is larger (256 code units) than that of L_{bio} (20 elements), one $s_i \in L_{text}$ has a much higher entropy than one $s_j \in L_{bio}$.

Algorithm 3.1 describes the re-coding process. The symbol $\oplus$ extends the sequence on the left side with the symbol on the right side. The function `utf2num` looks up the decimal symbol number of s_i in a standard UTF-8 table[2] (e.g., the letter 'A' corresponds to number 65). The function `num2bio` looks up the letter representation of the numbers 0 to 19 (according to Tab. 3.1).

Algorithm 3.1: Re-Coding L_{text} to L_{bio} [WSFK16, WSFK17, WSFK21].

$L_{bio} \leftarrow \emptyset$;
$L_{utf} \leftarrow \emptyset$;
for $s_i \in L_{text}$ **do**
  $L_{utf} \leftarrow L_{utf} \oplus \text{utf2num}(s_i)$;
end
for $a_i \in L_{utf}$ **do**
  $s_1 \leftarrow a_i/20$;
  $s_2 \leftarrow a_i\%20$;
  $L_{bio} \leftarrow L_{bio} \oplus \text{num2bio}(s_1) \oplus \text{num2bio}(s_2)$;
end

One option to compress the data needed to represent one log line and reduce the amount of data by 50% is to omit the leading character s_1 (cf. Alg. 3.1), and represent each $s_i \in L_{text}$ by one $s_j \in L_{bio}$ (cf. L_{bio}^{full} in Tab 3.2) instead of two. The leading character s_1 has less entropy compared to the trailing s_2 (cf. Alg. 3.1), because if all 256 units of A_{UTF-8} are occurring in the considered data only the first 12 letters of A_{bio} are used to represent s_1. But, usually less than 100 different units of A_{UTF-8} occur in computer log data. Moreover, these symbols are in general numerically represented in the same region. For example, the syslog standard [Ger09] requires that log data is represented in characters of the ASCII code [Cer69], which is equal to the first 128 symbols of UTF-8. Furthermore, the syslog standard allows only characters with decimal representation in the range from 32 to 126 (95 different characters), i.e., $a_i \in \{32, \ldots, 126\}$. Hence, the possible options for s_1 decrease to at most 5. Thus, only a quarter of all possible options is used to describe s_1. Consequently, s_1 stores much less information than s_2. Furthermore, one s_2 represents at most 5 symbols of A_{UTF-8}. Finally, when omitting s_1 the combination and sequence of $s_j \in A_{bio}$ in L_{bio} raises the entropy of each s_2 obtained from Alg. 3.1. Hence, the length of L_{bio} can effectively be cut to a half by accepting a "small"[3] ambiguity (cf. L_{bio} in Tab 3.2). In the remaining chapter, we always apply this method for re-coding L_{text} into L_{bio} and L_{bio}^{full} refers to the loss-free re-coding.

Finally, to complete step (iii), the data has to be transformed into the correct format. Most bioinformatics tools require data in the FASTA format, which has been introduced by Lipman and Pearson [LP85, PL88]. An example for this format is given in Fig. 3.2. The required header always starts with '$\geq$' and stores the name or information about the

[2] http://www.utf8-chartable.de/unicode-utf8-table.pl?utf8=dec [last accessed 01/21/2021]
[3] depending on the application case

L_{text}:	1	9	2	.	1	6	8	.	...
L_{utf}:	49	57	50	46	49	54	56	46	...
L_{bio}^{full}:	DL	DV	DM	DH	DL	DR	DT	DH	...
L_{bio}:	L	V	M	H	L	R	T	H	...

Table 3.2: Re-coding text from UTF-8 with and without compressing information (adapted from [WSFK17]) [WSFK21].

```
> 0x
LVMHLRTHLVLHPPGPGPNNKIECPIMKLPWKKWMMWKQPEKKKKQPRNLFPI...
> 1x
LVMHLRTHLVLHPPGPGPNNKIECPIMKLPWKKWMMWMQPEKKKKQPRNLFPI...
```

Figure 3.2: Example for two sequences in FASTA [WSFK21].

following sequence. In our case, it simply stores the line number or ID of the re-coded log line in the original log file. Later in step (vi), we use this information to re-translate the results after processing the log data with tools from bioinformatics.

3.3 Comparing and clustering log data applying bioinformatics tools

A promising extension of log analysis is the application of bio-inforamtics tools, such as CD-HIT [LJG02], CLUSTAL [HS88] and UCLUST [Edg10] for clustering log data. These bio-clustering tools can potentially improve the process of classifying normal system behvior and accuracy of anomaly detection, when applied to re-coded log data. The reasons for improvement are manifold. First, many common algorithms applied in IDS require decent knowledge about the syntax and semantics of the input data. However, this is not realistic for log data from different systems using unstandardized configurations. Second, many text mining and clustering algorithms lack the required degree of uncertainty when processing log data. For instance, if two words differ by just one character, they are usually considered as completely different during the clustering process, because for text mining, synonym tables are more appropriate. However, this is not true for log data, where text junks, such as ‘php-admin’ and ‘phpadmin’ should be considered similar, if not almost equal. If only one of the terms should be permitted, a signature-based rule should be applied. Alignment algorithms from the domain of bioinformatics support this by applying a different metric and measure similarity character-based instead of word-based, which eventually improves effectiveness. Third, most text mining algorithms do not handle special characters adequately, as they have different meanings in regular text and log messages. For instance ‘../../etc/passwd’ is hard to process, because ‘.’ and ‘/’ are usually considered as delimiters, which is not applicable to all log data. Additionally, certain log sequences, e.g., paths ‘../../etc/passwd’ and ‘../../../etc/passwd’,

have considerably different meaning, however look similar for text mining algorithms. Through applying deletions and insertions in the bio-representations, these properties are adequately handled by bioinformatics tools and distances are calculated accordingly. In the following, we describe how alignment algorithms from the domain of bioinformatics can be applied for comparing two log lines re-coded into A_{bio} and how bio-clustering tools can be used for grouping log data.

3.3.1 Pairwise log line comparison

Step (iv) applies sequence alignment algorithms to compare two log lines. Most bio-clustering tools base on such squence alignment algorithms. Some examples for such algorithms are the Needleman-Wunsch algorithm [NW70], the Smith-Waterman algorithm [SW+81] and Hirschberg's algorithm [Hir75]. Alignment algorithms use a scoring function d to calculate the distance between two sequences. When comparing two sequences L^A_{bio} and L^B_{bio} element by element, there can occur three possible cases:

1. **mismatch**: symbol s^A_j was replaced by symbol s^B_j,
2. **deletion**: symbol s^A_j was removed in L^B_{bio},
3. **insertion**: symbol s^B_j was inserted in L^B_{bio}.

The alignment between two amino acid sequences is always built under the assumption that L^A_{bio} and L^B_{bio} have common ancestors, i.e., they are homologous. Hence, the algorihms return the alignment which refers to the highest similarity [Mou04]. A score rates how similar two amino acid sequences are. The predefined score for a match is usually constant. In most cases, the score for a mismatch depends on the probability that s^A_j can evolve to s^B_j over time. These probabilities are based on empirical statistics and are represented in a 20×20 lower triangular matrix, which is called scoring matrix. The score for a gap caused by deletions or insertions is also predefined and can depend on the size of the gap, i.e., it is different for opening and extending a gap. The simplest definition for a scoring function d are unit costs. In the following, we apply the simple scoring system from Eq. (3.5), which does not take into account that s^A_j could evolve to s^B_j by a specific probability, because evolutions between amino acids are not applicable to log data.

$$\begin{aligned} d(s^A_j, s^B_j) &= \begin{cases} 1 & \text{if } s^A_j = s^B_j \\ -1, & \text{is } s^A_j \neq s^B_j \end{cases} \\ d(s^A_j, -) &= -1 \quad \text{deletion} \\ d(-, s^B_j) &= -1 \quad \text{insertion} \end{aligned} \tag{3.5}$$

When comparing two amino acid sequences, there are usually various options to build the alignment. In our model, since we consider sequences as homologous, the alignment with the highest score is chosen, because a higher score refers to a higher similarity. For

option	alignment	score
(i)	GAC GC-	$1-1-1=-1$
(ii)	GAC- --GC	$-1-1-1-1-1=-5$
(iii)	GAC G-C	$1-1+1=1$

Table 3.3: Example for different alignment options (adapted from [WSFK16]) [WSFK17, WSFK21].

example, given two amino acid sequences L^A_{bio} = GAC and L^B_{bio} = GC. We assume that L^A_{bio} and L^B_{bio} are homologous. As scoring function we apply d defined in Eq. (3.5). Table 3.3 summarizes the possible alignments. In this case option (iii) would be the optimal alignment, because it yields the highest score.

In the proposed model, the similarity between two amino acid sequences is calculated as the ratio between the number of identical symbols in the alignment and the length of the alignment as shown in Eq. (3.6). Equation (3.6) is a normalized version of the inverted Lvenshtein distance [Lev66], i.e., the sum of identical symbols is calculated instead of the number of changes. In case of the example in Tab. 3.3, the similarity for option (iii) would be approximately $66,66\%$.

$$\text{similarity} = \frac{\text{identicalSymbolsAlign}(L^A_{bio}, L^B_{bio})}{\text{lengthOfAlign}(L^A_{bio}, L^B_{bio})} \tag{3.6}$$

Figure 3.3 shows a full example of the comparison of the two bio-encoded sequences L^A_{bio} and L^B_{bio} from Fig. 3.2, generated with the BLAST tool [AGM+90]. BLAST generates an alignment of L^A_{bio} (c.f. `Query`) and L^B_{bio} (c.f. `Subject`). The resulting alignment is `Algn`, where gaps are inserted so that identical or similar characters are aligned in successive columns. In case there is a bijective mapping back to the original data L_{text}, the original L^A_{text} and L^B_{text} can be depicted aligned using an inverse function as shown in the bottom of Fig. 3.3. Eventually, the differences between the original input lines are marked with either 'X', which means different symbols on the respective positions in L^A_{text} and L^B_{text}, or '-', which means that there is a gap and input stream `Query` could not be aligned to `Subject` for the symbols on this position.

3.3.2 Log line clustering

Step (v), clustering log data builds on the previously defined alignment of two bio-encoded log lines. By re-coding a whole log data set and subsequently pairwise comparison of bio-encoded log lines through sequence alignment as shown in the previous section,

L^A_{text}: 192.168.191.4 - - [30/Sep/2014:00:22:05 +0000] "GET␣/login_page.php␣HTTP/1.1" 200 3307 "-" "Zabbix␣monitoring"
L^B_{text}: 192.168.191.4 - - [30/Sep/2014:00:22:25 +0000] "GET␣/␣HTTP/1.1" 200 5300 "-" "Zabbix␣monitoring"

L^A_{bio}: LVMHLRTHLVLHPPGPGPNNKIECPIMKLPWKKWMMWKQPEKKKKQPRNLFPIKNEGMSPVECHPFPPPFFAILHLRPMKKPNNKSPRGRPRMVWWGAPLNMGTNRGMER

L^B_{bio}: LVMHLRTHLVLHPPGPGPNNKIECPIMKLPWKKWMMWMQPEKKKKQPRNLFPIPPFFAILHLRPMKKPQNKKPRGRPRMVWWGAPLNMGTNRGMER

```
Query: LVMHLRTHLVLHPPGPGPNNKIECPIMKLPWKKWMMWKQPEKKKKQPRNLFPIKNEGMSPVECHPFPPPFFAILHLRPMKKPNNKSPRGRPRMVWWGAPLNMGTNRGMER
Algn:  LVMHLRTHLVLHPPGPGPNNKIECPIMKLPWKKWMMW QPEKKKKQPRNLFPI              PPFFAILHLRPMKKP NK PRGRPRMVWWGAPLNMGTNRGMER
Sbjct: LVMHLRTHLVLHPPGPGPNNKIECPIMKLPWKKWMMWMQPEKKKKQPRNLFPI--------------PPFFAILHLRPMKKPQNKKPRGRPRMVWWGAPLNMGTNRGMER

Query: 192.168.191.4 - - [30/Sep/2014:00:22:05 +0000] "GET␣/login_page.php␣HTTP/1.1" 200 3307 "-" "Zabbix␣monitoring"
Algn:  192.168.191.4 - - [30/Sep/2014:00:22:X5 +0000] "GET␣/␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣HTTP/1.1" 200 X30X "-" "Zabbix␣monitoring"
Sbjct: 192.168.191.4 - - [30/Sep/2014:00:22:25 +0000] "GET␣/--------------␣HTTP/1.1" 200 5300 "-" "Zabbix␣monitoring"

Diff:                                      X              --------------              X  X
```

Figure 3.3: Full example from real data: The first block shows the input in textual form; the second block the bio-encoded sequences; the third block the aligned output in bio-representation; the fourth block the aligned version in text representation and the fifth block outlines the differences ('X' means different symbols in the input streams and '-' means gaps) [WSFK21].

distances can be determined by calculating the similarity of two sequences applying Eq. (3.6). Bio-clustering tools then group the bio-encoded log lines so that the distances between any two cluster members $c_i \in C$ and $c_j \in C$ is lower than the distance to the next cluster center. This analysis can be performed with various existing bio-clustering tools, such as the well established CD-HIT [LJG02]. The clustering algorithm processes sequences incrementally. If sequence does not fit to any existing cluster, a new cluster is generated and the current sequence is the cluster representative. New sequences are compared against the existing clusters' representative sequences. For this purpose, CD-HIT first applies an efficient and fast short word filter similar to the one proposed in [HS98]. If a sequence is considered as similar to the representative sequence of a cluster, the alignment and the exact similarity is calculated. Based on this, the algorithm then decides if the sequence is a assigned to a cluster or generates a new one. Chapter 4 explains the procedure of incremental clustering in more details.

3.3.3 Outlier detection and time series analysis

The following section deals with step (vii) – outlier detection and time series analysis for detecting anomalies [HA04]. These are two powerful methods based on clustering that allow to investigate and interpret a computer network's system behavior. At first sight, these two techniques seem to be significantly different, but considering that time series analysis focus on discovering unexpected changes in a system's behavior, these trends can be seen as outliers as well, even though they do not consist of rare events [SWZ15].

Outlier detection

Outlier detection aims at revealing so called point anomalies [CBK09]. These outliers are clusters with just a few elements and/or large distance to other clusters, which define the normal state of an network environment. In case of log data, outlier clusters include rare or atypically structured events. Those outliers are log lines that require further investigations. Eventually, the previously defined model allows to apply high-performance tools from the domain of bioinformatics on log data to cluster log lines. During the re-translation from A_{bio} to A_{UTF-8} the clusters can be sorted by size to detect clusters of small size, which include outliers. Since it is also possible to generate a representative alignment for every cluster, i.e., to generate a multiple sequence alignment considering all log lines assigned to one cluster, the function defined in Eq. 3.6 can be used to calculate the distance between all obtained clusters. Hence, it is possible to discover the clusters, with the largest distance to the group of clusters describing the typical system behavior of a network environment.

Time series analysis

Having clusters with same relative size to each other over time, i.e., if the size of cluster C_A doubles, also the size of cluster C_B doubles, proves that a stable system behavior has been established. This means that the system behavior is rather static, including also

human users if existing. Major shifts in this situation picture can be the result of new components, configurations or updates on both hardware and software side. But, these changes can also relate to cyber attacks or invaders, which do not generate rare events (outliers), but produce new, previously unknown clusters or trigger changes in the ratio of size between existing clusters. This, for example, can happen when multiple outgoing connections are opened to ex-filtrate data from a database, i.e., stealing proprietary information. In this case, only scanning for rare events (outliers), does not raise an alarm. Hence, also time series analysis is a necessary and suitable tool for detecting anomalous and erratic behavior in a computer network.

The proposed approach has the potential to enable time series analysis of a network's system behavior. Since for every cluster a representative sequence exists (applying CD-HIT, this is the longest sequence of a cluster), which is the one every new sequence is compared with during clustering, it is possible to continuously extend the cluster file with new sequences and monitor the actual situation picture. Another approach would be to generate cluster files of specific time periods (for example, in intervals of a day or a couple of hours) and then compare them and verify, if new clusters occur and if the ratios of sizes between clusters are changing over time. When applying time series analysis, the clusters which account for further investigation are new clusters and clusters of which the size relatively to the other clusters has changed significantly. Section 7.1 explains in details how time series analysis using clustering can be applied to log data.

3.4 Evaluation

The remaining section evaluates the proposed approach for applying bio-informatics tools to cluster computer log data. As mentioned in the beginning of the chapter, the evaluation consists of a proof of concept that motivates further research on the concept of bio-clustering. Chapter 4 provides a detailed evaluation of a more advanced approach that applies incremental clustering and builds on the ideas of the bio-clustering approach.

The application case of the evaluation is outlier detection, which is the primary use of clustering in the area of anomaly detection. Therefore, we investigate the detection capability of the model and assess the runtime performance of the approach. The section structures as follows: First, we describe the set-up of the evaluation environment and the configuration of the different components of the model. Then, we introduce the use case, the evaluation of the detection capability builds on and depict the test data we use for the evaluation. Finally, we discuss the evaluation results.

3.4.1 Evaluation environment and model configuration

As test environment, we used a workstation with an Intel Xeon CPU E5-1620 v2 at 3.70GHz 8 cores and 16 GB memory, running Ubuntu 16.04 LTS operating system.

The implementation of the proof of concept consists of three main parts. First, we use a python script to re-code the log data from UTF-8 code to the alphabet of canonical amino

acids. Therefore, we apply the method described in Alg. 3.1. During the evaluation, we compare two different methods for re-coding log data. Once, we translate the log lines to L_{bio}^{full} (translation without loss of information) and once we compress the data by re-coding to L_{bio} (translation, which compresses the amount of data by just storing the second character with higher entropy, and therefore leads to potential loss of information), as shown in Tab. 3.2.

Next, we apply CD-HIT [LJG02] to cluster the re-coded log data. Since there exist no evolutionary connections between the characters of different log lines, we had to adjust the standard scoring matrix applied by CD-HIT, which uses properties of amino acid sequences. Hence, we modified the downloaded C++ scripts[4] and defined the scoring function as shown in Eq. (3.7). In this notation, gaps represent insertions or deletions.

$$\begin{aligned} d(s_j^A, s_j^B) &= \begin{cases} 6 & \text{if } s_j^A = s_j^B \\ -5, & \text{is } s_j^A \neq s_j^B \end{cases} \\ d_{\text{open gap}} &= -11 \\ d_{\text{extend gap}} &= -1 \end{aligned} \tag{3.7}$$

Furthermore, we configured the algorithm, so that every log line is assigned to the cluster, where the representative log line is the most similar one to the currently processed log line and not to the first cluster it matches to. Moreover, the length of the shorter log line sl must have at least x% length (with $x \in [0, 100]$) of the longer compared log line ll (cf. Eq. (3.8)), where x is the chosen similarity threshold, which specifies how similar two lines have to be to be assigned to the same cluster. Also the length of the calculated alignment must have at least the length of the shorter log line sl (cf. Eq. (3.9)) and at least x% of the longer log line ll (cf. Eq. (3.10)). This ensures a sequence alignment as long as possible.

$$\text{length}(sl) \geq x \cdot \text{length}(ll) \tag{3.8}$$
$$\text{length}(alignment) \geq \text{length}(sl) \tag{3.9}$$
$$\text{length}(alignment) \geq x \cdot \text{length}(ll) \tag{3.10}$$

Finally, we apply a python script to re-translate the amino acid sequences into human-readable UTF-8 coded text data. Therefore, the ID which is assigned to each amino acid sequence during the re-coding process is used to look up the log lines in the original log file, as described in Sec. 3.3.

[4] http://weizhongli-lab.org/cd-hit [last accessed 09/03/2019]

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

Table 3.4: Properties of the semi-synthetic log data used for evaluation of the bio-clustering approach [WSSS16, WSFK21].

3.4.2 Testdata generation

Our test environment consisted of virtual servers running the MANTIS Bug Tracker System version 1.2.1 on top of an Apache Web server, a MySQL database, a firewall and a reverse proxy. The log messages of these systems are aggregated using syslog. For generating the data, we applied a slightly modified version of the approach presented in [SSFF14]. With this method it is possible to generate log files of any size and time period for a given system by simulating user input in virtual machines. In our case, we created four user machines that exhibit a typical behavior on a bug tracker system, for example, logging in and out, submitting and editing bug reports. This allowed us to control the complexity of the scenarios and to inject attacks at known points of time. With this method, realistic conditions can be achieved. The produced log data is representative, because the deployed environment is also used in similar settings by real companies for managing bugs in their software.

For evaluating the proposed approach, we generated 4 different log files. In order to simulate different levels of complexity, we implemented two configurations - configuration I (low complexity: the virtual users only click on the same three pages, in the same order) and configuration II (high complexity, see [SSFF14]), where the users utilize all available features of the MANTIS bug tracker system. We logged the user activity for 10 hours to generate the log files. Table 3.4 shows that the data set length, i.e., number of log lines, is mostly affected by the number of simulated users. In both cases (running one virtual user and running four concurrent virtual users), changing from configuration I to configuration II generated around 15% less log lines. This is the case, because in configuration II there are more options for the virtual users to choose their next actions. Furthermore, there are more available actions that raise a longer waiting time until a virtual user performs his next action.

3.4.3 Detection capability

We evaluated the detection capability of the bio-clustering approach in context of an insider threat scenario. The scenario foresees an insider attacker who is an employee of an organization using the MANTIS bug tracker platform. The employee has valid credentials

```
Jul 16 08:47:32 v3ls1316.d03.arc.local kernel: [757325.314310]
   iptables:ACCEPT-INFO IN=eth0 OUT= MAC=00:50:56:9c:25:67:**:**:**:
   **:**:**:08:00 SRC=***.***.***.*** DST=169.254.0.2 LEN=60 TOS=0x00
    PREC=0x20 TTL=59 ID=36376 DF PROTO=TCP SPT=38947 DPT=80 SEQ
   =901703914 ACK=0 WINDOW=29200 RES=0x00 SYN URGP=0 OPT (020405
   B40402080A1D6066F20000000001030307)
```

Figure 3.4: Log line in which the MAC and IP address are logged during a data base access; the '$\star$' symbols mark the parts of the log line which are modified [WSFK17, WSFK21].

to log into the platform. Usually, he accesses the database through an application hosted on the Web server, but because of a mis-configuration he found out, which port allows direct access to the database. Additionally, he uses a private device to get access to the database to steal data for unauthorized use. In our scenario, the employee wants to access one specific database entry, which he would not be authorized to access, when connecting to the database through the Web server. Therefore, when he connects to the database, a different IP address and a different MAC address, which only occurs once when accessing the database, are logged. Hence, this log event can be detected as outlier. For simulating the scenario, we modified the log lines which are part of one logged data base access from the original log files and added it at a random location. The proposed approach, does not depend on the order of the log lines, because they are sorted by their length, starting with the longest log line, before clustering. The log line, which includes the important information about the MAC address and the IP address is shown in Fig. 3.4. The modified MAC and IP address are chosen randomly.

It also would be possible to detect this kind of attack with a common whitelist approach (i.e., explicitly specifying the known permitted IP and MAC addresses). However, this simple, but catchy scenario allows us to show the sensitivity of our proposed approach and proves its detection capability. Furthermore, the information gathered from this elementary test scenario serves as basis for more complex cases and more complex application possibilities such as time series analysis, as shown in Ch. 4.

To evaluate the detection capability of the bio-clustering approach, we added the modified log lines to all four log files mentioned in Tab. 3.4. In this evaluation, we defined clusters consisting of only one log line as outliers. We calculated two statistics to show the detection capability of the proposed model. First, we calculated the absolute number of false positives FP. We defined every cluster consisting of only a single log line and not including the modified version of the log line shown in Fig. 3.4 as FP. Furthermore, we calculated the false-positive-rate FPR, which we define as the ratio between the number of FP and the log file length (cf. Eq. (3.11)).

$$FPR = \frac{FP}{\text{length(log file)}} \tag{3.11}$$

Conf.	L_{bio}^{full}	L_{bio}
U1C1	0,91	0,88
U1C2	0,91	0,87
U4C1	0,92	0,86
U4C2	0,92	0,88

Table 3.5: Thresholds at which bio clustering detects the outlier [WSFK17, WSFK21].

For the evaluation, we ran the proposed algorithm on the test data varying the similarity threshold, applied for comparing the log lines, between 85% and 99%, raising it by 1% each run. Table 3.5 shows the similarity threshold for both methods (re-coding with and without compression of the data) at which the outlier we searched for was detected first. The outlier was also detected for all higher similarity thresholds. Table 3.6 summarizes some of the results for re-coding log data into L_{bio}^{full}, which is a translation without loss of information. Table 3.7 presents the results for re-coding log data into L_{bio}, which is a translation that compresses the amount of data, but leads to a loss of information (cf. Tab. 3.2).

Table 3.5 indicates the lowest similarity threshold for both re-coding methods and the four test datasets at which the outlier we searched for was detected. Table 3.5 demonstrates that a lower threshold can be chosen to detect the outlier, when re-coding log data to L_{bio}. Furthermore, the lowest threshold at which the outlier is detected, is independent from the number of users and the chosen complexity of the logged network environment. It only depends on the applied re-coding model. Moreover, Tab. 3.5 reveals that re-coding to L_{bio} is more sensitive for detecting outliers, because the outlier is detected at a lower threshold. This can be explained by the fact that when re-coding to L_{bio}^{full} every symbol is re-coded into two symbols of the canonical amino acids alphabet. For the first symbol, only at most 5 out of 20 letters are used (cf. Sec. 3.2). Hence, unique symbols used as leading symbol occur more often, than unique symbols used as trailing symbols (c.f. Sec. 3.2). Therefore, each symbol looses entropy and dissimilarity between to log lines is rated less strictly.

In contrast to the lowest threshold at which the outlier is detected, Tab. 3.6 and Tab. 3.7 show that the number of FP and also the FPR depend on the complexity of the logged network environment. The FP and FPR is higher for the more complex configuration. According to the results, the number of logged users only has a very low impact on the number of FP and the FPR. That was to be expected, because each user can carry out the same actions. Furthermore, the tables show that the number of FP and the FPR for re-coding to L_{bio}^{full} (cf. Tab 3.6) are a bit lower than for re-coding to L_{bio} (cf. 3.7). This can be explained with the fact that the lowest threshold at which the outlier is detected is lower when re-coding to L_{bio}. Again this results from the fact that re-coding to L_{bio} allows a more sensitive outlier detection than re-coding to L_{bio}^{full}. Furthermore in both cases, the FP and FPR start increasing much faster at a specific threshold. Again, because of the higher sensitivity this can be recognized for lower thresholds, when

Threshold	FP_{U1C1}	FPR_{U1C1}	FP_{U1C2}	FPR_{U1C2}	FP_{U4C1}	FPR_{U4C1}	FP_{U4C2}	FPR_{U4C2}
0.86	14	2,89E-05	202	4,89E-04	4	2,12E-06	160	1,00E-04
0.88	17	3,51E-05	286	6,92E-04	6	3,18E-06	378	2,36E-04
0.91	24	4,96E-05	696	1,68E-03	13	6,89E-06	1718	1,07E-03
0.92	27	5,58E-05	758	1,83E-03	16	8,48E-06	2052	1,28E-03
0.95	45	9,29E-05	1659	4,02E-03	40	2,12E-05	4955	3,10E-03
0.96	2223	4,59E-03	5397	1,31E-02	2973	1,57E-03	11978	7,49E-03
0.97	16972	3,50E-02	25763	6,24E-02	59289	3,14E-02	87107	5,44E-02

Table 3.6: FP and FPR results, when re-coding to L_{bio}^{full} (adapted from [WSFK17]) [WSFK21].

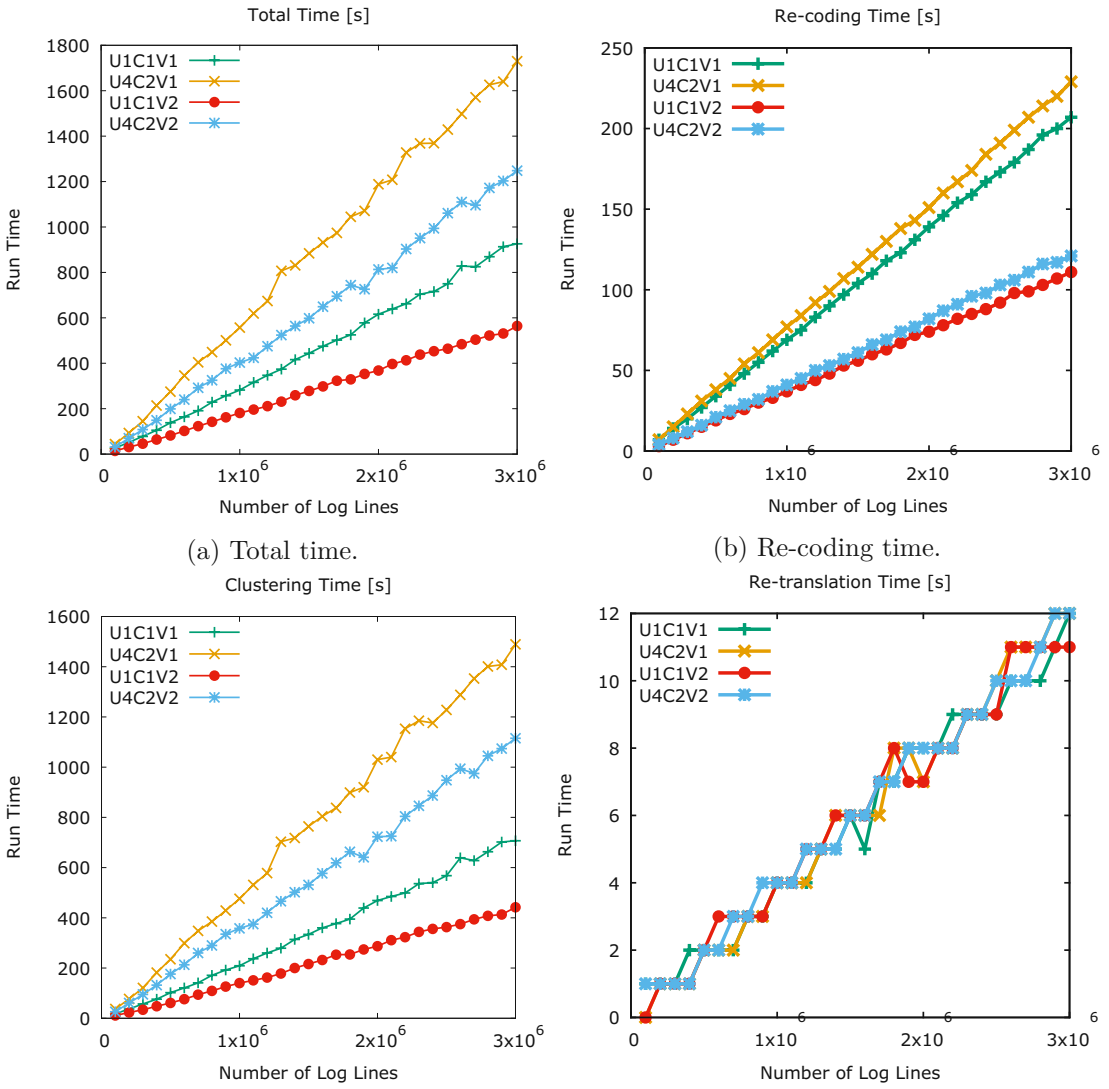
Threshold	FP_{U1C1}	FPR_{U1C1}	FP_{U1C2}	FPR_{U1C2}	FP_{U4C1}	FPR_{U4C1}	FP_{U4C2}	FPR_{U4C2}
0.86	15	3,10E-05	362	8,76E-04	15	7,95E-06	483	3,02E-04
0.87	18	3,72E-05	523	1,27E-03	16	8,48E-06	901	5,63E-04
0.88	22	4,54E-05	701	1,70E-03	19	1,01E-05	1878	1,17E-03
0.90	33	6,81E-05	825	2,00E-03	24	1,27E-05	2119	1,32E-03
0.92	48	9,91E-05	1160	2,81E-03	41	2,17E-05	3363	2,10E-03
0.93	523	1,08E-03	2388	5,78E-03	521	2,76E-04	6030	3,77E-03
0.94	8852	1,83E-02	13964	3,38E-02	21308	1,13E-02	35144	2,20E-02

Table 3.7: FP and FPR results, when re-coding to L_{bio} (adapted from [WSFK17]) [WSFK21].

re-coding to L_{bio} (at 93% similarity) than when re-coding to L_{bio}^{full} (at 96%). Thus, when using a higher similarity threshold, which generates more FP, the outliers could be clustered again, applying a lower threshold, to filter out less interesting outliers. Using realistic similarity thresholds, e.g., the thresholds summarized in Tab. 3.5, the number of FP and the FPR are very low and the detected outliers can be easily investigated manually by a system administrator.

3.4.4 Model scalability

To evaluate the scalability of the bio-clustering approach for detecting outliers, we generated log files of different size, i.e., consisting of different numbers of lines, for the simplest configuration $U1C1$ and the most complex configuration $U4C2$, cf. Tab. 3.4. The length of the log files ranges from 100,000 to 3,000,000 lines. For generating the log files, we applied the approach proposed in [WSSS16]. This algorithm allows to generate highly realistic semi-synthetic log files based on a small piece of real log data. We compared the runtime for re-coding to L_{bio}^{full} (V1) and re-coding to L_{bio} (V2). As similarity threshold, we used the values obtained from the analysis of the lowest value at which the outlier was detected (cf. Tab. 3.5). Besides the total runtime, we calculated the time for re-coding, clustering and re-translating. The plots in Fig. 3.5 demonstrate that the runtime of the bio-clustering approach is increasing linearly with the number of processed log lines. The total runtime is higher for the more complex configuration and also for the translation to L_{bio}^{full}. Depending on the complexity of the data in our test environment, the algorithm is able to process between 1800 and 5000 log lines per second. The re-coding time only depends on the re-coding method and is longer for re-coding to L_{bio}^{full}, since more symbols are generated. The clustering time depends on the complexity of the analyzed system and on the re-coding method, i.e., on the length of the analyzed



(a) Total time.

(b) Re-coding time.

(c) Clustering time.

(d) Re-translation time.

Figure 3.5: Runtime and scalability for the single steps of the proposed bio clustering approach [WSFK21].

sequences. The re-translation time only depends on the length of the log file. The most time consuming part is the clustering.

3.5 Outlook and further development

The proposed bio-clustering approach proves that methods from bioinformatics are suitable to model a computer system's or network's normal behavior. Furthermore, we

demonstrated its anomaly detection capabilities in a proof of concept. The evaluation showed that this novel bio-clustering is capable of fast log line clustering and scaling linearly with increasing number of log lines. Furthermore, it is not necessary to initially specify a number of clusters or delimiters that split the log lines into tokens. Additionally, because of the incremental clustering approach that processes log lines sequentially it is possible to analyze log files of any size.

However, the approach still shows a few weaknesses. First of all, it is not yet applicable for online anomaly detection. The reason for this is, that the applied clustering algorithm sorts the sequences by their length before clustering. Hence, the data must be available before the clustering process starts, why the current implementation of the approach is only appropriate for forensic analysis. Furthermore, the re-coding and re-translation process limits the usability of the bio-clustering approach. When applying a more efficient and sophisticated re-coding function, which is not a bijection, a look-up table is required to re-translate the bio sequences into human-readable format, so that the output can be used by system administrators and security analysts.

Nevertheless, the bio-clustering approach proves the usefulness of the concept behind the bioinformatics clustering tools for log data analysis and anomaly detection. Chapter 4 expands on the bio-clustering concept and introduce an propose an incremental clustering approach that enables efficient and fast log line clustering, similar to the bio-clustering approach, without the steps of re-coding and re-translating log data. Furthermore, the incremental clustering approach enables online anomaly detection by splitting the process into training and detection phase.

CHAPTER 4

Incremental log data clustering for processing large amounts of data online

This chapter builds on the concept of the bio-clustering approach proposed in Ch. 3 that applies bioinformatics tools for clustering log data. Hence, the incremental clustering approach described in the following adopts the idea of clustering log lines incrementally, while it does not require a re-coding of log lines to the alphabet of amino acids. Furthermore, the proposed approach implements online anomaly detection, i.e. log lines are processed at the time they are generated. Online anomaly detection demands high performance (large throughput of log lines per second), and high scalability, so that the approach becomes applicable to large-scale ICT networks. The proposed incremental clustering approach implements semi-supervised self-learning and therefore splits into a training and a detection phase. During the training the algorithm learns a model that characterizes the normal system behavior. After the training, new occurring log lines are compared against this baseline to detect anomalies. Major parts of the remaining chapter have been published in [WSL+17].

We developed a novel clustering approach, because log data exposes two major properties, which make clustering challenging: (i) the amount of log data is rapidly growing – modern ICT networks produce millions of log lines every day – and (ii) log data is rather dynamic – ICT network infrastructures and user behavior change quickly. Hence, clustering approaches that are applied for online anomaly detection have to fulfill some essential requirements: (i) process data timely, i.e. when it is generated, (ii) adopt the cluster map promptly, and (iii) deal with large amounts of data. Nevertheless, existing clustering approaches, that usually process all data at once, suffer from three major drawbacks, which make them unsuitable for online anomaly detection in log data:

(i) *Static cluster maps:* Adapting/updating a cluster map is time consuming and computational expensive. If new data points occur that account for new clusters, the whole cluster map has to be recalculated.

(ii) *Memory expensive:* Distance-based clustering approaches are limited by the available memory, because large distance matrices have to be stored – depending on the applied distance, n^2 or $\frac{n^2}{2}$ elements have to be stored.

(iii) *Computationally expensive:* Log data is stored as text data. Therefore, string metrics are applied to calculate the distance (similarity) between log lines. Their computation is usually expensive and time consuming.

In order to overcome these challenges, we introduce an incremental clustering approach that processes log data sequentially in streams to enable online anomaly detection in ICT networks. We propose a concept that comprises the following novel features:

- The processing time of incremental clustering grows linearly with the rate of input log lines, and there is no re-arrangement of the cluster map required. The distances between log lines do not need to be stored.
- Fast filters reduce the number of distance computations that have to be carried out. A semi-supervised approach based on self-learning reduces the configuration and maintenance effort for a system administrator.
- The modularity of our approach allows the application of different existing metrics to build the cluster map and carry out anomaly detection. We compare the most promising string metrics, against each other and a method that bases on Principal Component Analysis (PCA), adopting a numeric distance metric.
- Our approach enables detection of point anomalies – single anomalous log lines – by outlier detection. Collective anomalies – anomalous number of occurrences of normal log lines that represent a change in the system behavior – are detected through time series analysis.
- We evaluate our approach in a realistic application scenario. We assess the detection capability of our approach and show its scalability.

The remainder of the chapter structures as follows: First, Sec. 4.1 introduces the concept behind the incremental clustering approach. It describes the model for applying string metrics and PCA to carry out online anomaly detection. The section introduces several filters that limit the number of distance computations and lists a number of string metrics that can be used for comparing log lines pair-wise. Furthermore, the section outlines the concept of time series analysis. Afterwards, Sec. 4.2 evaluates the incremental clustering approach and compares the application of the two different models by addressing their advantages and disadvantages. Finally, Sec. 4.3 provides an outlook

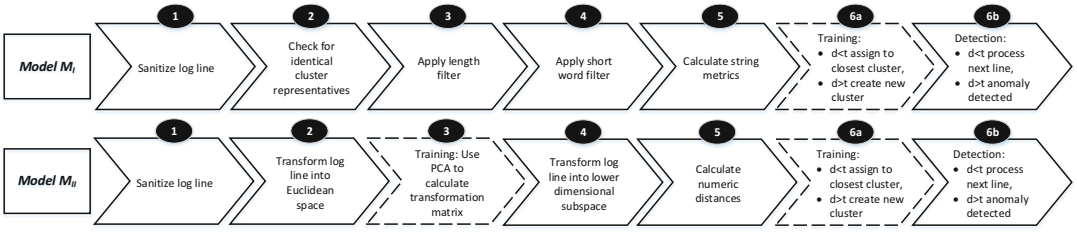


Figure 4.1: Work-flow of model M_I and model M_{II}. Steps marked by dashed line frames are only needed in the training phase [WSL+17].

on further development of the incremental clustering approach and summarizes future work.

4.1 Concept for incremental clustering

The following section describes the proposed concept of incremental clustering for anomaly detection in ICT networks. We define two models to realize this concept. Figure 4.1 visualizes these two models and their differences. Model I (M_I) deals with string metrics that are applied to compare two log lines. Filters reduce the computational complexity and speed up the clustering. Model II (M_{II}) follows an approach based on numerical distances. First, to apply model M_{II}, the textual log data is transformed into the Euclidean space; then, PCA is applied to reduce the amount of insignificant information; finally, the Euclidean distance[1] – a numerical distance metric – between two transformed log lines is calculated to compare them with each other. In both models, the last step decides whether a processed log line is anomalous or not. Both models are described in detail in the following.

The section structures as follows: First, the concept of incremental clustering is explained in detail. Then, model M_I and different string metrics are defined. Afterwards, model M_{II} is described. Finally, the concept for applying time series analysis for anomaly detection using incremental clustering is explained.

4.1.1 Incremental clustering

Incremental clustering focuses on high performance in order to support online clustering of fast growing data, such as log data. A key advantage of incremental clustering is to prevent recalculation of the whole cluster map every time a new data point – log line – occurs. Also, the number of expected clusters does not need to be specified. In opposite to traditional clustering approaches, data is processed in streams and not at once.

[1] $d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$

```
Cluster {size=6, id=355, members=[ClusterMember{lineNumber=16215},
    ClusterMember{lineNumber=17145}, ...]
  Representative: "database-0.v3ls1316.d03.arc.local␣mysql-normal␣
    #011#011#011#011#011␣␣WHERE␣bug_id␣=␣19291"
  - Jul 17 11:23:06 database-0.v3ls1316.d03.arc.local mysql-normal
    #011#011#011#011#011  WHERE bug_id = 19291
  - Jul 17 11:23:28 database-0.v3ls1316.d03.arc.local mysql-normal
    #011#011#011#011#011  WHERE bug_id = 18985
  - Jul 17 11:26:32 database-0.v3ls1316.d03.arc.local mysql-normal
    #011#011#011#011#011  WHERE bug_id=19033
  ...
}
```

Figure 4.2: This is an example of a cluster provided by the incremental clustering approach. Each cluster has a size, which is the number of assigned log lines, an ID and a list of members. Furthermore, each cluster is defined by a representative, which is the log line (without timestamp) that triggered the creation of the cluster. Additionally, the list of log lines assigned to the cluster is provided.

In the proposed incremental clustering approach, each cluster C is defined by a cluster representative c. The cluster representative is the log line l that triggered the creation of the cluster. We define $\mathcal{C}$ as the set of all cluster representatives, i.e., the cluster map. Figure 4.2 provides an example cluster.

In our approach, log data is processed line by line. First, line l is sanitized. This means that, among others, indentations are homogenized, because they are represented differently in different systems. For example, tabs can be represented by different numbers of spaces. Hence, multiple spaces are removed. Furthermore, the creationtime stamp is removed or blacked out during the clustering process, because it is unique for each log line and is not relevant for the clustering.

Next, a set of cluster candidates $\mathcal{C}_l \subseteq \mathcal{C}$ is built. Therefore, the currently processed log line l is compared to all existing cluster representatives $c_i \in \mathcal{C}$. If the distance d between l and c_i is smaller than a predefined threshold t, i.e. $d(l, c_i) \leq t$, c_i is added to $\mathcal{C}_l$.

After $\mathcal{C}_l$ is built, l is added to the closest cluster, i.e. the cluster with the representative $c_i \in \mathcal{C}_l$, that has the smallest distance $d(l, c_i)$. In case multiple clusters share the same distance, l is assigned to the one found first. If $\mathcal{C}_l = \emptyset$ a new cluster is added to $\mathcal{C}$ holding the cluster representative $c = l$.

Given that we apply this concept of incremental clustering to perform semi-supervised anomaly detection, the process comprises a training and a detection phase. During the self-learning training phase the cluster map $\mathcal{C}$ is built as described before. $\mathcal{C}$ describes a baseline of normal system behavior, the so-called ground-truth, against which, log lines are tested for detecting anomalies. During the detection phase each processed log line l, for which $\mathcal{C}_l = \emptyset$ holds, after l was compared to all $c \in \mathcal{C}$, is considered anomalous.

Since the detection process tests log lines against a predefined baseline of normal system behavior, the proposed method represents a white-listing approach.

We assume that the log data processed during the training phase is anomaly free. Hence, the proposed approach can be categorized as semi-supervised. However, this is not realistic: The training phase that runs on real data could already contain anomalies. To reduce the negative effects of possibly anomalous log lines, clusters that only contain a single line or a small number of lines that does not exceed a certain threshold, are considered anomalous after the training phase and are removed from the ground-truth. This reduces the risk that during the training phase malicious behavior is learned as normal. Therefore, the number of false negatives, i.e., not detected anomalies, can be decreased. However, at the same time the number of false positives might increase.

Since the normal system behavior is defined after the training, the maximum number of comparisons during the detection phase is constant.

4.1.2 Description of model M_I

Model M_I (cf. Fig. 4.1) implements the concept of incremental clustering introduced in Sec. 4.1.1 based on string metrics. The currently processed log line l is first sanitized (step 1). Then the set of cluster candidates $\mathcal{C}_l \subseteq \mathcal{C}$ is generated. First, the algorithm checks if the currently processed log line already exists in $\mathcal{C}$ (step 2). If so, the line is assigned to the corresponding cluster. Otherwise, a length filter (step 3) is applied. Clusters C are kept in $\mathcal{C}_l$ only if the length of their cluster representative, $|c|$, lies within a predefined range of $|l|$, for example $\pm 10\%$.

The resulting set of cluster candidates $\mathcal{C}_l$ is then filtered applying a short word filter (step 4) that compares the amount of matching k-mer (substrings of length k) between l and cluster representatives $c_i \in \mathcal{C}_l$ and removes cluster candidates that have less than the required number of matches [GLP11]. This method is often used to cluster biological sequences. Equation (4.1) calculates the number of required matching k-mer M to reach a specific similarity between two lines. L is the length of the shorter line, k the length of the k-mer and p the similarity threshold in percent. Figure 4.3 demonstrates the short word filter.

$$M = L - k + 1 - (1 - p)kL \tag{4.1}$$

For each remaining cluster representative $c_i \in \mathcal{C}_l$, the distance $d(l, c_i)$ is calculated using a string metric (step 5). The following section lists some existing string metrics that are suitable for this task. If for a cluster representative c_i the distance $d(l, c_i)$ exceeds the predefined threshold t, the cluster is removed from $\mathcal{C}_l$. Finally (step 6a/6b), the considered log line l is assigned to the cluster C_i with the smallest distance $d(l, c_i)$. In case that at the end of the process $\mathcal{C}_l = \emptyset$: (i) during the training phase a new cluster is created with representative l (step 6a), (ii) during the detection phase, an alarm is raised since l represents an anomaly (step 6b).

```
pos:    01 02 03 04 05 06 07 08 09 10
lineA:  #  1  2  3     q  u  e  r  y
        |  |  X  |  |  |  |  |  |  |
lineB:  #  1  4  3     q  u  e  r  y
2-mer   1        2  3  4  5  6  7
3-mer            1  2  3  4  5
4-mer            1  2  3  4
5-mer            1  2  3
```

Figure 4.3: Example for the short word filter: Two lines are compared that differ in position 03, which is highlighted with 'X'. The matching k-mer for $k = 1, 2, 3, 4, 5$ are marked with numbers. For example, to reach a similarity of 90% at least 7 2-mer must match (cf. Eq. (4.1)) [WSL+17].

4.1.3 String metrics

In order to compute the distance $d(l_a, l_b)$ or similarity $s(l_a, l_b)$ between two log lines l_a and l_b with their respective lengths $|l_a|$ and $|l_b|$, we apply the metrics defined in the following sections. We compare results regarding detection capability, scalability and computation time later in Sec. 4.2. The normalized distance $\tilde{d}(l_a, l_b)$ lies in the interval $[0, 1]$ and can be expressed through a normalized similarity $\tilde{s}(l_a, l_b)$ by calculating $\tilde{d}(l_a, l_b)$ (see Eq. (4.2)).

$$\tilde{d}(l_a, l_b) = 1 - \tilde{s}(l_a, l_b) \tag{4.2}$$

In the proposed approach, we apply $\tilde{s}(l_a, l_b)$ to calculate the distances $d(l, c_i)$, because the normalized values are more suitable for comparison, which makes it easier to predefine a similarity threshold t.

Levenshtein

Sometimes also referred to as edit-distance, the Levenshtein [Lev66] distance measures the number of edits (insertions, deletions and substitutions) of characters that are required to transform a string a into a string b. In mathematical terms, the distance is defined for $1 \leq i \leq |b|, 1 \leq j \leq |a|$ as the recurrence in Eq. (4.3).

$$A_{0,0} = 0, \quad A_{i,0} = i, \quad A_{0,j} = j$$
$$A_{i,j} = min \begin{cases} A_{i-1,j-1} + 0 & \text{(match)} \\ A_{i-1,j-1} + 1 & \text{(substitution)} \\ A_{i,j-1} + 1 & \text{(insertion)} \\ A_{i-1,j} + 1 & \text{(deletion)} \end{cases} \tag{4.3}$$

The distance between two strings is defined as $d_{LS}(a, b)$ (Eq. (4.4)) and can be normalized through $\tilde{d}_{LS}(a, b)$ (Eq. (4.5)). The complexity of the algorithm is in both time and space $\mathcal{O}(|a||b|)$.

$$d_{LS}(a, b) = A_{|a|,|b|} \tag{4.4}$$

$$\tilde{d}_{LS}(a, b) = \frac{d_{LS}(a, b)}{max(|a|, |b|)} \tag{4.5}$$

Jaro

The Jaro similarity [Jar89] is defined in Eq. (4.6), with m being the number of matching characters between a and b that occur within half the size of $max(|a|, |b|)$ and t being the number of transpositions of characters between a and b.

$$s_{Jaro}(a, b) = \frac{1}{3}\left(\frac{m}{|a|} + \frac{m}{|b|} + \frac{m - t}{2m}\right) \tag{4.6}$$

The computed similarity takes values in the interval $[0, 1]$ and is thus already normalized. There also exists a popular extension by Winkler [Win90] that improves the weight of strings with identical prefixes. The complexity in time and space of this algorithm is $\mathcal{O}(|a| + |b|)$ [Chr06].

Sorensen-Dice

The Sorensen-Dice metric [JMK14] splits a and b into bigrams and computes the ratio between the shared amount of bigrams and their common sum of bigrams. The similarity can be computed as in Eq. (4.7), with k_a and k_b being the number of bigrams that a and b can be decomposed into and k_t the number of bigrams that are identical in a and b.

$$s_{SD}(a, b) = \frac{2k_t}{k_a + k_b} \tag{4.7}$$

There is no need to normalize the result as it always lies in the interval $[0, 1]$. The complexity in time of this method consists of $\mathcal{O}(|a| + |b|)$ for splitting into bigrams and $\mathcal{O}(|a||b|)$ for the intersection of the two resulting bigram sets, which could be reduced to $\mathcal{O}(max(|a|, |b|))$ with a data structure that has $\mathcal{O}(1)$ for delete and contains operations.

Needleman-Wunsch

Similar to the Levenshtein metric, the dynamic programming approach by Needleman and Wunsch [NW70] computes the optimal global alignment of a and b based on the edit

distance, however with the difference that arbitrary penalties for each operation can be specified. The computation can be accomplished using the recurrence in Eq. (4.8), where $1 \leq i \leq |b|, 1 \leq j \leq |a|$.

$$A_{0,0} = 0, \quad A_{i,0} = -i, \quad A_{0,j} = -j$$
$$A_{i,j} = \max \begin{cases} A_{i-1,j-1} + 1 & \text{(match)}, \\ A_{i-1,j-1} + 0 & \text{(mismatch)}, \\ A_{i,j-1} - 1 & \text{(insertion)}, \\ A_{i-1,j} - 1 & \text{(deletion)} \end{cases} \tag{4.8}$$

The Needleman-Wunsch similarity can be found in the bottom-right element of the matrix and is defined as $s_{NW}(a, b)$ (Eq. (4.9)). This similarity measure can result in negative values for strings that are completely different in content and length. Thus in this case the result should manually be set to 0. Further, the similarity can be normalized as $\tilde{s}_{NW}(a, b)$ (Eq. (4.10)). The complexity of the computation is in both time and space $\mathcal{O}(|a||b|)$. This metric is closely related to the Smith-Waterman metric [SW+81] with the difference that this one searches for local alignments instead of global ones.

$$s_{NW}(a, b) = A_{|a|,|b|} \tag{4.9}$$
$$\tilde{s}_{NW}(a, b) = \frac{s_{NW}(a, b)}{max(|a|, |b|)} \tag{4.10}$$

Longest Common Subsequence

The longest common subsequence (LCS) [Hir75] is the longest sequence of characters that is contained in both a and b that however can be interrupted by mismatching characters in a and b. Usually the LCS is retrieved as a string. However, for our purposes only the length of the LCS is relevant. The computation of the length of the LCS is shown in Eq. (4.11), where $1 \leq i \leq |b|, 1 \leq j \leq |a|$.

$$A_{0,0} = 0, \quad A_{i,0} = 0, \quad A_{0,j} = 0$$
$$A_{i,j} = \begin{cases} A_{i-1,j-1} + 1 & \text{if } a_i = b_j \\ \max(A_{i,j-1}, A_{i-1,j}) & \text{else} \end{cases} \tag{4.11}$$

Again the calculated value is located in the bottom-right element of the matrix. A measure for the similarity between a and b is $s_{LCS}(a, b)$ (Eq. (4.12)) and can be normalized through $\tilde{s}_{LCS}$ (Eq. (4.13)). There exists an efficient algorithm introduced by Hirschberg [Hir75] to compute the length of the LCS with a complexity in time of $\mathcal{O}(|a||b|)$ and a complexity in space of $\mathcal{O}(|a| + |b|)$.

$$s_{LCS}(a,b) = A_{|a|,|b|} \tag{4.12}$$

$$\tilde{s}_{LCS} = \frac{s_{LCS}}{min(|a|,|b|)} \tag{4.13}$$

4.1.4 Description of model M_{II}

Model M_{II} (cf. Fig. 4.1) implements the concept of incremental clustering introduced in Sec. 4.1.1 based on numerical distance metrics.

Since log data is stored as text data, it has to be transformed into the Euclidean space (step 2). There are different reasonable methods to achieve this transformation. For the sake of simplicity, we count the occurrence of each character in each log line l and define k as the number of unique characters occurring in the log data, which is equal to the dimension of the considered Euclidean space. As already pointed out in Ch. 3, log files in syslog standard contain at most 95 unique symbols. However, the correct number of unique characters k has to be known in advance and can be derived during the training phase. Furthermore, during the detection phase it must be ensured that unknown characters are left out. Lines where previously unknown characters occur should be considered anomalous and thus raise an alarm.

A common problem that arises when clustering high dimensional data based on distance measures is called the curse of dimensionality [BGRS99]. Increasing the dimension, the difference between the largest and the smallest distance, between points of the considered data, converges towards zero. As a result, output of distance based algorithms becomes unusable.

In order to overcome this problem, we apply principal component analysis (PCA; step 3). A detailed description of this method can be found in [Shl14] and in other related work. PCA allows to reduce the number of dimensions, while as much information as possible is kept. The method uses an orthogonal transformation and projects the sample set from a k-dimensional space into an m-dimensional subspace (with $m \leq k$). The dimension m is equal to the number of considered principal components (PC). The PC are sorted by their variance starting with the largest and thus each added PC contributes less information than the one before. The used transformation is defined in Eq (4.14), where $X \in \mathbb{R}^{n \times k}$ is a data set of n elements with k attributes, $\Gamma \in \mathbb{R}^{k \times m}$ is the matrix that stores the first m eigenvectors of the covariance matrix of X, which is required for the transformation that projects X on the first m PC and $Y \in \mathbb{R}^{n \times m}$ which holds the projection of X onto the basis Γ.

$$Y = X\Gamma \tag{4.14}$$

The transformation matrix Γ is calculated during the training phase and then reused to transform new occurring log lines during the detection phase. Therefore, the detection phase is less computationally expensive than the training phase.

During the training phase n log lines are used to calculate Γ. Determining the best choice for the number of PC m is not trivial and depends on the data as well as the dimension k. Our empirical studies showed that 6 is an appropriate amount of PC for our anomaly detection approach. A number m lower than 4 resulted in a low number of true positives and therefore in a large number of false negatives.

Before a log line l is clustered, it is transformed into a numerical data point $l_x \in \mathbb{R}^k$ in the k-dimensional Euclidean space by counting the number of occurrence of each character (step 2). Then l_x is transformed into $l_y \in \mathbb{R}^m$ (cf. Eq. (4.15)) in the m-dimensional subspace (step 4) by applying Eq. (4.14).

$$l_y = l_x \Gamma \tag{4.15}$$

After the transformation matrix Γ is calculated (step 3), the incremental clustering (step 6a) and the anomaly detection (step 6b) are carried out as described in Sec. 4.1.1. The cluster representatives c used in model M_{II} are defined as the transformation l_y (cf. Eq. (4.15)) of the log line l from which cluster C was obtained. As distance metric (step 5) we use the Euclidean distance d_2 (cf. Eq. (4.16)). Again, to achieve modularity, also other numerical metrics can be used.

$$d_2(a, b) = \sqrt{\sum_{i=1}^{m} (a_i - b_i)^2} \tag{4.16}$$

4.1.5 Time series analysis

This section describes how the incremental clustering approach can be leveraged for time series analysis. The previously presented models yield a specific number of clusters, each with a certain amount of cluster members. Their sum represents the cluster size. The absolute cluster size obtained after the training phase strongly depends on the amount of training data. Assuming that a monitored system's behavior does not significantly change and that the initial data is large enough to be considered as a reasonable sample, it is expectable that increasing the data set by any factor will cause that all cluster sizes grow by the same factor, while the relations between the cluster sizes remain the same.

Outliers, i.e., log lines with large dissimilarity from the rest of the data, are not the only detectable type of anomaly in log data. Another indicator for anomalies are changes in the properties of the clusters, for example, changes in the relative size of the clusters over time. Our approach to detect this kind of anomalies is an extension of the previously described models, and analyzes the relative cluster sizes. First, a time window in which the relative cluster sizes are observed is defined. The relative cluster size is equal to the cluster size divided by the number of log lines processed within the considered time window.

During the training phase, the cluster map $\mathcal{C}$ is built as described in Sec. 4.1.1. After the training phase, the relative cluster sizes are calculated for each cluster. Therefore,

the absolute cluster sizes are divided by the number of log lines processed during the training phase. Afterwards, in the detection phase, every time a log line is assigned to a cluster, a counter for the cluster's size is increased by 1. After the time window is over the relative cluster sizes are calculated and compared with the relative cluster sizes obtained during the training phase. Based on the change of the relative cluster sizes, anomalies are detected. Before the next time window starts, the cluster size counters are reset to 0.

The difference between the relative cluster sizes obtained during the training phase $s_i^{t_0}$, where i is the cluster number, and the relative cluster sizes obtained during the j-th time window $s_i^{t_j}$ can be considered as an indicator for anomalous system behavior. If the value s obtained from Eq. (4.17) is close to 0 for a cluster, it means that the cluster shows normal system behavior, while values greater or lower than 0 indicate anomalous system behavior.

$$s = s_i^{t_0} - s_i^{t_j} \tag{4.17}$$

Due to the fact that a change of the number of lines in one cluster will inevitably influence the number of lines in at least one other cluster, the average difference can be considered to decide if the system behavior, in a specific time window, is anomalous or not (cf. Eq. (4.18), where $n \in \mathbb{N}$ depicts the number of clusters $|\mathcal{C}|$, i.e., size of $\mathcal{C}$).

$$S = \frac{1}{n} \sum_{i=1}^{n} |s_i^{t_0} - s_i^{t_j}| \tag{4.18}$$

To automate the anomaly detection process, a threshold can be defined to raise an alarm when the aforementioned value S exceeds it. The choice of this threshold is not trivial and relies on expert knowledge as the resulting values strongly depend on the structure of the data, the training data size, the time window size and the threshold that was used for clustering.

A weakness of this technique are small clusters and especially outliers, since a change of these has a stronger influence on the relative cluster sizes. To deal with this, we sort the clusters by their size in descending order after the training phase and sum the sizes until we reach 99% of the total amount of lines processed. The remaining log lines are then assigned to a single new cluster. Afterwards, during the detection phase, all lines that cannot be assigned to any of the existing clusters and therefore would be identified as outliers, are added to the cluster representing the remaining 1% of log lines of the training data. The size of this cluster can then be compared in every consecutive time window.

A more elaborated approach for time series analysis is presented in Sec. 7.1.

4.2 Evaluation

This section describes the conducted evaluation of the proposed incremental log line clustering approach[2] for anomaly detection. We assess the detection capability of the models defined in Sec. 4.1 using different parameters, their runtime performance and scalability.

The section structures as follows: First, the evaluation environment and its configuration is specified. Next, the test data used for the evaluation as well as the different attack scenarios are outlined. Finally, the evaluation measures are described and the observed results discussed.

4.2.1 Evaluation environment

The test environment was deployed on a workstation with an Intel Xeon CPU E5-1620 v2 at 3.70GHz 8 cores and 16 GB memory, running Ubuntu 16.04 LTS operating system and Oracle Java 8u102. The workstation runs an Apache Web server hosting the MANTIS Bug Tracker System[3], a MySQL database, a firewall and a reverse proxy. The log messages of these systems are aggregated using syslog.

4.2.2 Testdata

To evaluate the presented approach, we used log data from a real system. We generated the log data in a staged process with designed scenarios. This allowed us to control the actual content and to extract the log lines caused by attacks. To create realistic attacks, we exploited known vulnerabilities of the MANTIS Bug Tracker System, listed in the CVE database[4]. For generating the data, we applied a slightly modified version of the approach presented in [SSFF14]. With this method, it is possible to generate log files of any size/time interval for a given system by running virtual users in virtual machines. In our case, we created four user machines that exhibit a typical behavior on a bug tracker system, for example, logging in and out, submitting and editing bug reports. We generated log data over a time window of six hours. After four hours, one of the users changed his behavior and performed four attacks, each in a 30 minute interval (see Fig. 4.4). We used the first four (anomaly free) hours of the obtained log data as training data, and the remaining two hours to evaluate the detection process.

4.2.3 Attack scenarios

In order to evaluate our anomaly detection approach in a realistic context, we searched for known security vulnerabilities of software used in our test environment. In case of the MANTIS Bug Tracker System version 1.2.18, we found multiple CVE entries that

[2] A prototype implementation of the incremental clustering approach can be found here: https://github.com/ait-aecid/aecid-incremental-clustering [last accessed: 09/10/2020].

[3] https://www.mantisbt.org/ [last accessed: 10/30/2019]

[4] CVE list search: https://cve.mitre.org/index.html [last accessed: 9/25/2019]

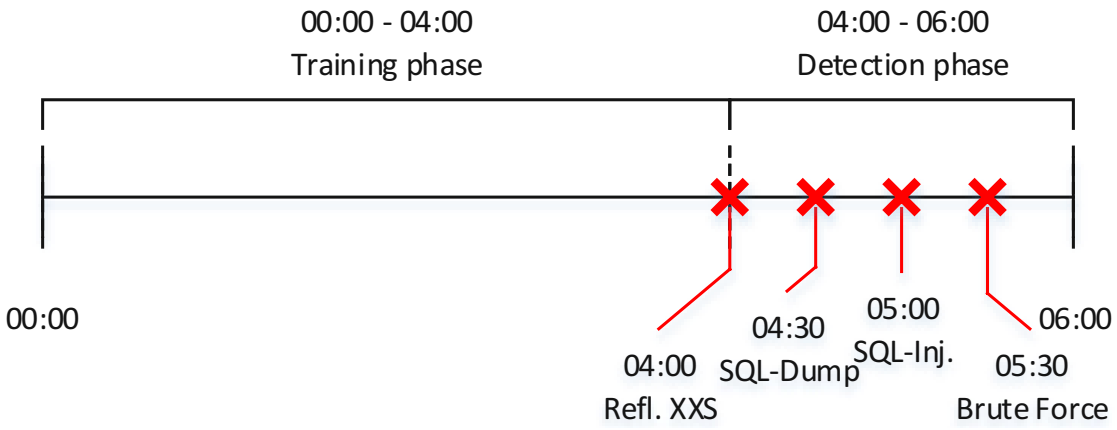


Figure 4.4: Time-line describing the test data generation. The red crosses mark the four attacks [WSL+17].

```
Cookie: MANTIS_MANAGE_USERS_COOKIE=0%3Ausername%20INTO%20OUTFILE
    %20%27/var/www/file. txt%27%20--%20%3A1%3A0
Aug 21 07:50:58 v3ls1316.d03.arc.local mysql-manual-import: WHERE (1
    = 1)
Aug 21 07:50:58 v3ls1316.d03.arc.local mysql-manual-import: ORDER BY
    username INTO OUTFILE '/var/www/file.txt' --  1 LIMIT 0,50
```

Figure 4.5: The figure shows the cookie we used for the SQL-Injection. Furthermore, it presents the log lines generated by the anomalous SQL-Query.

hinted at bugs and exploits of the system. Out of them, we selected two vulnerabilities appropriate for our use cases. We chose one resembling an SQL-Injection (CVE-2014-9573) and another one that resembles a reflective XSS attack (CVE-2016-6837). The SQL-Injection is performed by adding a cookie whose value is not sanitized correctly before appending it to the query. This attack can be seen in the log file as an anomalous SQL-Query and is not trivial to identify. Figure 4.5 provides the used cookie and the log lines generated by the anomalous SQL-Query. In the same time interval also an SSH connection and some crontab statements occured which we also considered anomalous as nothing similar was present in the training data. The reflective XSS attack inserts a rather large script into a URL the user visits. Thus, the corresponding log lines should be easy to detect, because they are significantly longer than normal log lines.

Furthermore, we added a scenario where an insider with direct access to the SQL server executes an SQL-Dump. This generates several hundreds of SQL-Query lines in the log data; some of them looking suspicious, and some of them looking rather normal. Only parts of these will be identified as anomalous.

Moreover, we simulated a brute force attack which consists of repeated attempts to log

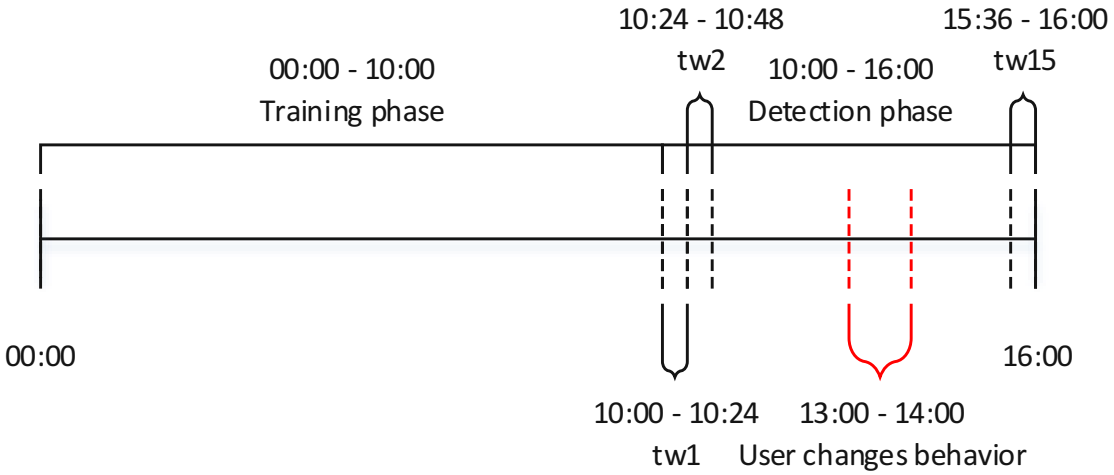


Figure 4.6: Time line describing the test data generation. Black brackets symbolize the time windows and the red bracket the time interval in which one of the users changed his behavior.

into the admin account with random passwords. We expect this to be difficult to reveal as logging in with wrong credentials is part of the normal user behavior that is included in the training phase.

We are aware of the fact that these anomalies could also be discovered by signature-based approaches, but: (i) the attacks and vulnerablities must be known in advance, (ii) the appropriate signatures are required and need to be frequently updated, (iii) they need to be set up correctly, and (iv) no zero-day detection is possible when using signatures. Our incremental clustering approach is more generally applicable, more flexible and not relying on predefined signatures.

Finally, in order to evaluate the detection capabillity of time dependencies (cf. Sec. 4.1.5), i.e. changes in the system behavior over time, by observing relative sizes of clusters for multiple time windows, we created a data set where no specific attack occurred. But the behavior of one user structurally changed after a certain time. We accomplished this by altering the probabilities with which the virtual user uses the features of the MANTIS Bug Tracker System. The change of user behavior is reflected in the log data. In detail, we used log data created by four users that produce log lines for 16 hours. The first 10 hours are used for training and the remaining 6 hours are split into 15 time windows, each lasting for 24 minutes (see Fig. 4.6). The behavior was changed after 13 hours for one hour; due to the produced log lines during the attack, we expect to observe this behavior in the 8th, 9th and 10th time window. A realistic scenario for this event can be that a user, normally using the system, adopts a strange behavior that could indicate anomalous activities, for example, visits of specific pages much more frequently than usual.

4.2.4 Results

The following section summarizes evaluation results. We evaluate the accuracy of the incremental clustering approach for anomaly detection by calculating the F_1-score and computing the ROC curves. Finally, we assess the scalability of the system.

F_1-score

In order to compare the different metrics and their ability to reveal anomalies in aforementioned attack scenarios, we measure the amount of true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN). We compute *Precision* (Eq. (4.19)) and *Recall* (Eq. (4.20)). The so-called F_1-score (Eq. (4.21)) combines these two values and rates the accuracy of the evaluated anomaly detection approach. It takes a high value if both *Precision* and *Recall* are close to 1 and a low value if at least one of them is near 0 [EES10].

$$\text{Precision} = \frac{TP}{TP + FP} \tag{4.19}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{4.20}$$

$$F_1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.21}$$

In Fig. 4.7, the F_1-scores for each metric are plotted against the distance threshold t for each attack scenario. Note, Eq. (4.2) describes the relation between similarity and distance. The threshold t, used for calculating the string metrics, ranges in the unit interval $[0, 1]$. A threshold set to $t = 1$ implies that every new log line generates a new cluster and a threshold set to $t = 0$ indicates that all lines are assigned to one single cluster. Note, when using PCA which applies a numerical distance, the threshold t can be varied in the interval $[0, \infty)$.

Figure 4.7 shows that the results depend on the attack scenario. As Fig. 4.7a demonstrates, the reflective XSS attack is easily detected by all metrics, even with a high distance threshold, i.e. a low similarity threshold. This can be explained by the anomalous length of the log lines caused by the script, which therefore are already sorted out by the line length filter, and the high amount of special characters.

In contrast to that, the lines generated by the SQL-Dump (Fig. 4.7b) require a smaller threshold to be successfully detected. The lines of the SQL-injection (Fig. 4.7c) are also detected appropriately, however a large variability can be observed, with some F_1-scores falling below 0.1 for certain thresholds. An expected effect was that the F_1-score peaks somewhere at a threshold in the range between 0.2 and 0.6 and decreases for low and high thresholds. This can be explained by the fact that a smaller threshold leads to an increase of false positives causing a small value for *Precision*, while a higher threshold

leads to fewer true positives causing a small value for *Recall*. Only in the region where both of these scores reach a high value, the F_1-score maximizes.

Finally, the brute force login attack (Fig. 4.7d) shows how a poor outlier detection looks like as almost all F_1-scores of the string metrics yield zero, meaning that not a single true positive was found. Unlike the string metrics, the Euclidean metric is able to detect this kind of attack quite well. As no administrator logged in during the training phase, the brute force attack with administrator credentials leads to a point in the PCA-transformed space that lies just far enough away from the normal user logins to be identified by the Euclidean distance based anomaly detection. While the string metrics consider these log lines normal as most parts of the malicious line are identical to normal lines except from the username, the number of different characters in the name is weighted higher by the PCA causing the detection of the anomaly. However, it is not a big issue that the string metrics cannot detect a brute force attack, since such anomalous behavior can be detected with the approach presented in Sec. 4.1.5 that applies time series analysis for anomaly detection. Evaluation results presented later in this section prove that.

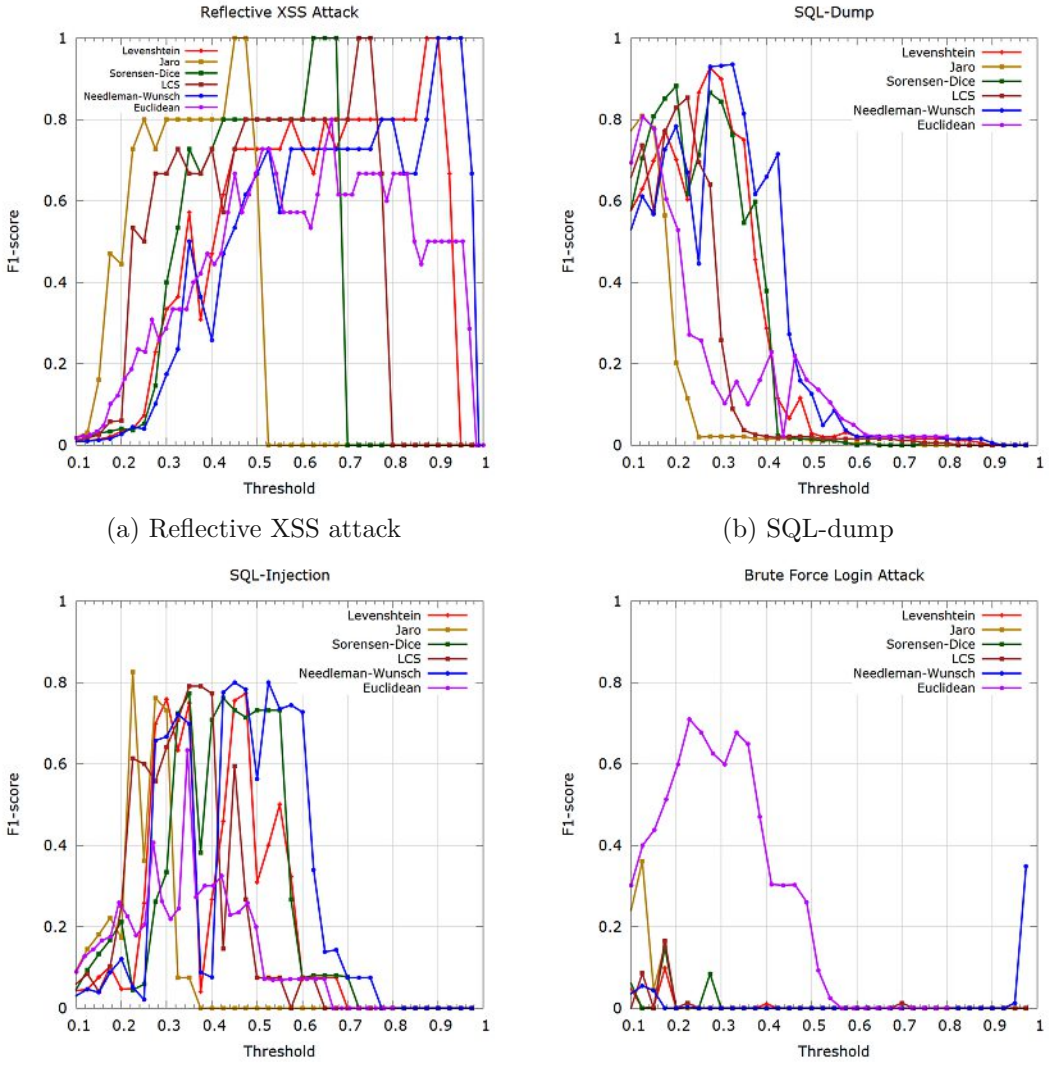
ROC-curve

Another common method for evaluating the performance of a classification system is the Receiver Operating Characteristic (ROC) displayed as a curve of the true positive rate $TPR = \frac{TP}{TP+FN}$ (identical to *Recall*) against the false positive rate $FPR = \frac{FP}{FP+TN}$. Furthermore, the first median is added as a baseline that distinguishes a good anomaly detection from a bad one. The closer the points are to the top left corner of the plot, the more accurate the detection results are. Points under the first median mean that randomly guessing provides a better detection result than the tested algorithm.

The ROC-curves obtained in our experiment are shown in Fig. 4.8. They display the results for each metric and the different attacks. Figures 4.8a, 4.8b and 4.8c show the values for the reflective XSS attack, the SQL-Dump and the SQL-Injection, respectively. All the points are located on the origin or above the first median and most of them are located far in the top-left corner (notice that the scale of the x-axis was set to a value that facilitates the visualization). The results show that the evaluated approach is able to detect the simulated attacks with a high probability. The ROC-curve of the brute force attack displays that the detection of these anomalies with the string metrics has a low accuracy as many of the points are located below the first median. Again, the Euclidean metric proves to be a good choice in this scenario.

Runtime and scalability

The runtime of the anomaly detection is of high importance for most practical applications. We have shown that the chosen threshold has a large influence on the detection performance. Figure 4.9 demonstrates that the threshold cannot be chosen arbitrarily small as the required runtime increases exponentially for decreasing thresholds. This can be easily explained considering that a smaller threshold leads to a higher number of



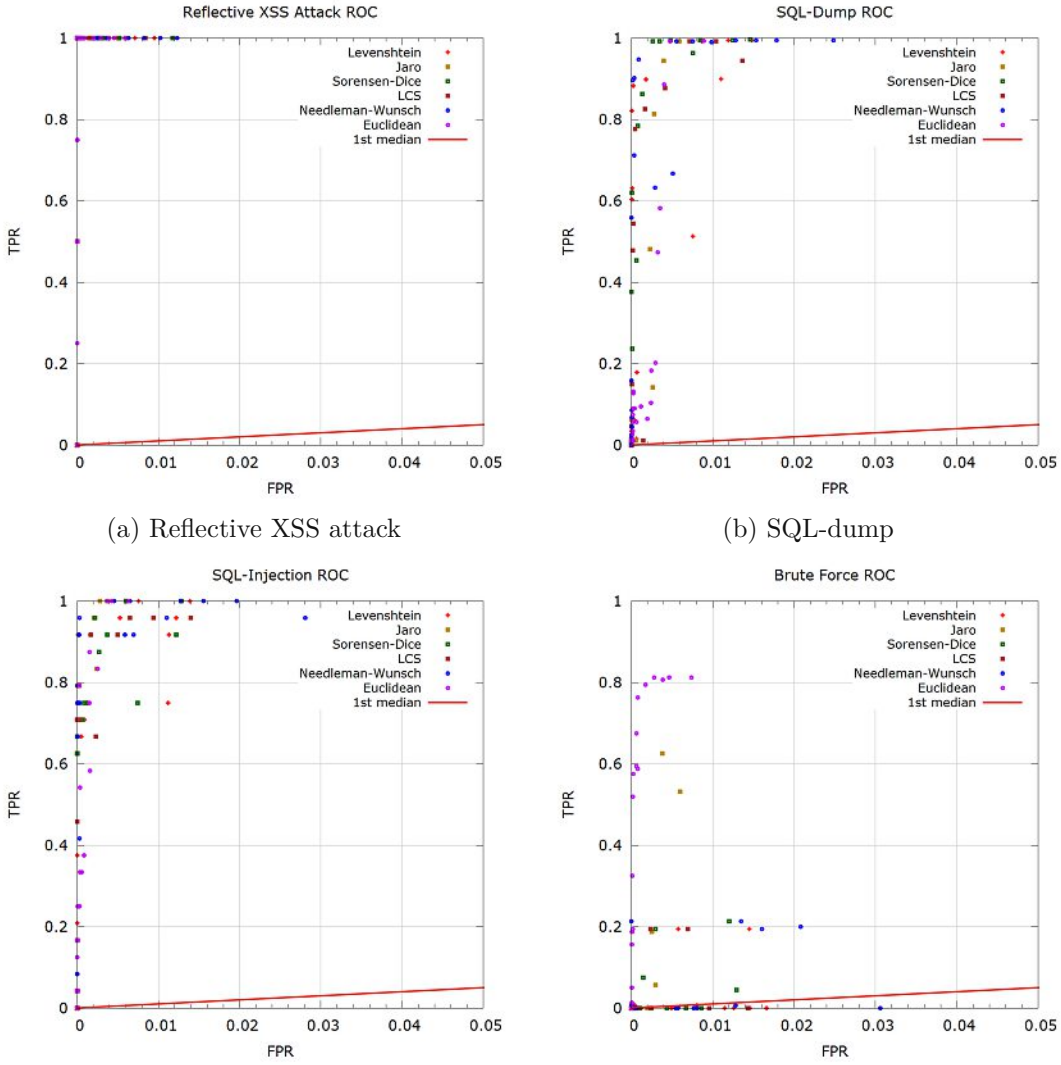
(a) Reflective XSS attack

(b) SQL-dump

(c) SQL-injection

(d) Brute force login attack

Figure 4.7: F_1-score comparison of metrics in different attack scenarios.



(a) Reflective XSS attack

(b) SQL-dump

(c) SQL-injection

(d) Brute force login attack

Figure 4.8: ROC curves of metrics in different attack scenarios (adapted from [WSL+17]).

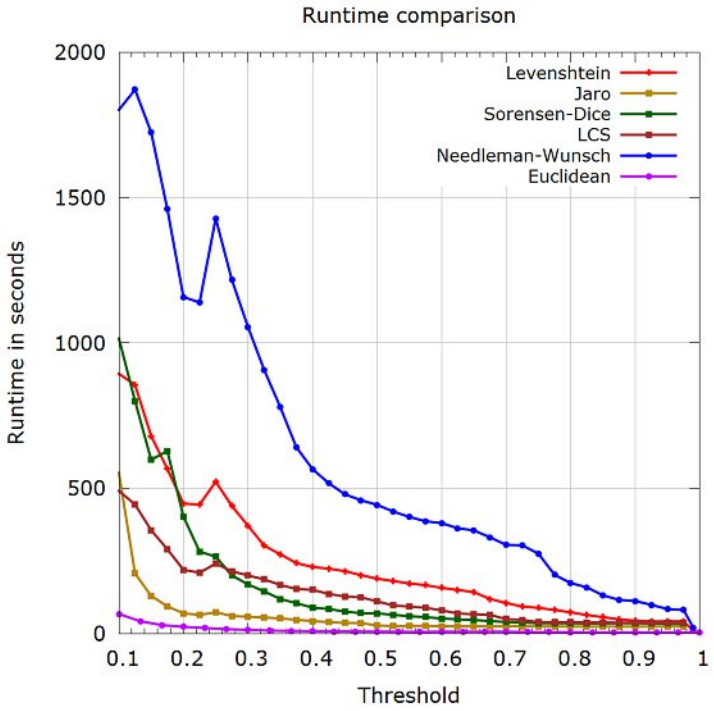


Figure 4.9: Total runtime comparison of metrics.

clusters and thus more computation time is needed to calculate the distances between all cluster candidates. It can also be seen that the string metrics, especially the Needleman-Wunsch metric, require a higher computational effort. In contrast to that, with the Euclidean distance we are able to find the clusters very fast as each log line only requires a single matrix multiplication for the computation of the point in the PCA-transformed space, and the runtime only depends on the amount of clusters that this point needs to be compared with.

We further studied the scalability of our algorithm using different metrics. The results are depicted in Fig. 4.10, where the runtimes of the algorithms applying the metrics for a predefined threshold of 0.8 are plotted against the number of lines that were processed during the testing phase. The plot shows that our algorithm is able to process log lines at constant time rate, enabling online anomaly detection that can be used in practical application scenarios.

Time series analysis

In order to test the anomaly detection model that bases on time series analysis in a realistic way, we designed a scenario where a user suddenly changes his behavior to use the system. He changes his behavior for a specific time window by clicking on Web elements with different probabilities than usual, as previously explained in Sec. 4.2.3.

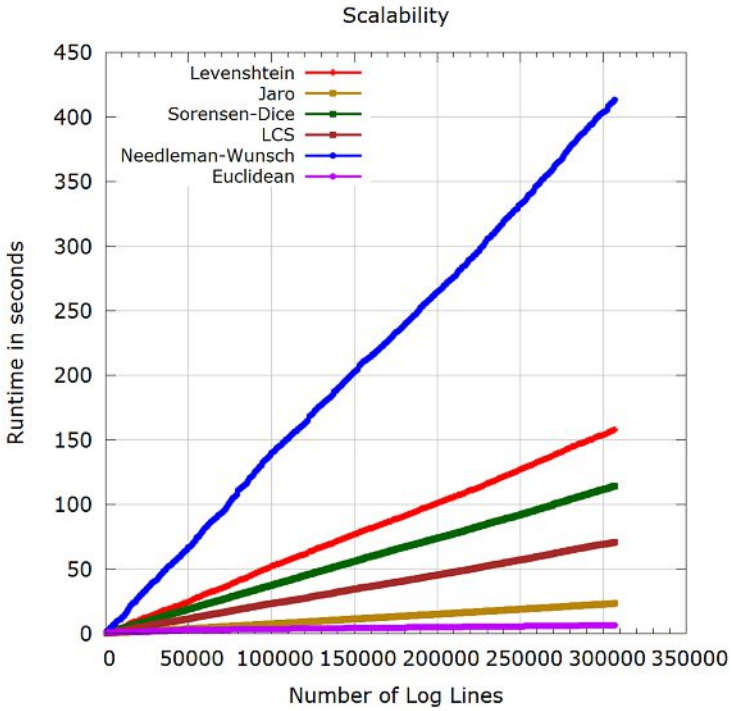


Figure 4.10: Scalability analysis.

For this evaluation, we run the incremental clustering with the Levenshtein distance and a distance threshold $t = 0.2$, which refers to a similarity of 0.8. Figure 4.11 visualizes the average deviation of relative cluster sizes for every time window, which was computed as described in Eq. (4.18). It shows that there is an increase in the cluster size during the 8th, 9th and 10th time window. This are also the time windows we expected to detect the anomalous user behavior. Looking at the deviations during each time window in detail, we exemplary chose 6 clusters to demonstrate the detection of the changing user behavior. The clusters are displayed in Fig. 4.12. The figure shows that some time windows contain anomalies. Spikes into the negative region indicate that the log lines representing the cluster appeared with a lower frequency than in the training phase, while spikes into the positive region indicate that the log lines that represent the cluster appeared with a higher frequency. Finally, Fig. 4.13 shows the difference to the respective average deviations for each cluster and time window.

Our evaluation exemplary proves that the proposed method, i.e. comparing relative cluster sizes of different time windows, allows to detect changes in the system behavior. Figure 4.11 indicates the time windows where a change in the system behavior occurs, which might be caused by malicious activities. To automate the detection process, in the presented scenario a threshold of 0.05% for the average deviation per time window could be set to detect an anomalous system behavior. By coloring the suspicious time windows

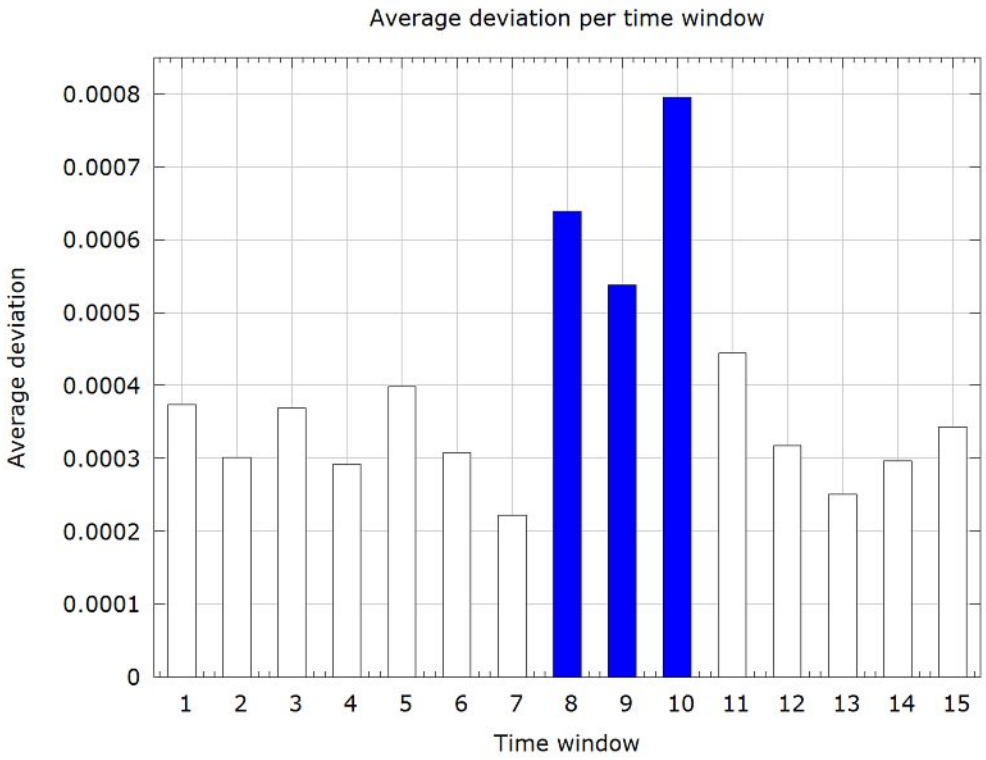


Figure 4.11: Average cluster size deviation per time window.

in Fig. 4.12 and Fig. 4.13, a system administrator can detect the clusters in which the log line frequencies change most. This supports to determine the reason, why the system behavior changed.

Interpretation of the results

Finally, Tab. 4.1 shows the ability of the metrics to detect anomalies in log data in the presented attack scenarios. At least one of the metrics is suited to detect each of the attacks. However, the brute force attack was not intended to be detected easily by our algorithm that only focuses on the similarity of log lines and not on any temporal correlations. Nevertheless, we could show that the Euclidean distance in combination with PCA was able to detect the anomalous log lines. We also implemented an extension of our algorithm that compares the number of elements included in each cluster. After the training phase, it compares the relative sizes of each cluster observed during the training phase with the relative cluster sizes computed within given time windows during the detection phase. This approach allows to detect brute force attacks as the corresponding lines only occur very few times in the training data and more often during the detection phase. This leads to larger relative sizes of related clusters. Furthermore, this method supports the detection of changes in the system behavior over time.

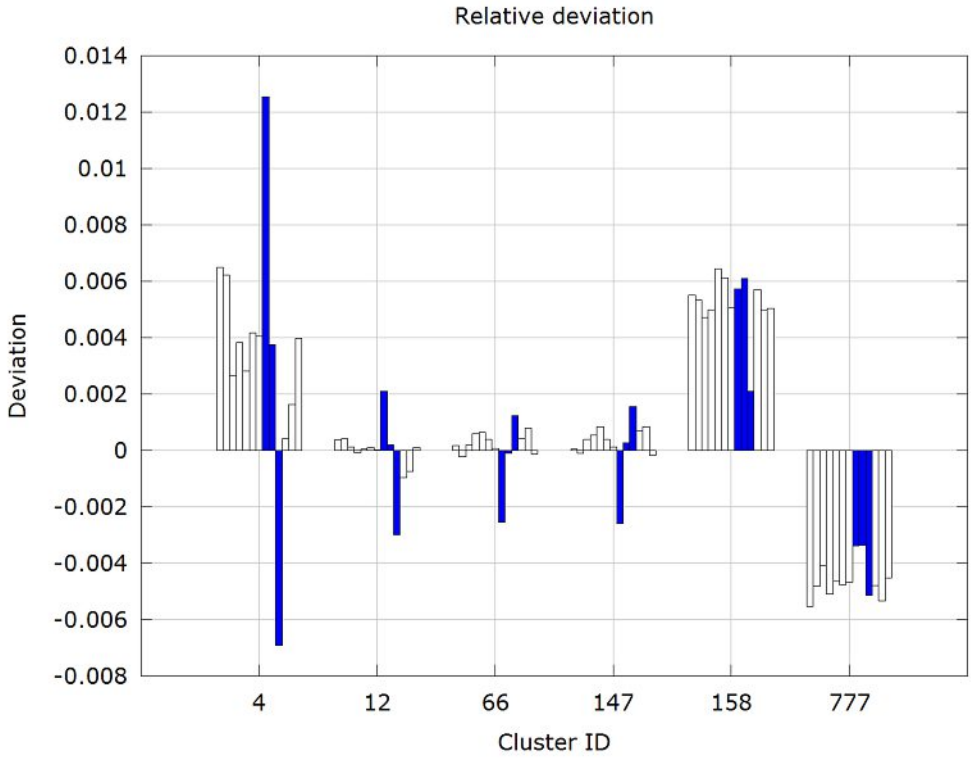


Figure 4.12: Relative cluster size deviation.

Metric	XSS	SQL-Dump	SQL-Inj.	Brute F.
Levenshtein	✓	✓	✓	–
Jaro	✓	∼	∼	–
Sorensen-Dice	✓	✓	✓	–
Needleman-W.	✓	✓	✓	–
LCS	✓	✓	✓	–
Euclidean	∼	∼	∼	✓

Table 4.1: Overview metrics and attacks.

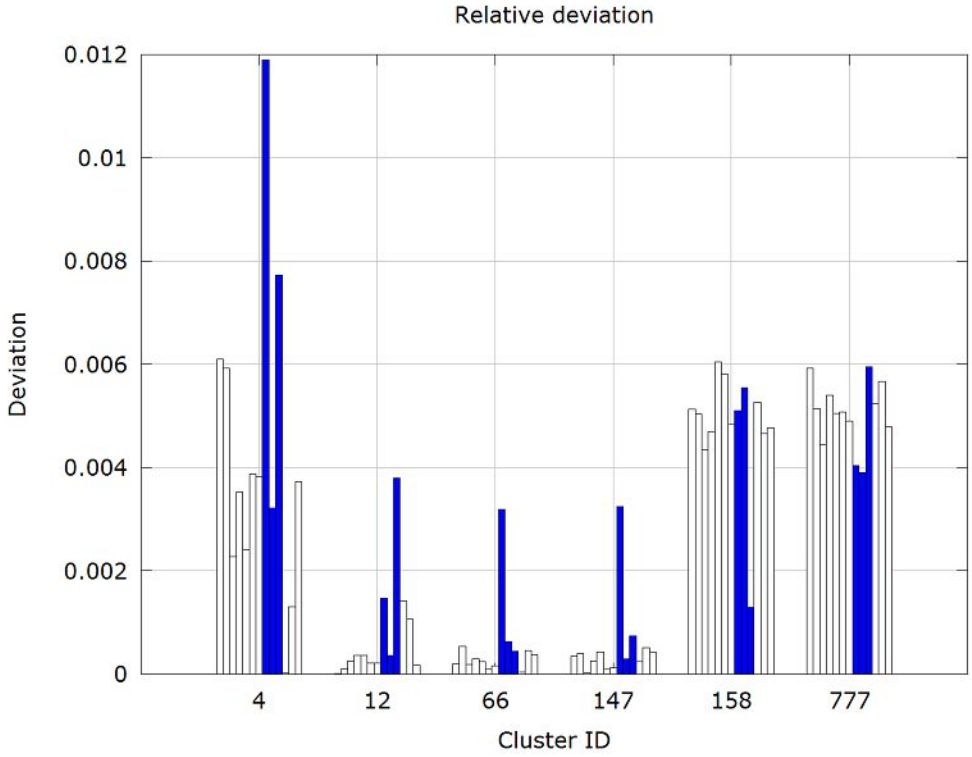


Figure 4.13: Divergence from average cluster size deviation.

4.3 Outlook and further development

The proposed incremental clustering approach for detecting anomalies in log data bases on the bioclustering approach described in Ch. 3. It mitigates the disadvantages of traditional clustering approaches, which lack the ability of processing large amounts of log data in acceptable time, only allow token-based comparison or do not enable online anomaly detection. In order to provide all these features, the incremental clustering approach for log data picks up the idea of incrementally processing data entities and applying filters to reduce the number of distance calculations from bioclustering. However, in opposite to the bioclustering approach introduced in Ch. 3, the incremental clustering does not require a re-coding function and is able to process any textual input and not only biological sequences consisting of canonical amino acids. Hence, the incremental clustering does not demand a re-translation process. Furthermore, it enables online anomaly detection by splitting the process into a training phase, where it learns the normal system behavior and a detection phase, where it monitors log lines and reveals deviations from the normal system behavior. Additionally, besides outlier detection, we provided a proof of concept for the application of incremental clustering for time series analysis. Later, Sec. 7.1 discusses the topic time series analysis as application of incremental clustering for log-based anomaly detection in details.

In Sec. 4.2 we evaluated the detection capabilities of the incremental clustering approach regarding outlier detection and time series analysis in realistic scenarios. However, there are much more opportunities to use the output of clustering in the area of cyber security. First, as shown in Fig. 4.2, each cluster is described by a cluster representative, which corresponds to the log line that initiated the cluster. However, depending on the similarity threshold used for clustering, the log lines within a cluster can show a certain difference to the cluster representative. Therefore, it would be useful to generate a template for each cluster that consists of the static parts of the log lines within the cluster, i.e. of parts that occur in every log line of the cluster in the approximately same location, and replaces variable parts with wildcards. Thus, it is easier to understand the content of the log lines of each cluster and makes analysis and interpretation of the clustering output easier for system administrators and security analysts. Additionally, these templates can be used to generate signatures for signature-based IDS, or could be used as log line parsers to enable further analysis, such as rule-based anomaly detection. Such a rule-based anomaly detection approach is presented in Sec. 7.2. However, currently there exist no efficient template generators that allow to generate character-based templates in acceptable time with a computational complexity lower than $O(n^m)$, where n is the length of the shortest log line and m is the number of lines in a cluster. The reason for this is that there exists no algorithm for calculating multi-line alignments for any type of text. Again, there only exist solutions for biological sequences that build on heuristics that take biological relationships between amino acids into account. Hence, Ch. 5 provides a novel approach that allows to efficiently compute approximations of the optimal character-based templates for pre-clustered log data and reduces the computational complexity to $O(mn^2)$.

CHAPTER 5

Creating character-based templates for log data

The following chapter presents a novel approach for generating character-based templates for pre-clustered log data. Thus, this approach extends the bio-clustering and incremental clustering proposed in Ch. 3 and Ch. 4. Cluster templates provide meaningful descriptions of the content of clusters, support the generation of log parsers that enable further analysis and can be used as signatures in signature-based IDS. Major parts of the remaining chapter have been published in [WHL+20].

Grouping log lines using clustering and classification algorithms is an established method to analyze a computer networks' log data. Clustering is also the basis of further analysis methods, such as outlier detection and time series analysis, which are often applied in cyber security and threat detection. These methods allow to detect suspicious anomalous events and changes in network behavior which are consequence of malicious misuse caused by attackers and malware or erratic behavior initiated by misconfiguration and faulty usage. Once log data are clustered, it is possible to statistically describe these clusters' properties, such as size, or diameter. However, most clustering algorithms provide no or only inaccurate and insufficient information on the content of a log line cluster. Thus, template generators are required that allow to generate meaningful cluster descriptions. Additionally, templates support the process of generating log parsers [HZH+16]. Numerous security applications benefit from templates and template generators, including security information and event management (SIEM) solutions, IDS, parser and signature generators. Furthermore, templates can be applied for log classification in general, for log reduction through filtering, and for event counting.

A template is basically a string that consists of substrings which occur in every log line of a cluster in a similar location. Those substrings are referred to as static parts of the log lines of the cluster. They are separated by wildcards, which represent variable parts

```
Cluster:
database-1.server.d3.local mysql-normal ORDER BY status-system
database-0.server.d4.local mysql-normal GROUP BY status-network
database-1.server.d3.local mysql-normal GROUP BY status-system
database-0.server.d4.local mysql-normal ORDER BY status-network

Template token-based:
[*] mysql-normal [*] BY [*]

Template character-based:
database-[*].server.d[*].local mysql-normal [*]R[*] BY status-[*]t[*]
```

Figure 5.1: Example of templates for a cluster of SQL logs [WHL+20].

of the log lines, such as usernames, IP addresses, and identifiers (ID). Furthermore, a template has to match all log lines of the corresponding cluster.

The unsolved problem of generating a sequence alignment for more than two log lines, i.e., generating a multi-line alignment, is one of the main reasons why currently existing template generators follow token-based approaches and not character-based ones. In this context, tokens are substrings of a string, separated by a predefined delimiter, e.g., space or comma. Token-based template generators first split log lines into tokens. Afterwards, they generate a template, where tokens that represent static parts of the log lines, i.e., occur in all log lines in the same location, remain part of the template, and all other tokens are replaced by wildcards. The biggest advantage of token-based template generators is their high performance with respect to runtime. However, this procedure leads to some significant drawbacks. Token-based template generators prevent that tokens corresponding to substrings with high similarity, which only differ in a few symbols, become part of a template. Thus, they consider words and terms that can be spelled differently, such as `php-admin`, `PHP-Admin` and `phpadmin`, or when SQL queries are used, `username` and `u.username`, as completely different. Furthermore, those approaches require a predefined list of delimiters, which strongly depends on the present log data. Moreover, due to the token-based approach, larger parts of log lines are covered by wildcards, since tokens are considered entirely different, even if they only vary in a single symbol. Additionally, it is often not clear how many tokens a single wildcard represents. Most of the times, a single wildcard replaces a different number of tokens, depending on the log lines that match the template.

In contrast to token-based template generators, character-based approaches do not rely on predefined building blocks in the form of tokens. These approaches recognize static and variable parts of log lines independently from predefined delimiters. Figure 5.1 provides an example for the two different types of templates (assuming spaces as delimiters for the token-based approach) for a certain cluster.

In this chapter, we propose an approach for generating character-based templates to overcome the disadvantages of token-based approaches. The main challenge to achieve

this goal is to calculate a multi-line sequence alignment [Not07], i.e., a sequence alignment for more than two log lines. A sequence alignment arranges two character sequences by aligning their identical or similar parts and recognizing optional and variable characters. There exist many efficient algorithms and string metrics [GF13], such as the Levenshtein distance and the Needleman-Wunsch algorithm, to achieve this for two character sequences. Furthermore, there are algorithms for genetic or biologic sequences to calculate pair-wise and multi-line alignments, which however require knowledge about the evolution of nucleotides and are therefore not suitable for log data [Not07]. Algorithms to align multiple sequences of any characters with no evolutionary context are still missing. The main reason is the difficulty to overcome the high computational complexity of this problem, which is at least $O(n^m)$, where n is the length of the shortest log line and m is the number of lines in a cluster.

Hence, we propose a character-based cluster template generator that incrementally processes the lines of a log line cluster and reduces the computational complexity $O(n^m)$ to $O(mn^2)$. The main contributions are:

(i) Four algorithms to compute multi-line sequence alignments for any strings;

(ii) An incremental approach to efficiently generate character-based templates that provide a more detailed representation than token-based templates;

(iii) A universally applicable template generator for log data independent from delimiters;

(iv) A template generator that overcomes the problem of too generic or over-fitting templates.;

(v) Evaluation of the accuracy of the proposed algorithms, as well as qualitative and quantitative comparison to token-based approaches carried out on real data.

The remainder of the chapter structures as follows: Section 5.1 introduces the concept behind the approach for generating character-based log line templates. Next, Sec. 5.2 proposes three pure character-based approaches for generating character-based templates and one hybrid approach that combines the token-based and character-based approach. Afterwards, Sec. 5.3 evaluates the four proposed algorithms in details and compares them with token-based approaches. Finally, Sec. 5.4 concludes the chapter and provides an outlook on further development.

5.1 Concept for generating character-based templates

In the following, we describe a novel concept that allows to efficiently generate character-based templates for groups of similar log lines, e.g., pre-clustered log lines. The goal of computing a template for a group of log lines is to determine static and variable parts occurring in all of the lines. This allows to recognize shared properties and enables the design of meaningful log line cluster descriptions in form of templates that can be used

for further analysis. Since the aim is to determine common properties, templates are generated for log lines that reach a certain similarity, because otherwise a template would not provide any benefit.

In the remainder, the term template always refers to character-based templates. Furthermore, we define the template of a log line cluster as an ordered list of substrings that occur in the same order in each log line of the cluster. In case of the given example in Fig. 5.1, the template would be [`database-`, `.server.d`, `.local mysql-normal`, `R`, `BY status-`, `t`]. The example shows that for the words `ORDER` and `GROUP` only the character `R` remains part of the template. While there exist several solutions to determine a template for two log lines, it is not trivial to efficiently compute the optimal template for a group of log lines. For two log lines, the template can be generated by simply calculating the pairwise string alignment applying, for example, the Levenshtein (LV) distance or the Needleman-Wunsch algorithm. On the contrary, generating a template for a group of log lines, a so-called multi-line alignment, is complicated. The computational complexity to calculate the optimal template for a group of log lines, applying comparison-based algorithms that omit any heuristics, cannot be lower than $O(n^m)$, where n is the length of the shortest log line within a cluster and m is the number of lines in a cluster. The computational complexity is that high, because each line of a cluster has to be compared with each other line. Due to the large amount of log data, which template generators might have to process, both n and m can be large, which results in a long runtime. On the opposite, for token-based template generators this is not such an issue, because n then refers to the number of tokens within the log lines, which is much smaller than the number of characters. Thus, the goal of the approach we propose is to efficiently compute an approximation of the optimal template for a group of log lines, where each log line of the cluster has to be processed only once.

The approach we propose significantly reduces the computational complexity of computing a character-based log cluster template. Figure 5.2 illustrates the process flow for generating templates for log line clusters. The algorithm processes log lines sequentially and thus follows an incremental approach, which has to handle each line only once. In each step, the algorithm adapts the template. In the following, the term *current template* refers to these temporary templates. Initially, the first line of a cluster defines the current template for the cluster. Next, the algorithm calculates the pairwise alignment between the initial template, i.e., the first line of the cluster, and the second line of the cluster. Afterwards, the algorithm compares the current template with each remaining line in the cluster and adapts the template accordingly. In order to efficiently accomplish this adaptation, we propose four different procedures for this task and compare their advantages and disadvantages. The runtime of these algorithms mainly depends on the applied distance. Our approach uses the LV-distance, because of its relatively low computational complexity of $O(n^2)$, compared to other string metrics that can be applied for calculating pairwise alignments. Hence, it is possible to process a cluster in less than $O(mn^2)$ runtime, where n is the length of the shortest line and m is the number of lines in the cluster. Furthermore, it is possible to modify these algorithms by replacing the

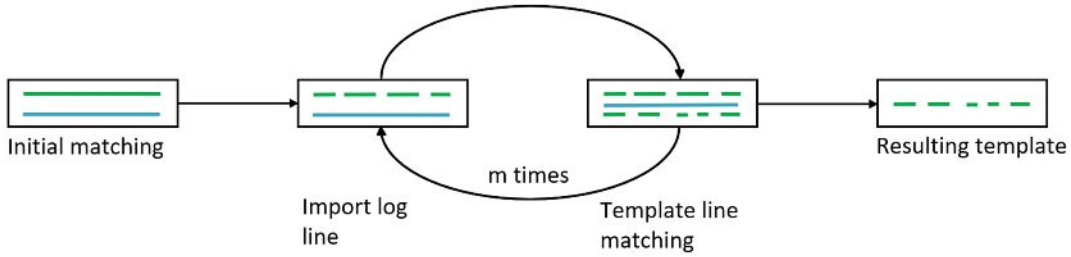


Figure 5.2: Template generation process flow [WHL+20].

LV-distance with any other string metric that allows to calculate an alignment. Since the input data is pre-clustered, the resulting template has a high similarity to the optimal template, as shown in the evaluation presented in Sec. 5.3 by calculating two different metrics that measure the accuracy of the algorithms.

5.2 Cluster template generator algorithms

This section introduces four different algorithms to generate character-based templates for pre-clustered log data. The first two algorithms follow quite different approaches, while the third one combines the advantages of both and simultaneously mitigates their disadvantages. The fourth algorithm combines the token-based and character-based approach. All proposed algorithms build on the calculation of pairwise string alignments, which leverages string metrics. In this chapter, we focus on the Levenshtein-distance (LV-distance). It is possible to replace the LV-distance by any other distance, which determines the shared substrings of two compared strings. We also experimented with the Needleman-Wunsch-distance, but in comparison to the LV-distance the runtime is significantly higher for an output of comparable quality.

The remaining section first describes the initial matching between the initial template, i.e., the first processed log line, which is the one with the earliest timestamp, unless otherwise stated, and the second line of a log cluster, which is the one with the second earliest timestamp. This step is identical for all four algorithms. Afterwards, we define the three purely character-based algorithms *merge*, *length* and *equalmerge*, which enable matching a template with a log line. Thus, they incrementally process all log lines of a log cluster in temporal order to sequentially refine the template, so that the resulting template matches all log lines of the cluster. Finally, we introduce the *token_char* algorithm which combines the token-based and character-based approach to calculate character-based templates.

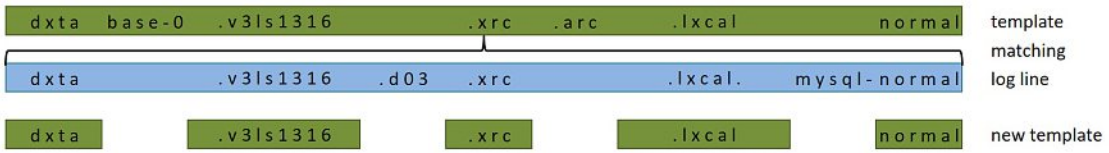


Figure 5.3: Initial matching [WHL+20].

5.2.1 Initial matching

Since a template is defined as a list of substrings that occur in the same order in each log line of a cluster, a string-list characterizes each template. In the following, the term *block* refers to these strings.

Initially, the first template is equivalent to the temporal first line of the cluster. Thus, the string-list consists of a single string which is equal to the first log line of the cluster. Next, the algorithm calculates the LV-distance between the initial template, which is a string, and the second log line of the cluster. The string-list of the template, which is equal to the first line, is now adapted to the substrings shared with the second line according to the LV-distance.

Figure 5.3 illustrates how the first matching of log lines is accomplished. The green blocks represent the template before and after the matching, and the blue block corresponds to the log line which the current template is matched to. Additionally, Alg. 5.1 describes the implementation of the initial matching between two log lines, i.e. strings S_1 and S_2, which is a combination of the calculation of the LV-distance between two strings and a modification of the commonly used backtrace procedure to compute the alignment of two strings based on the resulting scoring matrix of the LV-distance calculation [JM09]. The algorithm described in Alg. 5.1 takes as input the scoring matrix of the LV-distance M and the *path* in M that relates to the optimal alignment. In the algorithm x describes the line index in the scoring matrix M and y the column index. The *path* is represented by the list of directions that have to be taken through the scoring matrix M during the backtrace procedure. Furthermore, the algorithm represents the template T as a list of substrings. In the for loop, the algorithm extends the currently generated substring, which is $\text{LAST}(T)$, with the currently processed character if the direction is *diagonal* and the compared strings have equal characters at the compared position[1]. It ends the substring and appends an empty string to list T, which represents the template, if the direction is *right* or *down*. The latter is only done, if the last element of the list $\text{LAST}(T)$ is not an empty string. In the returned list of substrings T, empty strings represent gaps, which are defined as wildcards for the text between two blocks of a template.

[1]Note, the direction is also diagonal when a character should be replaced.

Algorithm 5.1: STRING_STRING_MATCHING(S_1, S_2) [WHL+20].

```
1  M ← LV_MATRIX(S1, S2);
2  path ← Path in M from M[0][0] to M[LEN(S1)][LEN(S2)];
3  T ← [''];
4  x ← 0;
5  y ← 0;
6  for directions ∈ path do
7      if direction = diagonal then
8          x ← x + 1;
9          y ← y + 1;
10         if S1[x] = S2[y] then
11             LAST(T).APPEND(S1[x]);
12         else
13             T.APPEND('');
14         end
15     else if direction = down then
16         x ← x + 1;
17         if LAST(T)! = '' then
18             T.APPEND('');
19         end
20     else if direction = right then
21         y ← y + 1;
22         if LAST(T)! = '' then
23             T.APPEND('');
24         end
25     end
26 end
27 return T
```

5.2.2 Merge algorithm

The merge algorithm is the most straightforward of the considered algorithms. Figure 5.4 depicts the matching between a template and a log line. First, the algorithm converts the template into a single string by merging the blocks together, i.e., by concatenating the strings in the list into a single string. Then, the LV-distance between this aggregated string and the log line is calculated. Thus, the previous template is adapted, according to the LV-distance, so that it matches also the new log line. Note, it is prohibited that the algorithm deletes already existing gaps in the template, because if this happens the template does not fit previously processed lines anymore. However, gaps are not considered as mandatory, i.e. they do not have to occur in all lines. Algorithm 5.2 describes the linear procedure consisting of three steps: (i) merge the current template T_1 to a single string S_1, (ii) use Alg. 5.1 to compute the alignment T_2 between the merged

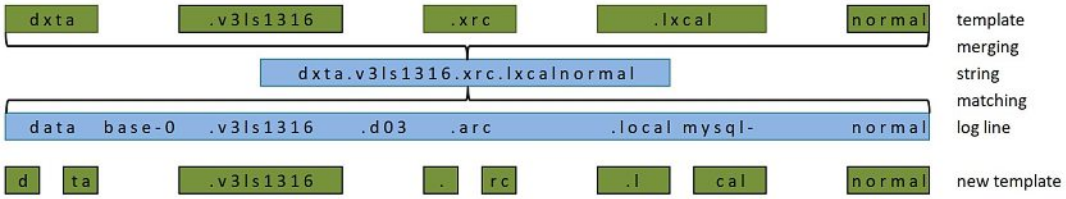


Figure 5.4: Merge algorithm matching: The green blocks represent the template, the upper blue block the merged template and the lower blue block the log line [WHL+20].

template S_1 and the log line S_2, and (iii) ensure that no gaps that existed in the previous template T_1 are missing in the resulting template T.

Algorithm 5.2: Merge(T_1, S_2) [WHL+20].

```
1 S_1 ← Merge_Template _to_String(T_1);
2 T_2 ← String_String_Matching(S_1, S_2);
3 T ← Align_Gaps(T_1, T_2);
4 return T
```

5.2.3 Length algorithm

The merge algorithm always calculates the LV-distance for a log line and the current template, which results in a rather long runtime. Hence, the length algorithm instead only calculates the LV-distance for blocks and corresponding substrings of the log line. This reduces the runtime, because the length of the strings, for which the algorithm calculates the LV-distance, is shorter.

The length algorithm processes the blocks in order of their lengths, beginning with the longest one. Since the algorithm does not process the blocks from left to right and calculates the LV-distance between blocks of the template and corresponding substrings of the log line, it first has to localize which block corresponds to which part of the log line. The localization process is described in more details later in this section. Processing the blocks in order of their length prohibits that smaller blocks prevent larger ones from becoming part of the new template, or to force the algorithm to split them. Therefore, the template tends to include more characters which results in a higher coverage, i.e., on average more characters of the log lines are part of the template of the corresponding cluster. Furthermore, longer blocks are considered more significant for a cluster than shorter ones.

Figure 5.5 supports the description of the length algorithm. The algorithm starts with the localization procedure. For that purpose, it marks all blocks of the template that occur as identical substrings in the log line, starting with the longest one. Figure 5.5 depicts this in the first two lines by connecting block 1 and 3 with equal substrings in the log line. During the marking process, the algorithm does not consider the whole line

for all blocks, but only a valid section to sustain the order of the blocks. For example, the second processed block `.lxcal` in Fig 5.5, can only mark a substring in the section `.d03.arc.local.mysql-normal`, because it has marked blocks to the left and to the right. Empirical studies support to only consider blocks consisting of more than two characters in this phase to avoid that larger blocks are excluded from the resulting template. This leads to templates of higher quality.

Once the algorithm marked all blocks of the template that identically occur as substrings in the log line, it processes the remaining blocks of the template, again in the order of their lengths starting with the longest. Lines three to five in Fig. 5.5 visualize this procedure. Each unmarked block of the template is matched with the corresponding section of the log line. As Fig. 5.5 illustrates, the matched block gets either split or deleted according to the LV-distance. After the matching, the substring that matched the block becomes a marked section and is not further considered in the matching process. For example, the algorithm matches the first processed block `.lxcal` in the lower part of Fig 5.5 with the corresponding substring `.local`. Thus, the algorithm marks this substring, which is illustrated by the dashed rectangle. Therefore, the algorithm matches the third block with a shorter section than the first block.

Note, if at any point during this procedure two blocks have the same size, the algorithm processes the leftmost one first. The fact that similar log lines usually differ more from each other towards the end, supports this decision. As Alg. 5.3 demonstrates, in opposite to the merge algorithm, the input template is modified and returned and not generated from scratch. Therefore, the gap alignment can be omitted. The length algorithm consists of two for loops, one for the marking process and a second one that matches unmarked blocks. Hence, Alg. 5.3 applies Alg. 5.1 to match all blocks (*str* in the Alg. 5.3) from the current template T_1, that have not been marked yet, to corresponding substrings in log line S_2. Once a substring of S_2 has been matched, it is marked so that no other block of T_1 can be matched to it. Algorithm 5.4 describes the function `CORRESPONDING_SUBSTR`. It returns for a block of template $T[j]$ the corresponding substring in log line S. Note, if there is no corresponding substring, the algorithm returns an empty string.

Because of the marking procedure of the length algorithm, the algorithm has to calculate the LV-distance only for the remaining unmarked blocks. Therefore, the runtime of the length algorithm is significantly lower than the runtime of the merge algorithm, which calculates the LV-distance for the whole template and log line. Since the log lines are considered pre-clustered, they have a high similarity, which means that the the marking process significantly reduces the runtime. However, while the marking process reduces the runtime, it might also reduce the quality of the template, because the matching is optimized with respect to sections within the strings and not globally over the whole strings.

Algorithm 5.3: Length(T_1, S_2) [WHL+20].

```
for str ∈ T_1 ordered by length do
    if str ⊆ corresponding_substr(S_2, str) then
        Mark str in T_1 and S_2;
    end
end
for unmarked str ∈ T_1 ordered by length. do
    replace str with String_String_Matching
     (str, corresponding_substr(S_2, str));
    Mark the matched string in S_2;
end
return T_1
```

Algorithm 5.4: corresponding_substring$(S, T[i])$ [WHL+20].

```
if ∃ marked block T[j] in T, with j < i then
    j ← Next smaller index of a marked block in T;
    m ← Index of last marked character of T[j] in S;
else
    m ← 0;
end
if ∃ marked block T[k] in T, with k > i. then
    k ← Next higher index of a marked block in T;
    n ← Index of first marked character of T[k] in S;
else
    n ← len(S);
end
return S[m, n]
```

5.2.4 Equalmerge algorithm

Figure 5.6 depicts the matching between a template and a log line applying the equalmerge algorithm. The following algorithm combines the features of the merge and the length algorithm. Equally to the length algorithm, the equalmerge algorithm first marks the blocks, which occur as substrings in the log line. After the marking, the algorithm merges the blocks remaining between the marked blocks of the template identical to the merge algorithm. The algorithm merges the unmarked blocks according to their corresponding section. Hence, for example, it merges in line three of Fig. 5.6 the remaining unmarked blocks between block 1 and block 2 from line one to a single block. Finally, the newly created blocks are matched with the related sections of the log line. These blocks are split or gaps are included according to the LV-distance. Equally to the merge algorithm, it is prohibited that the algorithm deletes gaps.

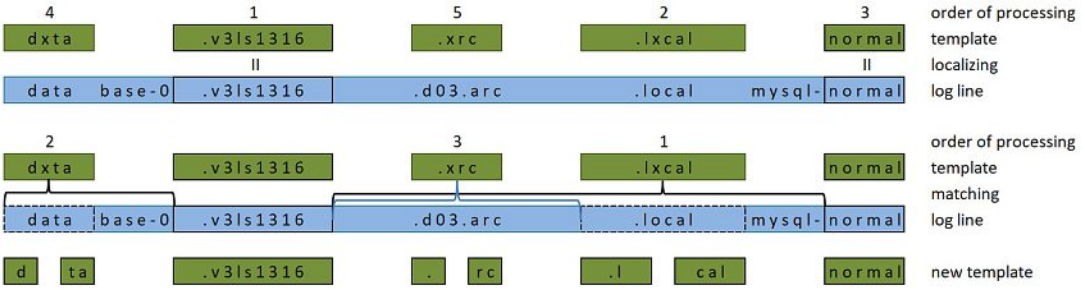


Figure 5.5: Length algorithm marking and matching: The green blocks represent the template and the blue blocks the log line. The upper part illustrates the marking. The lower part visualizes the matching of the remaining blocks. The horizontal brackets highlight the sections of the log line which are matched with the blocks. The dashed rectangle in the lower blue block represents the marked section which originates from the matching with block 1 from above [WHL+20].

Algorithm 5.5 and Alg. 5.3 show that the implementations of the equalmerge and the length algorithm are similar to each other and differ only in the second for loop. In the second for loop of the equalmerge algorithm adjacent unmarked strings, i.e. unmarked strings between marked strings, are aggregated to *adj_strings*. Afterwards, Alg. 5.2 is applied to compute the alignment (T_3) between *adj_strings* and the corresponding substring of the log line S_2. Finally, alignment T_3 replaces the strings in the current template T_1 that have been aggregated to *adj_strings*.

Algorithm 5.5: EQUALMERGE(T_1, S_2) [WHL+20].

```
for str ∈ T1 ordered by length do
    if str ⊆ CORRESPONDING_SUBSTR(S2, string) then
        Mark str in T1 and S2;
    end
end
for unmarked str ∈ T1 do
    adj_strings ← adjacent unmarked strings of str in T1 including str itself;
    T3 ← MERGE(adj_strings, CORRESPONDING_SUBSTR(S2, str));
    Replace adj_strings in T1 with T3;
    Mark T3 and the matched string in S2;
end
return T1
```

The equalmerge algorithm implements a refinement of the length algorithm. Since it calculates the LV-distance between the merged blocks of the template and the corresponding substring of the log line, it has a slightly longer runtime than the length algorithm, but simultaneously the resulting template inherits the higher quality of the merge algorithm.

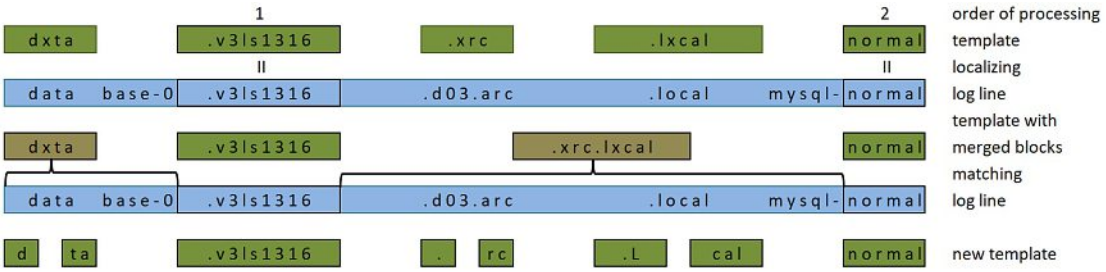


Figure 5.6: Equalmerge algorithm matching [WHL+20].

At the same time, the runtime of the equalmerge algorithm is shorter than the one of the merge algorithm, while the decrease of the quality of the template is smaller than the one of the length algorithm.

5.2.5 Token_char algorithm

Since most template generators operate token-based, we developed a hybrid approach, which should combine the advantages of both token-based and character-based approaches. While, for example, token-based templates are easier to convert into parser models, character-based templates provide a more detailed description of log line clusters and provide more accurate signatures. Thus, to accomplish a hybrid template, we separate the template into two layers. The first layer comprises the token-structure, which contains the token-list that stores the tokens. The second layer composes the character-structure. Therefore, a character-structure is assigned to each gap, which contains a character-based template for the tokens that are replaced by the gap. In the end, the token and the character structure are merged to a character-based template.

Figure 5.7 depicts the procedure of the matching performed by the token_char algorithm and supports the algorithm's description. The initial step of the token_char algorithm differs from the previous algorithms. First, the algorithm converts all log lines of a cluster into token-structures, i.e., lists of tokens. Therefore, the algorithm splits the log lines into substrings at predefined delimiters. Hence, this algorithm inherits the disadvantage of token-based template generators, which have to split all log lines at the same delimiters, whether it is useful or not. Next, between each token, a character-structure is established which initially contains the corresponding delimiter. Finally, the token-char-structure of the temporal first log line represents the initial template.

The following describes the matching procedure between a token-char-template and the token-char-structure of a log line. In the first step, the algorithm matches the two token-structures. Therefore, the algorithm searches for tokens in the log line's token-structure that correspond to the tokens in the token-structure of the template. The distance metric the algorithm uses is a modification of the LV-distance, which treats tokens like characters and weights their value for the accuracy of the template by their length. This is necessary, because the normal LV-distance applied to token-structures would provide

the template with the most tokens, without taking into account that a token consisting of a larger number of characters supports a template with higher coverage. Otherwise, a template with low coverage would be accepted as long as it consists of a large number of tokens. Thus, our algorithm matches the tokens according to the LV-distance with the difference, if two tokens of the template match the same corresponding token of the log line's token-structure, the score assigned by the algorithm for computing the LV-distance is decreased by the length of the token. This is reasonable, because when calculating the LV-distance, positive scores represent penalties, i.e., positive values correspond to required modification operations when transforming one string into another. Note, the result is not a distance, but a sufficient score for this algorithm. The first two lines of Fig. 5.7 depict the matching of the token-structures.

Next, the algorithm converts the tokens of the token-structure of the template which do not match any of the log line's into character-structures and merges all adjacent character-structures. Hence, there exists exactly one character-structure between two tokens as line 3 of Fig. 5.7 shows. Finally, the char-structures of the current template are matched with the corresponding, so far unmatched, parts of the log line. For this purpose, any of the previously described algorithms for generating character-based templates can be used. Lines 3 and 4 in Fig. 5.7 visualize the final step and line 5 shows the resulting template.

For the evaluation of the algorithm, we chose the merge algorithm, because it provides the most accurate templates among the algorithms, as the evaluation in Sec. 5.3 shows. The disadvantage of the longer runtime is mitigated, because of the shorter length of the compared strings.

Algorithm 5.6 describes the implementation of the token_char algorithm. First, the algorithm splits log line S_2 into tokens and transforms it into a token structure T_2. Then it performs the token matching between the current template T_1 and the token-structure of log line T_2. In this step, the algorithm also generates the character structure of the log line. The algorithm compares the character structures $string\_template_1$ of the current template and their corresponding character-structures $string\_template_2$ of the log line in a for loop. For that purpose, the algorithm iterates over the gaps of the token-structures T_1 and T_2, which as mentioned refer to the character-structures. For matching the character-structures, the algorithm applies Alg. 5.2. Finally, the resulting alignment of the character-structures replaces the corresponding character-structure $string\_template_1$ in the current template T_1.

5.3 Evaluation

The following section presents the evaluation of our approach for generating character-based cluster templates[2]. First, we describe the data used for the evaluation. Next, we

[2]A prototype implementation of the cluster template generator approach can be found here: https://github.com/ait-aecid/aecid-template-generator [last accessed: 09/10/2020].

Algorithm 5.6: Token_char(T_1, S_2) [WHL+20].

```
T_2 ← Split_into_tokens(S_2);
Token_Matching(T_1, T_2);
for (string_template_1, string_template_2) ∈ Gaps(T_1, T_2) do
    Replace string_template_1 in T_1 with Merge(string_template_1,
      string_template_2);
end
return T_1
```

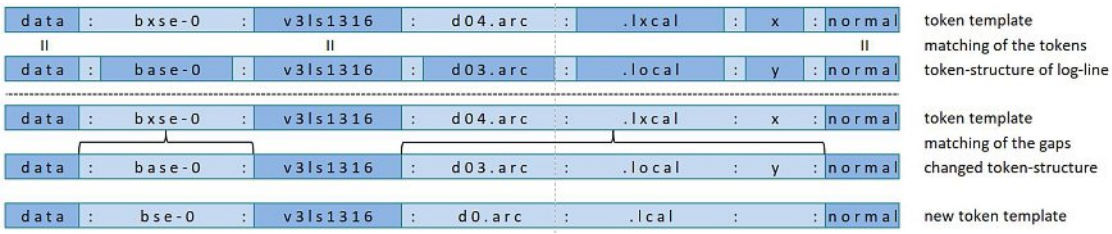


Figure 5.7: Token_char algorithm matching. Dark blue parts represent token-structures and light blue parts character-structures. Colons represent any fixed set of predefined delimiters [WHL+20].

define a similarity score that we calculate alongside the F-score to asses accuracy and quality of the algorithms introduced in Sec. 5.2. Finally, we interpret and discuss the evaluation results. All evaluations haven been carried out on a Notebook with an Intel Core i7-5600U CPU 2.60 GHz and 16 GB RAM running Windows 7 64-Bit. The assessed algorithms have been implemented in Python 3.7.

5.3.1 Test data

For the evaluation of our approach, we use three different data sets. This demonstrates the broad applicability of the approach for various log data types. The first data set, we refer to as *DS-A*, originates from a network that runs a MANTIS Bug Tracker System[3]. Therefore, the data set contains logs from an Apache Web server hosting the MANTIS platform, a MySQL database, a reverse proxy and a firewall, as well as a mail server. The log messages of these systems are aggregated using syslog. The data set consists of 1.6 million log lines that reflect 10 hours of system usage. The second data set, we refer to as *DS-B*, derives from the same system. DS-B includes the syscalls of the system, which have been collected using the auditd service. The third data set, we refer to as *DS-C*, includes logs from a Hadoop File System running on a 203-node cluster on Amazon's EC2 platform [XHF+09]. DS-C consists of 11 million lines that reflect almost 3 days of system behavior. Since the evaluated algorithms require pre-clustered data, we clustered the data applying the incremental clustering approach from Ch. 4, using

[3] https://www.mantisbt.org/

	DS-A	DS-B	DS-C
data set size	10.000	133.000	200.000
line length	60 / 135.94 / 1959	79 / 211.10 / 328	92 / 139.03 / 311
word #	3 / 12.60 / 133	8 / 32.67 / 58	9 / 13.72 / 31
cluster #	352	180	21
cluster size	1 / 28.41 / 605	1 / 741.47 / 13857	1 / 9523.81 / 46585

Table 5.1: Properties of the subsets of the described data used for evaluation. For the line length, the number of words (space separated substrings) and the cluster size, the table provides values for minimum, mean and maximum [WHL+20].

a similarity threshold of 0.9 for DS-A and DS-B, and 0.6 for DS-C. We selected the similarity threshold with respect to the structure and complexity of the data. We chose a lower similarity for DS-C, because the data set includes larger variable parts and a higher similarity threshold would lead to a large number of small clusters that would represent an inappropriate cluster arrangement that includes many similar clusters.

5.3.2 Evaluation metrics

We used two different evaluation metrics to assess and compare the different algorithms. The first one is a score for similarity, which is defined in the following, and the second one is the F-Score.

The Sim-Score measures the similarity between the log lines of a cluster and its corresponding template. The algorithms for generating character-based templates provide templates that match all log lines of a cluster. Therefore, the ratio between the number of characters the template consists of and the average log line length is a measure for similarity between a template and the log lines of a cluster. In the Sim-Score, the average log line length corresponds to the mean of the number of characters the log lines of a cluster consist of. Consequently, the resulting Sim-Score for each algorithm is the mean of these similarities. The Sim-Score is calculated as shown in Eq. (5.1), where n is the number of clusters, m_i the number of log lines in the i-th cluster, T_i the template of the i-th cluster, $L_{i,j}$ is the j-th log line of the i-th cluster and $|\cdot|$ denotes the number of characters of a template or a log line.

$$\text{Sim-Score} = \frac{1}{n}\sum_{i=1}^{n}\frac{|T_i|}{\frac{1}{m_i}\sum_{j=1}^{m_i}|L_{i,j}|} \tag{5.1}$$

The Sim-Score is an evaluation metric that indicates how accurate the templates are. One advantage of the Sim-Score is that it does not rely on any additional information about the clusters, such as a ground truth, which defines the optimal template. Thus, it can be calculated directly after generating templates, for any data set. Table 5.1 presents properties of the data we used for evaluating the Sim-Score.

The second metric we used to evaluate the proposed algorithms for generating character-based templates is the F-score (see Eq. (5.2)). The F-Score allows an assessment of the accuracy of the generated templates. However, in opposite to the Sim-Score, the calculation of the F-Score requires a ground-truth to identify true positives (TP), false postives (FP) and false negatives (FN), as Eq. (5.2) indicates. Therefore, we first had to create a character-based ground truth for all data sets.

$$F\text{-score} = \frac{2TP}{2TP + FN + FP} \tag{5.2}$$

Furthermore, we defined the terms TP, FP and FN as follows[4]:

- TP are defined as the characters which appeared in the same order in both the ground truth and the created templates.
- FP are characters, which occur in the template but not in the ground truth.
- FN are characters, which occur in the ground truth but not in the template.

FP are an issue that cannot simply be ignored. The major reason for FP are over-fitting templates. The algorithms tend to create overly accurate templates, because they only generate them from the perspective of the log lines that are associated with a cluster and not taking other knowledge into account as humans would do. Reasons for this are characters that actually represent variable parts of a log line, but occur in each log line of a cluster. However, these characters are not part of the ground truth, because they, for example, refer to an IP address or a part of a timestamp, which might only be static for the training data and thus are not considered static in the ground truth. An example is a variable within the same cluster that takes the values `192.67.200.155` and `192.67.200.12`. In this case, `192.67.200.1` becomes part of the template, although the last character `1` belongs to a variable part of a log line. Hence, the resulting template would not match the IP address `192.67.200.2`, which might be also valid.

5.3.3 Sim-Score evaluation results

The following section discusses the results of the evaluation of the Sim-Score. As previously mentioned, the calculation of the Sim-Score does not require any additional information, such as a predefined ground truth. Thus, the Sim-Score is suitable to be calculated for any log data set. Furthermore, to compare the character-based template generators with a token-based approach, we also generated token-based templates, using the part of the token_char algorithm that generates the token-structure of the template.

Tables 5.2, 5.3 and 5.4 present the evaluation results of the Sim-Score for the different template generator algorithms using the data sets described in Tab. 5.1. As expected,

[4]Since gaps can be optional they do not influence the Sim-Score.

	merge	length	equalmerge	token_char	token
Sim-Score	96.38%	96.24%	96.37%	95.18%	85.27%
Time (s)	435.20	23.46	25.52	29.54	8.49

Table 5.2: Sim-Score comparison on DS-A [WHL+20].

	merge	length	equalmerge	token_char	token
Sim-Score	91.40%	90.71%	91.42%	91.42%	77.27%
Time (s)	35179.35	55.56	63.37	843.51	366.76

Table 5.3: Sim-Score comparison on DS-B [WHL+20].

	merge	length	equalmerge	token_char	token
Sim-Score	71.96%	70.41%	71.95%	71.96%	52.67%
Time (s)	11207.57	344.21	227.22	1387.87	154.14

Table 5.4: Sim-Score comparison on DS-C [WHL+20].

the proposed character-based algorithms yield a much higher Sim-Score than the token-based approach. However, the token_char provides comparable results to the pure character-based algorithms. The differences between the Sim-Scores of the character-based algorithms are so small that they can be neglected. Nevertheless, the results of the runtime are of greater significance. The merge algorithm shows the longest runtime among all tested algorithms. This is the case, because all other character-based algorithms first divide the line into shorter segments by marking parts of the line that are equal to tokens of the template. Then, they match the remaining shorter parts of the line and the template by calculating the LV-distance. Whereas, the merge algorithm calculates the LV-distance for the whole log line and the whole template. While the length and the equalmerge algorithms showed a comparable runtime on the data sets DS-A and DS-B, the equalmerge algorithm outperforms all the others on DS-C on runtime. Due to the lower similarity threshold during clustering and larger variable parts in the log data, Sim-Scores for DS-C decrease for all algorithms. Furthermore, the larger variable parts in DS-C are the reason, why the equalmerge algorithm outperforms the length algorithm. While the equalmerge algorithm merges the blocks that are not marked and then calculates the LV-distance, the length algorithm first has to localize all blocks of the current template in the log line at hand. Due to the large variable parts the number of blocks the template consists of increases in every step. Furthermore, the different sizes of the data sets affect the token_char approach more than the others. The reason for this is, that the token_char algorithm has to do the matching for the token-structure and all character-structures of the template. The runtime of the pure token-based approach is rather long when processing DS-B. This is, because of the long lines consisting of a large number of tokens.

5.3.4 Scalability

The next section summarizes results on the evaluation of the scalability of the different algorithms. Figure 5.8 visualizes the results for the different algorithms, showing the number of lines on the x-axis and the runtime on the y-axis. For the evaluation of the scalability, we chose a cluster from DS-B that comprises more than 1000 log lines. Then, we measured the runtime it took to calculate the template for the cluster for 5 to 1000 lines in steps of 50 lines. Figure 5.8 demonstrates that the runtime of all algorithms scales linearly with respect to the cluster size m, which results in a computational complexity of $O(m)$. Figures 5.8b and 5.8c demonstrate that the length and the equalmerge algorithm scale equally well with respect to the runtime and gradient, followed by the token_char algorithm in Fig. 5.8d. The merge algorithm, see Fig. 5.8a, has the worst runtime and gradient.

5.3.5 Cluster arrangement

We also investigated the impact of the order of the log lines in a cluster on the resulting template and the process of generating it. Therefore, we changed the order of the log lines in the clusters as follows:

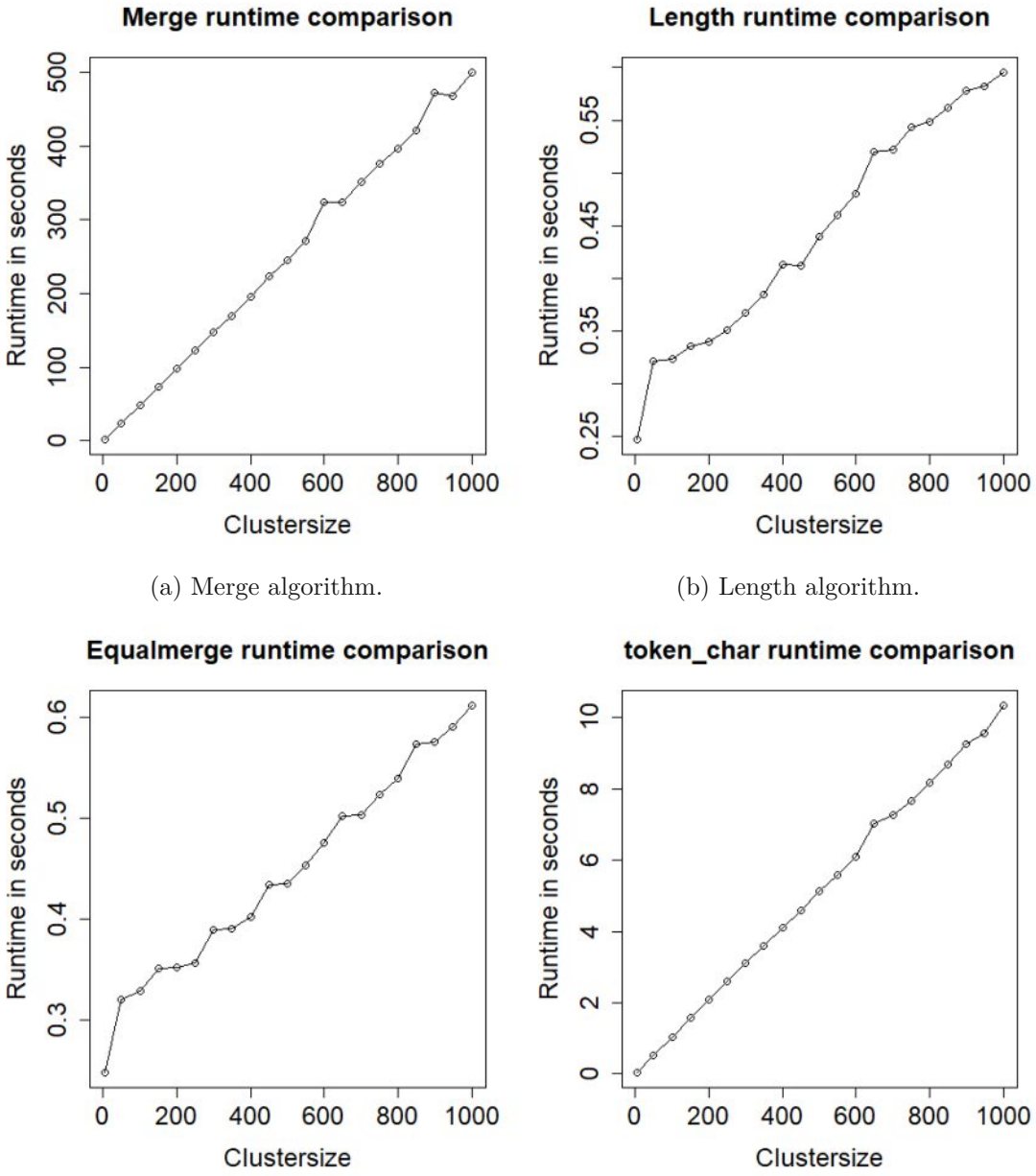
(i) The original order (original),

(ii) starting with the two lines that have the maximum LV-distance in the whole cluster and the following lines have the original order (maxfirst),

(iii) ordering the lines by the LV-distance to each other starting with the line that has the largest distance to the others (maxdist),

(iv) ordering the lines by the LV-distance to each other starting with the line that has the smallest distance to the others (mindist).

Table 5.5 summarizes the results of the cluster arrangement evaluation carried out on the first 10.000 lines of the data set DS-A. The lower part of the Table shows after processing which percentage of log lines in a cluster the template does not change any more. Our evaluation proves, that the order has impact on the number of processed log lines after which the template does not change anymore and therefore on the runtime. Ordering the lines by the LV-distance to each other starting with the line that has the largest distance to the others (maxdist) showed the best results, closely followed by starting with the two lines that have the maximum LV-distance and the following lines have the original order (maxfirst). Those two approaches improve the runtime in opposite to keeping the original order, while using the mindist approach increases the runtime. However, there was virtually no impact on the Sim-Score as the upper part of Tab. 5.5 points out. Since ordering the lines within a cluster by their LV-distance is computational expensive with $O(n^n)$, where n is the number of lines, the runtime improvement can only be realized when the lines are already in the correct order.



(a) Merge algorithm.

(b) Length algorithm.

(c) Equalmerge algorithm.

(d) Token_char algorithm.

Figure 5.8: Runtime comparison [WHL+20].

	original	maxfirst	maxdist	mindist
merge Sim-Score	96.38%	96.44%	96.43%	96.47%
length Sim-Score	96.24%	96.41%	96.40%	96.26%
equalmerge Sim-Score	96.37%	96.42%	96.41%	96.44%
token_char Sim-Score	95.18%	96.38%	96.37%	96.38%
merge last change	82.76%	76.20%	71.35%	93.55%
length last change	82.06%	74.96%	70.57%	92.93%
equalmerge last change	82.21%	74.94%	70.66%	93.06%
token_char last change	70.87%	67.84%	67.25%	78.63%

Table 5.5: Cluster arrangement [WHL+20].

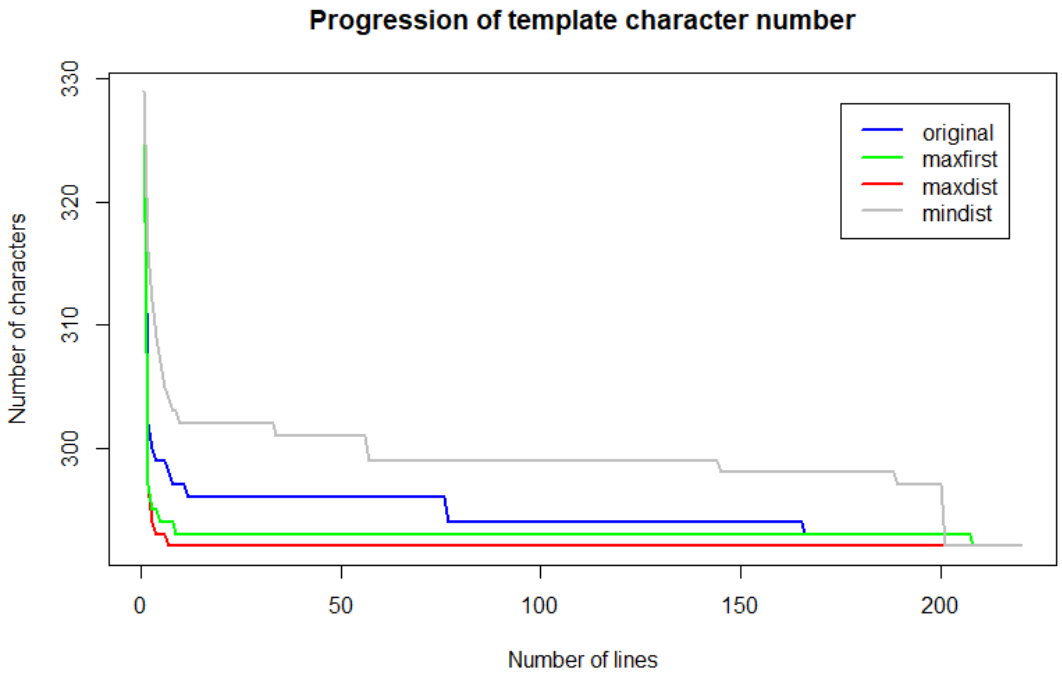


Figure 5.9: Progression of cluster template character number [WHL+20].

Additionally, Fig. 5.9 visualizes the progression of the change in the number of characters the template of a representative cluster consists of. Therefore, we plotted the number of characters the current template exists of over the number of processed lines for the four different cluster arrangements. The figure demonstrates that for the maxfirst and maxdist arrangement the template gets stable after a few lines, while the mindist arrangement, requires major changes in the template towards the end. The original arrangement lies between the others.

data size	10K	50K	1600K
merge Sim-Score	96.38%	95.79%	94.99%
length Sim-Score	96.24%	95.44%	94.40%
equalmerge Sim-Score	96.37%	95.71%	94.73%
token char Sim-Score	95.18%	93.53%	93.26%

Table 5.6: Evaluation of different datasets [WHL+20].

5.3.6 Evaluation of different data set sizes

In this section, we evaluate the influence of the data set size on the resulting templates. Therefore, we compared the Sim-Score of the whole data set DS-A, a subset consisting of the first 10.000 lines and a subset of the first 50.000 lines. The results, summarized in Tab. 5.6, indicate a small decline in the Sim-Score with increasing data set size. This can be explained as follows: The larger the data set, the more log lines are assigned to each cluster. Therefore, the similarity of the log lines within a cluster decreases, which as described in Sec. 5.3.4, affects the Sim-Score of the template. But, the lower Sim-Scores do not refer to templates of lower quality. Indeed, while the Sim-Score only slightly decreases, over-fitting is reduced. Hence, the quality of the templates actually increases, because of the more diverse set of log lines, which more accurately reflects the system behavior. Finally, we can conclude that the data set size does not strongly affect the quality of the resulting template.

5.3.7 Robustness

Furthermore, we evaluated the robustness of the algorithms, which is especially important for the length and the equalmerge algorithm. Since these two algorithms first mark parts of the template that equally occur in the currently processed log line, they imitate the longest common subsequence [GF13]. This might cause problems, if the lines within a cluster are different, but substrings in different positions are marked as equal, due to the fact that there are many variable parts in the log lines. Considering the strings `ayyaa`, `aayya`, `aaa`, the optimal template would be `a[*]a[*]a`, but because the first created template would be `[*]ayya[*]`, the final template becomes `[*]a[*]a[*]`, which leads to a lower similarity between the strings and the template.

Additionally, the localizing step in the length and equalmerge algorithm could be erratic, when the template includes two equal blocks, that only occur once in the currently processed log line. Therefore, a false marking can happen. For example, considering the strings `stringstring`, `string string` and `tring string`. The first two yield the template `string[*]string`, but because the algorithm localizes the first block in the rearmost part of the log line, the second block is marked with the empty string. Thus, the created template would be `[*]string[*]`, although `[*]tring[*]string` would be the the optimal one.

Hence, we ran the algorithms on the first 10k lines of data set DS-A, which was clustered

similarity	0.9	0.8	0.7	0.6	0.5
merge	96.38%	91.14%	82.76%	74.97%	71.09%
length	96.24%	90.48%	81.25%	73.76%	68.49%
equalmerge	96.37%	90.98%	82.34%	74.31%	70.64%
token_char	95.18%	90.47%	82.04%	73.85%	69.41%

Table 5.7: Robustness evaluation for different minimum similarities between log lines within a cluster [WHL+20].

using different similarity thresholds. The lower the threshold during clustering, the more dissimilar are the log lines within a cluster. In this way we can evaluate the effects of the marking step in the length and equalmerge algorithm, because which blocks are marked as substrings in the log line depends on the similarity of the log lines in a cluster. The effects can be seen when comparing the Sim-Score of the length and the equalmerge algorithm with the results of the merge algorithm, which does not include the marking step.

Table 5.7 demonstrates that there is no extensive decrease of the Sim-Score in either of the algorithms, which is only the case if the marking had a severe impact. Therefore, all of the algorithms can be considered robust.

5.3.8 *F*-score evaluation

Since the *F*-score requires a ground truth, we chose data sets from Hadoop and Thunderbird available on the Internet [Tea20], where these data sets each have 2000 lines and the corresponding token-based ground truths are also available. We created the character-base ground truths, which are the optimal template, based on the token-based ground truths.

The *F*-score was calculated for each algorithm as described in Sec. 5.3.2. Furthermore, we also tested the token-based ground truth against the character-based one. Since the token-based ground truth (token GT) is the optimum which token-based template generators can achieve, the resulting *F*-Score is the maximum any token-based approach can reach.

Table 5.8 presents the results of the *F*-score evaluation. The evaluation proves that all character-based algorithms yield more accurate templates than a token-based ever could. Merge, equalmerge and token_char provide the best *F*-score, followed by the length algorithm and the token ground truth. The *F*-scores of merge and equalmerge are the same, because they both created the same templates for these sets of log data. The token_char algorithm also had the same *F*-score, but yielded different templates, because it placed the gaps differently.

	merge	length	equalmerge	token_char	token GT
H F-score	0.9910	0.9902	0.9910	0.9910	0.8853
TB F-score	0.9958	0.9941	0.9958	0.9958	0.9296

Table 5.8: Test against character-based ground truth: H F-score is the F-score for the Hadoop data set and TB F-score the F-score for the thunderbird data set [WHL+20].

Algorithm	Performance	Accuracy
merge	- -	++
length	+	+
equalmerge	+	++
token_char	∼	+
token	++	- -

Table 5.9: Comparison of performance and accuracy [WHL+20].

5.3.9 Feature Analysis

Finally, we assess the features of the different algorithms with respect to performance and accuracy, which are summarized in Tab. 5.9. The merge algorithm provides the most accurate template according to *Sim*-Score and F-score. However, it lacks performance and therefore should not be applied for time critical tasks. The length algorithm provides comparable accurate templates, while optimizing performance in opposite to the merge algorithm. The performance boost is achieved by marking blocks of the current template that occur as substrings in the log line. Therefore, the length of the strings for which the LV-distance has to be calculated, can be significantly reduced. The equalmerge algorithm combines the length and the merge algorithm and performs almost as good as the length algorithm, while providing templates that are almost as accurate as the ones computed by the merge algorithm. The token_char algorithm performs slightly better than the merge algorithm, but is surpassed by the performance of the length and equalmerge algorithms. Moreover, the templates provided by the pure character-based approach are more accurate. Hence, we recommend for any application to apply the equalmerge algorithm instead of the token_char approach. The pure token-based approach shows the best performance, while providing the least accurate templates. Additionally, all the disadvantages mentioned in the beginning of this chapter have to be considered when applying token-based approaches for generating templates.

5.4 Outlook and further development

The proposed approach for generating character-based templates for log data extends clustering approaches, such as the bioclustering presented in Ch. 3 and the incremental clustering introduced in Ch. 4. The templates provide meaningful descriptions for the content of the log lines within a cluster. Furthermore, the approach solves the problem

of computing multi-line alignments for any kind of text. Therefore, it allows to compute more accurate templates than token-based approaches do.

Log line templates have many different application cases. First, they provide meaningful and essential information for system administrators and security analysts by summarizing the content of log line clusters. Moreover, log line templates can be used to generate test data as the BAESE approach presented in Sec. 7.3 proves. Furthermore, the templates can be directly applied for intrusion detection as signatures, when, for example, transformed to regular expressions. Additionally, the same way they can be applied as log parsers and therefore enable further analysis. However, log parsers applying lists of regular expressions often suffer from a lack of performance for parsing, due to a complexity of $O(n)$, where n is the length of the list. This fact often makes online parsing of log data impossible, which consequently also harms the runtime performance of log analysis tasks, such as anomaly detection. Hence, the next chapter proposes a novel tree-based parsing approach that reduces the complexity of parsing to $O(\log(n))$. Additionally, we introduce a parser generator approach that allows to automatically create such such tree-based log data parsers.

CHAPTER 6

A tree-based log parser generator to enable log analysis

While Ch. 3 and Ch.4 describe novel procedures for efficiently clustering large amounts of log data, Ch. 5 provides a novel approach for generating character-based log line templates. The latter processes pre-clustered log data and enables log line parsing, which enables further analysis possible, such as log event classification and rule-based anomaly detection. However, using lists of log line templates for parsing is rather inefficient. Hence, the following chapter proposes a novel highly efficient tree-based log parser approach and provides a solution for automatic parser generation to enable log event classification and further log analysis operations. Major parts of the remaining chapter have been published in [WLSK19].

Log data occurs in form of unstructured text lines that describe a certain system or network event. Thus, log parsing is an important task prior to log analysis. A log parser knows the syntax, i.e. unique structure, of the data produced by a monitored system or service. Log parsers carry out preprossessing steps to enable further analysis, such as signature and rule verification or anomaly detection. Therefore, parsers sanitize time stamps, disassemble log lines into meaningful tokens, e.g., white-space separated strings, assign an event type to each line and filter out lines that are irrelevant for further analysis.

However, the following major challenges occur when parsing log data: First, today's modern systems and networks produce large amounts of log data, up to several thousands lines per second in a medium-sized infrastructure. Thus, parsing log lines must be highly efficient to enable online log analysis, which is especially necessary for critical tasks, including intrusion detection and safety monitoring. Current log parser approaches apply sets of distinct regular expressions to parse log data. Especially, in large and complex networks large amounts of different log event types can occur and each requires separated regular expressions. Hence, this process is quite inefficient, with a computational

complexity of $O(n)$ per log line, where n is the number of regular expressions. While this is acceptable for forensic analysis, it is not for online analysis, especially when it is carried out on the host. Second, each device and network is unique and therefore shows a unique system behavior, because of the users who operate it and the services and applications it runs. Hence, every system needs specific parsers. Furthermore, the complexity of today's networks increases fast and technologies evolve quickly. As a result, also logging infrastructures and the syntax of log lines changes frequently. Consequently, it is a cumbersome and time consuming task to define and maintain log parsers manually. Finally, to provide an efficient parsing process, most state of the art parsers dissect log lines rudimentary, meaning, they, for example, only separate timestamp, host name and message, or parse only specific information such as timestamps, host names and IP addresses. This makes it hard to analyze information stored in the log message and leads to a loss of information. In this chapter, we present the following contributions to address these challenges:

(i) A tree-like parser that could be seen as a single very large regular expression that models a system's log data. During parsing a log line, the parser leaves out irrelevant parts of the model and reduces the complexity for log line parsing to $O(\log(n))$.

(ii) AECID-PG, a density-based [Vaa03] log parser generator approach that automatically builds a tree-like log parser. In opposite to many other parser generator approaches, AECID-PG does not rely on distance metrics. Instead, it uses the frequency with which log line tokens occur.

Since, AECID-PG does not rely on the semantics of the monitored log data, it can be applied in any domain to any log data that has static syntax. Furthermore, the tree-like structure of the parser allows to reference log line parts that include interesting information efficiently, using the relating path of the parser tree. This simplifies accessing information in log lines and speeds up further analysis of log data, such as rule and signature verification, and does not lead to a loss of information stored in log lines before analysis.

The remainder of the chapter structures as follows: First Sec. 6.1 introduces the concept behind the tree-like parser. Next, Sec. 6.2 describes the density-based parser generator approach. Afterwards, Sec. 6.3 evaluates the tree-based parser generator approach in details. Finally, Sec. 6.4 provides an outlook on further development.

6.1 Tree-based parser concept

We propose a novel log data parser approach that leverages a tree-like structure and takes advantage of the inherent structure of log lines [WSSF18, WLSK19]. Currently, most log parsers simply apply a set of regular expressions to process log data. The set

describes all possible log events and log messages, when the monitored system or service runs in a normal state. Each regular expression looks for static and variable parts that are usually separated by white-spaces, and describes one type of log event or log message. Regular expressions applied in parsers can be depicted as templates. For example, in the template `Connection from * to *`, `Connection`, `from` and `to` are static and `*` are variable. Those templates are generated applying clustering (cf. Ch. 4) and template generators (cf. Ch. 5). Subsequently, to parse log data, all of these regular expressions are applied in the same order to each log line separately until the line matches a regular expression. This procedure is inefficient, with a complexity of $O(n)$ per log line, where n is the number of regular expressions.

The proposed tree-based parser approach aims at reducing the complexity of parsing and therefore increasing the performance. Since there are no commonly accepted standards, and industry best practices only define certain aspects of log syntax, developers may freely choose the structure of log lines produced by their services or applications. For example, the syslog [Ger09] standard dictates that each log line has to start with a time stamp followed by the host name. However, the remainder of the syntax can be chosen without any restrictions. It is noteworthy that log lines usually consist of static and variable tokens, which are separated by delimiters, such as white-spaces, semicolons, equal signs, or brackets.

Applying standards, such as syslog, causes log lines produced by the same service or application to be similar in the beginning but differ more towards the end of the lines. Consequently, modeling a parser as a tree, leads to a parser tree that comprises a common trunk and branches towards the leaves, see Fig. 6.1. The parser tree represents a graph theoretical rooted out-tree. This means, during parsing, it processes log lines token-wise from left to right and only parts of the parser tree that are relevant for the log line at hand are reached. Hence, this type of parser avoids passing over the same log line more than once as would be done when applying distinct regular expressions. As a result, the complexity for parsing reduces from $O(n)$ to $O(\log(n))$. Eventually, each log line relates to one path, i.e. branch, of the parser tree.

Figure 6.1 visualizes a part of a parser tree for ntpd (Network Time Protocol daemon) logs (see Fig. 6.2). This example demonstrates that the tree-based parser consists of three main building blocks. The nodes with bold lines represent tokens with static text patterns. This means that in all corresponding log lines, a token with this text pattern has to occur at the position of the node in the tree. For example, the first node represents the service name, which in this case of syslog data, has to occur in all log lines generated by the ntpd service, after a preamble consisting of a timestamp and the hostname. Oval nodes represent nodes that allow variable text until the next separator or static pattern along the path in the tree occurs. For example, the second node relates to the process ID (PID), which is variable and separated by square brackets. The third building block is a branch element. The parser tree branches, when in a certain position only a small number of different tokens with static text occur. This is the case, for example, when a component generates log lines for different events, as in Fig. 6.1 after the third node.

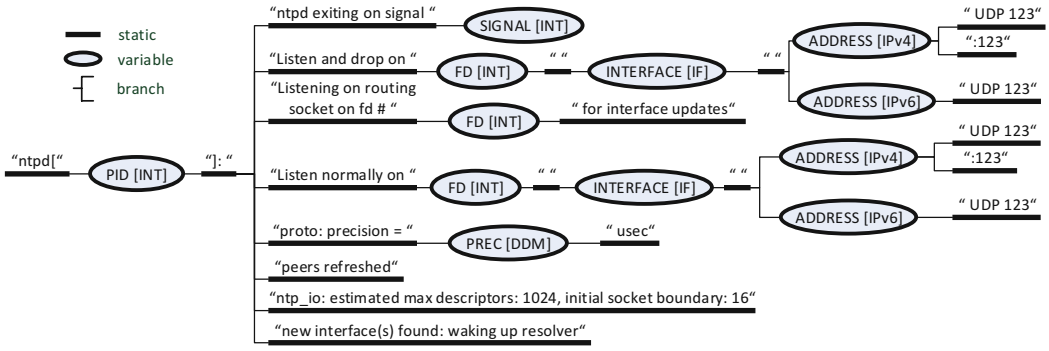


Figure 6.1: The tree describes the parser model for ntpd (Network Time Protocol) service logs as shown in Fig. 6.2. Strings under quotes over bold lines are static elements. Oval entities allow variable values, bold lines mark static parts of the data and forks symbolize branches. An example of a parsed log-line is provided in Fig. 6.3 [WSSF18].

```
0: Jun 14 16:17:12 ghive-ldap ntpd[16721]: Listen and drop on 0 v4wildcard 0.0.0.0 UDP 123
1: Jun 14 16:17:12 ghive-ldap ntpd[16721]: Listen and drop on 1 v6wildcard :: UDP 123
2: Jun 14 16:17:12 ghive-ldap ntpd[16721]: Listen normally on 2 lo 127.0.0.1 UDP 123
3: Jun 14 16:17:12 ghive-ldap ntpd[16721]: Listen normally on 3 eth0 134.74.77.21 UDP 123
4: Jun 14 16:17:12 ghive-ldap ntpd[16721]: Listen normally on 4 eth1 10.10.0.57 UDP 123
5: Jun 14 16:17:12 ghive-ldap ntpd[16721]: Listen normally on 5 eth1 fe80::5652:ff:fe5a:f89f UDP 123
6: Jun 14 16:17:12 ghive-ldap ntpd[16721]: Listen normally on 6 eth0 fe80::5652:ff:fe01:1aee UDP 123
7: Jun 14 16:17:12 ghive-ldap ntpd[16721]: Listen normally on 7 lo ::1 UDP 123
8: Jun 14 16:17:12 ghive-ldap ntpd[16721]: peers refreshed
9: Jun 14 16:17:12 ghive-ldap ntpd[16721]: Listening on routing socket on fd #24 for interface updates
```

Figure 6.2: Example of ntpd service logs [WSSF18].

Figure 6.3 shows an example of a parsed log line from Fig. 6.2. The figure demonstrates that the parser can consist of several components, named `ModelElements`, which refer to the nodes of the parser tree. It shows, that the variable nodes can have different properties, such as allowing only integers or IP addresses. The most frequently used are[1]:

- `FixedDataModelElement`: Match a fixed (constant) string.
- `FirstMatchModelElement`: Branch the model taking the first branch matching the remaining log line.
- `AnyByteDataModelElement`: Match anything till the end of a log line.

[1] A more exhaustive list of model elements can be found in the AMiner (which is an agent that can apply the parser) documentation at: https://github.com/ait-aecid/logdata-anomaly-miner/blob/master/source/root/usr/share/doc/logdata-anomaly-miner/aminer/ParsingModel.txt [last accessed: 9/27/2019]

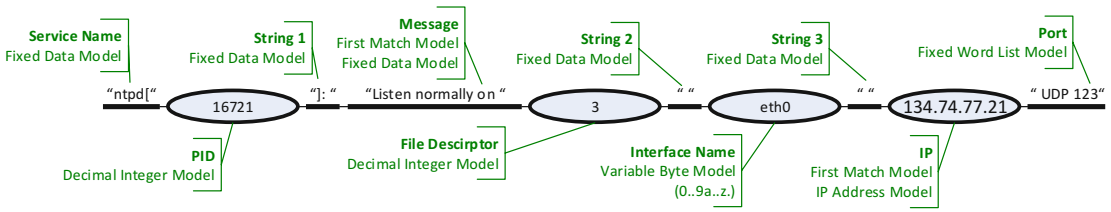


Figure 6.3: Example of log line parsing (cf. Fig. 6.2 line number 3 and Fig. 6.1) [WSSF18].

- `DateTimeModelElement`: Simple datetime parsing.
- `DecimalIntegerValueModelElement`: Parsing integer values.
- `IpAddressDataModelElement`: Match an IPv4 address.
- `SequenceModelElement`: Match all the sub-elements exactly in the given order.
- `FixedWordlistDataModelElement`: Match one of the fixed strings from a list.
- `VariableByteDataModelElement`: Match variable length data encoded within a given alphabet.

In a nutshell, applying a tree-like parser model provides the following advantages, regarding performance and quality of log analysis:

(i) In opposite to an approach that applies distinct regular expressions, a tree-based parser avoids to pass over the same data entity more than once, because it follows for each log line one path of the parser tree, in the graph-theoretical tree that represents the parser, and leaves out irrelevant model parts.

(ii) Because of the tree-like structure, the parser model could be seen as a single, very large regular expression that models a system's log data. Therefore, the computational complexity for log line parsing is more like $O(\log(n))$ than $O(n)$ when handling data with separate regular expressions.

(iii) The tree-like structure allows to reference all the single tokens with an exact path as Fig. 6.4 demonstrates. Thus, parsed log line parts are quickly accessible so that rule checks can just pick out the data they need without searching the tree again. Furthermore, it allows to quickly apply anomaly detection algorithms to the different tokens and to correlate the information of different tokens within a single line and across lines.

```
Jun 14 16:17:12 ghive-ldap ntpd[16721]: Listen normally on 3 eth0 134.74.77.21 UDP 123
/model/syslog/time: 'Jun␣14␣16:17:12'
/model/syslog/host: 'ghive-ldap'
/model/services/ntpd/sname: 'ntpd['
/model/services/ntpd/pid: '16721'
/model/services/ntpd/s1: ']:␣'
/model/services/ntpd/msg/text: 'Listen␣normally␣on␣'
/model/services/ntpd/msg/descriptor: '3'
/model/services/ntpd/msg/s2: '␣'
/model/services/ntpd/msg/if: 'eth0'
/model/services/ntpd/msg/s3: '␣'
/model/services/ntpd/msg/ipv4/ip: '134.74.77.21'
/model/services/ntpd/msg/ipv4/port/: '␣UDP␣123'
```

Figure 6.4: Path model of log line 3 from Fig. 6.2 [WSSF18].

6.2 AECID-PG: tree-based log parser generator

AECID-PG implements a density-based parser generator approach, which uses token frequencies instead of a distance metric to determine whether patterns should be static or variable and if a branch element is required. However, the main difference to existing approaches is that this computation is carried out locally in every node of the generated parser tree, rather than for all log lines.

In the remaining section, we will use Fig. 6.5 to explain our analytical model to generate log parsers. For convenience, the tree represents synthetic log data that includes log lines such as `T D X I Z`, where `T` represents the time stamp of the line. Each line is split into tokens separated by white-spaces. The example line would be split into the tokens `T`, `D`, `X`, `I`, `Z`. Assuming that tokens `X` and `Z` represent variable parts of the log line, the related path of the parser tree includes also variable nodes. Hence, in Fig. 6.5, letters represent tokens with static and stars tokens with variable patterns.

6.2.1 Challenges when generating tree-like parsers

The simplest method to automatically build a tree-like parser for log data is to use a set of training log lines that represents the normal system behavior and define a tree, where all parts of the log lines are considered static. Therefore, all nodes represent static patterns, and the tree includes all possible unique paths occurring in the log data. Thus, the parser generator creates branches when it registers a sequence of tokens, which is not yet present in the parser tree. For example, in Fig. 6.5 in column Y_3, the parser generator creates a branch at the top node that represents token `E`. Before, this path represented only log lines with the token sequence `T A E G`. Once, the parser generator adds the branch element, and the nodes `H` and `J` it can also parse log lines with token sequence `T A E H J`. Building a parser tree like that results in a parser, which would perfectly parse the training log data, but if one applies it to other log data, even if it origins from the same system, many log lines would be unparsed, i.e. not reflected accurately by this parser tree. Reasons for this are that (i) unique log line parts, such as IDs and

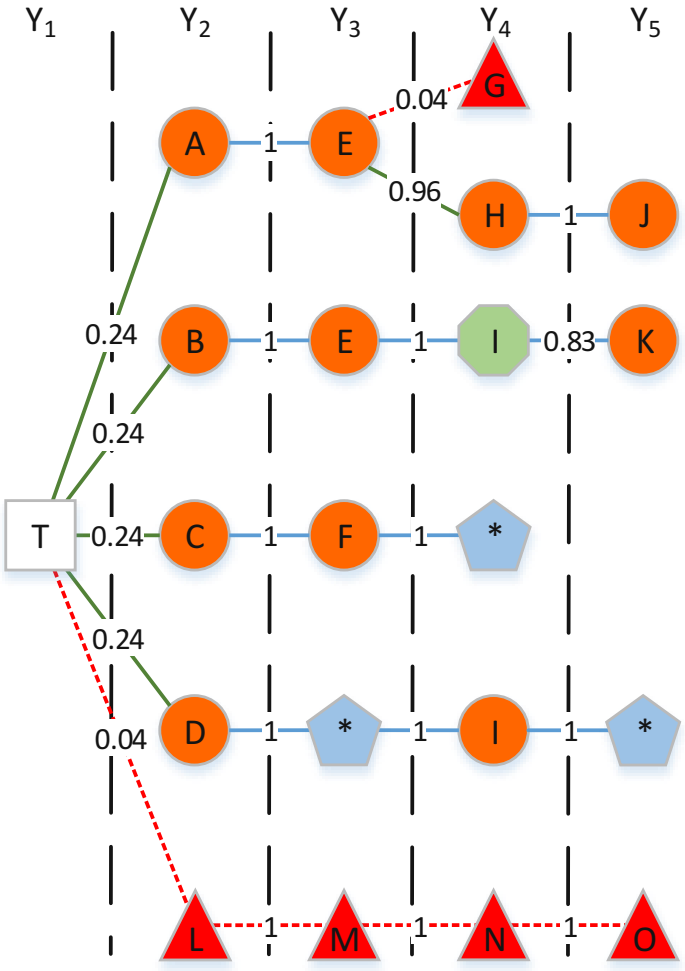


Figure 6.5: A synthetic parser tree. The square node represents the preamble including the time stamp `T`, orange circles static nodes, blue pentagons variable nodes, red triangles nodes that occur too rarely to be part of the parser, and green hexagons optional nodes, where log lines optionally can end [WLSK19].

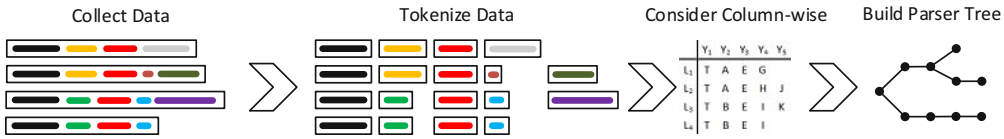


Figure 6.6: AECID-PG process flow [WLSK19].

time stamps, and (ii) highly variable parts, such as sensor values, are considered static. Thus, the resulting parser would over-fit the training data and could not be practically applied due to its complexity. To avoid an over-fitting parser, AECID-PG applies a set of rules to decide whether it should create a node that represents static text, a node that allows variable text, or a branch into more than one node that represent static text. Also paths that occur too rarely, such as between E and G in the top of column Y_3 and Y_4 are omitted by the parser generator due to the fact they are outliers and therefore are not part of the normal system behavior.

6.2.2 AECID-PG concept

Figure 6.6 visualizes the concept of the AECID-PG approach. In the following, we assume that the parser generator processes log lines in one batch. The approach basically splits into four steps: (i) Log data is collected. We consider textual log data which one or more computer systems or network components produce sequentially in form of log lines. (ii) Each log line is tokenized, i.e. split into meaningful strings. Therefore, a predefined list of delimiters is used that can include symbols such as white-spaces, colons, equal signs, brackets, etc. The tokens form the basis to build the parser tree, because they define the nodes of the tree. (iii) The data is transformed into a table, where column Y_i stores a list of the i-th token of the log lines. AECID-PG processes the data column-wise instead of line by line, to improve the runtime of the parser generator. This is faster, because the algorithm applies hash-tables for this purpose and the maximum number of tokens per log line is usually significantly lower than the number of lines the trainings data set consists of. (iv) The algorithm builds the parser tree. Therefore, nodes of tree-depth i correspond to tokens in column Y_i, as also shown in Fig. 6.5. An edge between two consecutive nodes can only exist, if the corresponding tokens at least once occur consecutively in the same log line. The next section describes how the algorithms decides, which kind of node, i.e., static, branch, variable, etc., it generates.

6.2.3 AECID-PG rules

AECID-PG applies four rules to build a parser tree and to determine the properties of a node. To describe these rules, we define the path-frequency PF_{ij}^k, which describes the frequency by which node n_i^k from column Y_k reaches node n_j^{k+1} in column Y_{k+1} (cf. Eq. (6.1)), where $|n_i^k|$ defines the number of lines of the trainings set that reached node n_i^k, $k = 0, \ldots, m$ stands for the column number, i.e. tree depth, and $i = 0, \ldots, p$ corresponds with the index of the nodes in column Y_k and $j = 0, \ldots, q$ with the index of the nodes in

column Y_{k+1}. We assume that the path-frequency is only calculated between consecutive nodes that are linked with an edge e_{ij}^k, i.e. path.

$$PF_{ij}^k = \frac{|n_j^{k+1}|}{|n_i^k|} \tag{6.1}$$

In the following, we assume the algorithm builds the parser tree for one column after another, starting with Y_1. All of the following steps are applied to all $n_i^k \in Y_k$, with $i = 0, \ldots, p$. This means also that the following steps are carried out for each node, i.e. the algorithm has only to consider the remaining log lines described by the path of the current node. First, the algorithm applies the previously described simplest approach for the current column Y_{k+1}. This means, it keeps all unique tokens as nodes with static patterns. After the initialization of Y_{k+1}, it applies the following rules to refine the tree in the current column. Hence, the algorithm decides whether nodes with static or variable patterns are required, and whether the parser tree needs a branch or not.

Rule 1. When starting from node n_i^k, if there is no node n_j^{k+1}, with existing e_{ij}^k and PF_{ij}^k greater than or equal to θ_1, with $\theta_1 \in [0, 1]$, the algorithm creates a node with a variable pattern, i.e., the parser allows any input (cf. Eq. (6.2), where VAR stands for a node with a variable pattern).

$$\{n_j^{k+1} : \exists e_{ij}^k \wedge PF_{ij}^k \geq \theta_1\} = \emptyset \Rightarrow VAR \tag{6.2}$$

Rule 1 ensures that the algorithm avoids generating nodes with static patterns for tokens that occur rarely in the log data and therefore would lead to an over-fitting parser. In Fig. 6.5, this is represented by the blue pentagonal nodes with a star inside.

Rule 2. The second rule is evaluated if there exists exactly one of the generated nodes n_j^{k+1}, with existing e_{ij}^k and PF_{ij}^k greater than or equal to θ_1, i.e. $|\{n_j^{k+1} : \exists e_{ij}^k \wedge PF_{ij}^k \geq \theta_1\}| = 1$. Rule 2 distinguishes the following two cases:

(a) If $n_j^{k+1} \in \{n_j^{k+1} : \exists e_{ij}^k \wedge PF_{ij}^k \geq \theta_1\}$ additionally satisfies Eq. (6.3), the algorithm generates a single successive node n_j^{k+1} of n_i^k, with a static pattern, that only allows the text of the corresponding token.

$$PF_{ij}^k \geq \theta_2, \text{with } \theta_2 \in [0, 1] \tag{6.3}$$

(b) If $n_j^{k+1} \in \{n_j^{k+1} : \exists e_{ij}^k \wedge PF_{ij}^k \geq \theta_1\}$ does not satisfy Eq. (6.3), the algorithm creates a node with variable pattern VAR, i.e. the parser allows any input.

Rule 2 ensures that the algorithm does not build a parser model that rejects too many log lines, if the path-frequency to only one node exceeds θ_1, because, for example, if $\theta_1 = 0.1$, the algorithm could reject up to 90% of the log lines that reached the preceding

node. Therefore, the path-frequency to this node has to exceed a second higher threshold θ_2. Figure 6.5 provides an example for Rule 2 in line one between column Y_3 and Y_4. Assuming $\theta_1 = 0.1$ and $\theta_2 = 0.9$, the path-frequency to the upper node `G` does not exceed θ_1 and therefore the node is marked with a red triangle and omitted in the final parser tree. On the other hand, the path-frequency to the lower node `H` exceeds θ_1 and θ_2 and therefore the node is marked with an orange circle and is part of the final parser tree as node representing a static text pattern.

Rule 3. The third rule is evaluated if there exist more than one of the generated nodes n_j^{k+1}, with existing e_{ij}^k and PF_{ij}^k greater than or equal to θ_1, i.e. $|\{n_j^{k+1} : \exists e_{ij}^k \wedge PF_{ij}^k \geq \theta_1\}| > 1$.

Rule 3 distinguishes the following two cases:

(a) If $n_j^{k+1} \in \{n_j^{k+1} : \exists e_{ij}^k \wedge PF_{ij}^k \geq \theta_1\}$ additionally satisfies Eq. (6.4), where $J = \{j = 0, \ldots, q : n_j^{k+1} \in \{n_j^{k+1} : \exists e_{ij}^k \wedge PF_{ij}^k \geq \theta_1\}\}$ is the set of the indexes of the nodes that satisfy Rule 1, the algorithm generates successive nodes n_j^{k+1} of n_i^k for all $n_j^{k+1} \in \{n_j^{k+1} : \exists e_{ij}^k \wedge PF_{ij}^k \geq \theta_1\}$, with a static pattern, that only allows the text of the corresponding token.

$$\sum_{j \in J} PF_{ij}^k \geq \theta_3, \text{with } \theta_3 \in [0, 1] \tag{6.4}$$

(b) If $n_j^{k+1} \in \{n_j^{k+1} : \exists e_{ij}^k \wedge PF_{ij}^k \geq \theta_1\}$ does not satisfy Eq. (6.4), the algorithm creates a node with variable pattern VAR, i.e., the parser allows any input.

Similarly to Rule 2, Rule 3 ensures that the algorithm does not build a parser tree that rejects too many log lines. For example, if $\theta_1 = 0.1$, the algorithm could reject up to 80% of the log lines that reached the preceding node, if only 2 nodes have higher path-frequencies than θ_1. Thus, additionally the sum of the path-frequencies to the nodes, which exceed θ_1, has to exceed also a higher threshold θ_3. In Fig. 6.5, the transition between Y_1 and Y_2 provides an example for Rule 3. Assuming $\theta_1 = 0.1$ and $\theta_3 = 0.95$, the sum of the path-frequencies to the orange circled nodes, representing nodes corresponding to static text patterns, which each exceeds θ_1, exceeds θ_3. If that would not be the case a pentagonal blue node, representing a node corresponding to a variable pattern, would have been generated.

Since, some log lines might end before the path ends, rule 4 is required.

Rule 4. The fourth rule is evaluated, if some log lines end in a node, i.e. before the path ends, and all others succeed. Rule 4 evaluates the following two cases:

(a) If the ratio of lines that end in n_i^k is higher than $\theta_4 \in [0, 1]$, the algorithm generates all succeeding nodes as optional nodes, i.e. lines can either end before, or reach all succeeding nodes. Otherwise, all lines have to succeed or are considered unparsed.

(b) If the ratio of lines that do not end in n_i^k is lower than $\theta_5 \in [0, 1]$, the path ends in node n_j^k and there are no succeeding nodes. Otherwise, either Rule 4a is true or all lines have to succeed.

Note that θ_4 always has to be greater than or equal to θ_5. In Figure 6.5, in column Y_4 the top third green octagonal node provides an example for Rule 4. Assuming $\theta_4 = 0.1$ and $\theta_5 = 0.8$, it is possible that optionally some lines end in this node and some exceed it till the end of the path.

6.2.4 Features

The remaining section summarizes AECID-PG's most important features. First of all, while most log parser generators only use white-spaces to tokenize log data, AECID-PG provides the option to freely choose a delimiter and even to define a list of delimiters. Hence, AECID-PG adapts better to log data with different properties and therefore is broadly applicable.

Furthermore, AECID-PG considers path-frequencies locally in each node. Thus, two paths in the parser tree that represent two independent log line classes do not influence each other. Furthermore, it is easier for the parser generator to create branches the farther away the nodes are from the root node, i.e., the higher the current tree-depth is. This suits the fact, that log lines are more similar in the beginning than in the end. For example, a syslog line usually starts with time stamp, host name, and in most times service name, before the structure and the content become looser [Ger09].

However, to ensure that the thresholds θ_1, θ_2, θ_3, θ_4 and θ_5 are globally correct and with increasing tree depth IDs do not become nodes with static patterns, which would make the parser inapplicable, for log data that differs from the training data, AECID-PG includes an optional damping mechanism. The damping mechanism is a function that increases thresholds θ_i in relation to the current tree depth k, and applies the damping constant Δ (see. Eq (6.5), where $|n_i^k|$ is the number of lines that reached node n_i^k).

$$\theta_{i_{k+1}} = \theta_{i_k}(1 + \Delta), \quad \Delta = 1 - \frac{|n_j^{k+1}|}{|n_i^k|} \tag{6.5}$$

Moreover, AECID-PG is able to detect predefined patterns, that correspond to the ones the AMiner [WSSF18] (see Sec. 7.2), a log sensor for anomaly detection that leverages a tree-based parser, applies, such as IP addresses, date times, integers, or specified alphabets. These nodes are similar to nodes that allow variable patterns. However, they demand for certain properties of the parsed log line parts.

	BGL	HPC	HDFS	Zookeeper	Proxifier
Line Length	10-102	6-104	8-29	8-27	10-27
#Templates GT	112	44	14	46	7
#Templ. AECID-PG	120	17	10	17	9
θ_1	0.05	0.05	0.02	0.2	0.05
$\theta_2 = \theta_3$	0.6	0.9	0.9	0.9	0.95
$\theta_4 = \theta_5$	0.01	0.01	0.01	0.01	0.01
Δ	0.1	0.1	0.1	0.01	0.01
Delimiters	‘ ’	‘ ’, =, (,)	‘ ’	‘ ’, @	‘ ’, (,)

Table 6.1: Experimental data: number of words per log line, number of templates in the ground-truth, number of templates generated with AECID-PG, AECID-PG input parameters [WLSK19].

6.3 Evaluation

The following section discusses the evaluation of AECID-PG[2]. The section describes the real world data we used for the application, the calculation of the F-Score and its results, approaches we compared with AECID-PG and a real world application example. We do not evaluate the performance of AECID-PG, because generating a parser is not a time critical task. Furthermore, the parser approache reduces the complexity of parsing from $O(n)$ to $O(\log(n))$ by definition.

6.3.1 Experimental data

For the evaluation of AECID-PG, we used five real-world data sets: (i) logs from the supercomputer system BlueGene/L (BGL)[OS07], (ii) HPC logs from a high performance cluster, which has 49 nodes with 6,152 cores and 128GB memory per node [LLC], (iii) logs from a 203-node cluster on Amazon EC2 platform (HDFS)[XHF+09], (iv) logs from Zookeeper installed on a cluster with 32-nodes [Tea20] and (v) logs form the standalone software Proxifier [Tea20]. Table 6.1 describes important properties of the data and demonstrates the complexity of the data sets. The table shows that the data sets are significantly different regarding length of log lines with respect to white-space separated words (excluding time stamps), and number of templates in the ground-truth, which relates to the number of different events logged in the log files (cf. Tab. 6.1).

6.3.2 F-score evaluation and comparison with other approaches

For the evaluation of the accuracy of AECID-PG, we calculated the F-Score (cf. Eq. (4.21)) as shown in [MRS17] and compared AECID-PG to five other parser generator approaches: (i) SLCT (Simple Logfile Clustering Tool), a density-based clustering

[2]An implementation of AECID-PG can be found here: https://github.com/ait-aecid/aecid-parsergenerator [last accessed: 09/10/2020].

approach that generates log patterns [Vaa03], (ii) IPLoM (Iterative Partitioning Log Mining) applies a heuristic three-step hierarchical partitioning approach to generate templates [MZHM12], (iii) LKE (Log Key Extraction) that applies clustering and heuristics [FLWL09], (iv) LogSig uses word pair generation and clustering before it generates log templates [TLP11] and (v) Drain, a log parser generator using a fixed depth tree approach [HZZL17]. For the choice of input parameters, we oriented us on [HZZL17] and [Tea20]. The input parameters of AECID-PG (see Tab. 6.1) depend on the complexity of the input data. For example, a more complex data set requires a smaller θ_1, which makes it easier to generate branches.

For the F-Score evaluation, we randomly chose 2000 lines from each of the log files described in the previous section. The data, as well as implementations of the aforementioned log parser generator algorithms and their configurations are provided by [Tea20], who also provide a ground-truth for each log file that we leveraged to calculate the F-Score. For the calculation of the F-Score, first, all log lines of the experimental data have been assigned to the correct template of the ground-truth. Then, the parser generators have been applied to the data. Once the parser has been generated, for example in case of AECID-PG, we created a template for each path in the parser-tree, i.e., the path between root node and each leaf node, including optional nodes. Next, we assigned the log lines of the experimental data to the corresponding templates. Finally, we calculated the true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN), as [MRS17] describes: A *TP* decision assigns two lines which are assigned to the same template of the ground-truth also to the same template when considering the templates of the parser generator, a *TN* decision assigns two lines which are assigned to different templates of the ground-truth also to different templates of the parser generator, a *FP* decision assigns two lines which are assigned to different templates of the ground-truth to the same template of the parser generator, and a *FN* decision assigns two lines which are assigned to the same template of the ground-truth to different templates of the parser generator.

Table 6.2 summarizes the results of the F-Score evaluation. The input parameters we used for AECID-PG are given in Tab. 6.1. The F-Sore values demonstrate that parsers generated with AECID-PG are either more accurate than the parsers of the compared parser generators or at least comparably accurate. Furthermore, we calculated the average F-Score, which is stored in the last column in Tab. 6.2. AECID-PG achieves the highest average F-Score, which proves its broad applicability.

6.3.3 Application scenario

The remaining section describes an application scenario for AECID-PG. Specifically, we took an HDFS log file [XHF+09] of around 2 days, consisting of more than 11 million lines. We randomly chose 1% of the lines as training data and used AECID-PG to create a parser tree for HDFS logs. The input parameters we used are: $\theta_1 = 0.005$, $\theta_2 = \theta_3 = 0.95$, $\theta_4 = \theta_5 = 0.001$, $\Delta = 0.01$, and delimiters white-space and underscore. The selected thetas suite HDFS logs, because the data includes many different structured

	BGL	HPC	HDFS	Zookeeper	Proxifier	Avg
AECID-PG	0.9556	0.9626	0.9996	0.9487	0.8496	0.94322
SLCT	0.6355	0.8109	0.4143	0.8218	0.8707	0.71064
IPLoM	0.9999	0.6485	0.869	0.9995	0.8609	0.87556
LKE	0.4765	0.1793	0.9637	0.8224	0.8684	0.66206
LogSig	0.2653	0.8662	0.9493	0.9906	0.8467	0.783618
Drain	0.9896	0.8576	1	0.9995	0.8609	0.94152

Table 6.2: Comparison of F-score results for different parser generators and log files [WLSK19].

events that however also share many similar parts. Thus, a low θ_1 encourages many branches, while high θ_2 and θ_3 ensure that variable parts do not generate static nodes in the parser tree. Similarly to $theta_1$, low values of θ_4 and θ_5 allow log lines to end prematurely at optional nodes. During the training, 0.17% of the training data have not been represented by the resulting parser-tree, because these lines occurred too rarely and therefore have been considered as outliers. We then transformed the parser-tree into a parser for the AMiner[3] and applied it to the whole data set, which only consists of log lines representing normal system behavior. The result was that only 0.23% of the lines were unparsed. Furthermore, the parser processed 84.800 lines per second and it took 132 seconds to process the whole file on a workstation with an Intel Xeon CPU E5-1620 v2 at 3.70GHz 8 cores and 16 GB memory, running Ubuntu 16.04 LTS.

6.4 Outlook and further application

In this chapter, we presented a novel approach for a tree-based parser generator for textual computer log data that allows to generate parsers which improve the performance of parsing and enable online log analysis, such as anomaly detection. The tree-based structure of the parser, reduces the computational complexity for log line parsing enormously in comparison to the application of lists of regular expressions as applied by traditional log parser approaches. This is especially important in the area of cyber security to enable, for example, online anomaly detection and therefore, detection of attacks in real time. Furthermore, the tree-like structure of the parser allows to conveniently access information provided by log lines for further analysis, as the approach proposed in Sec. 7.2 shows. The evaluation demonstrated the broad applicability of the AECID-PG approach and demonstrated its functionalities in a real world scenario.

Currently, the parser generator processes the training data as batch, i.e. all log lines at once, and finally provides the tree-like parser. The parser can be applied, for example, with the AMiner[4] (see Sec. 7.2) a sensor for online anomaly detection [WSSF18]. The goal is to further develop AECID-PG so that it can also sequentially build the parser tree

[3] https://ub.com/ait-aecid/logdata-anomaly-miner/ [last accessed 7/21/2020]
[4] https://github.com/ait-aecid/logdata-anomaly-miner/ [last accessed 7/21/2020]

and adapt the parser according to changes in the system behavior. Hence, this would allow to automatically react to changes in log data syntax, for example, caused by new included network components and services, as well as software updates.

CHAPTER 7

Application cases

This chapter introduces three application cases for the methods described in previous chapters. All three applications have been developed in course of the work on this thesis and demonstrate how the technical results can be exploited. Besides leveraging the methods proposed in previous chapters, the applications have different levels of abstraction and make use of each other. Thus, the remainder of the chapter structures as follows (see Fig. 7.1):

1. *Time series analysis (TSA):* Section 7.1 introduces an advanced anomaly detection approach that processes log data and applies TSA to reveal malicious system behavior. The approach uses the incremental clustering proposed in Ch. 4, which bases on the ideas of the bio-clustering introduced in Ch. 3.

2. *Automatic Event Correlation for Incident Detection (AECID):* Section 7.2 describes the IDS AECID and its core component the AMiner. AECID applies log analysis for anomaly detection. Log based anomaly detection was the initial motivation for the thesis and served as primary use case for the evaluations of methods proposed in previous chapters. Hence, a large amount of research carried out in course of this thesis has been used to further develop AECID. For example, the parser generator described in Ch. 6 allows to automatically generate log parsers for AECID. Furthermore, the TSA approach described in Sec. 7.1 can be applied with AECID.

3. *Benchmarking and Analytic Evaluation of IDS in Specified Environments (BAESE):* Section 7.3 outlines a concept for generating realistic semi-synthetic test data for the evaluation of IDS that process log data. The BAESE approach applies the bioclustering (Ch. 3, the incremental clustering (Ch. 4), the character-based template generator (Ch. 5) and the parser generator (Ch. 6). Furthermore, it can

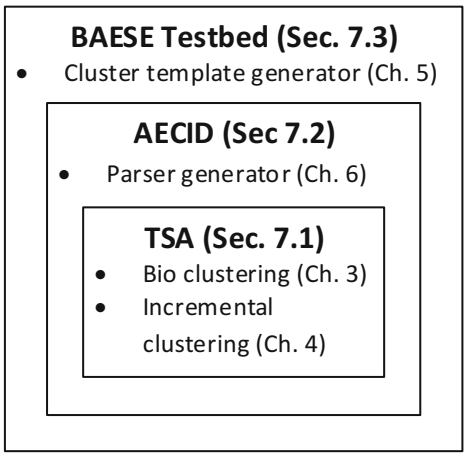


Figure 7.1: This figure depicts the structure of Ch. 7 and outlines how the approaches from other chapters contribute to the three application cases.

be leveraged to assess, compare, configure, and optimize solutions such as the TSA approach (Sec. 7.1) and AECID 7.2

This chapter heavily bases on [WSSF18], [SSK+18], [SSS+17], [LSW+19], [LWS+18a], [LWS+18b], [WSSS16], [WSSF15] and [WS16]. Since, the methods applied in these three application cases have been already evaluated in previous chapters, this chapter does not include detailed evaluation results. However, information regarding evaluation can be found in publications mentioned above.

7.1 Time series analysis: unsupervised anomaly detection beyond outlier detection

Major parts of the following section have been published in [LWS+18b], [LWS+18a] and [WSL+17][1].

[1]The research for this subsection has been conducted in close collaboration with my colleague Max Landauer, who wrote his master thesis about time series analysis in system log data and has been supervised and supported by me during this time. Thus, texts, concepts, models and results presented in this subsection have been created in close collaboration of Max Landauer and me. Furthermore, these contributions are considered mainly as evaluation results and demonstration of the applicability of the incremental clustering approach presented in Sec. 4, and thus are not considered as a core result of the present thesis. I thank Max Landauer for explicitly consenting me the rights to use all our texts and results created during our cooperative research and attesting that the text and results have been created in course of a close collaboration.

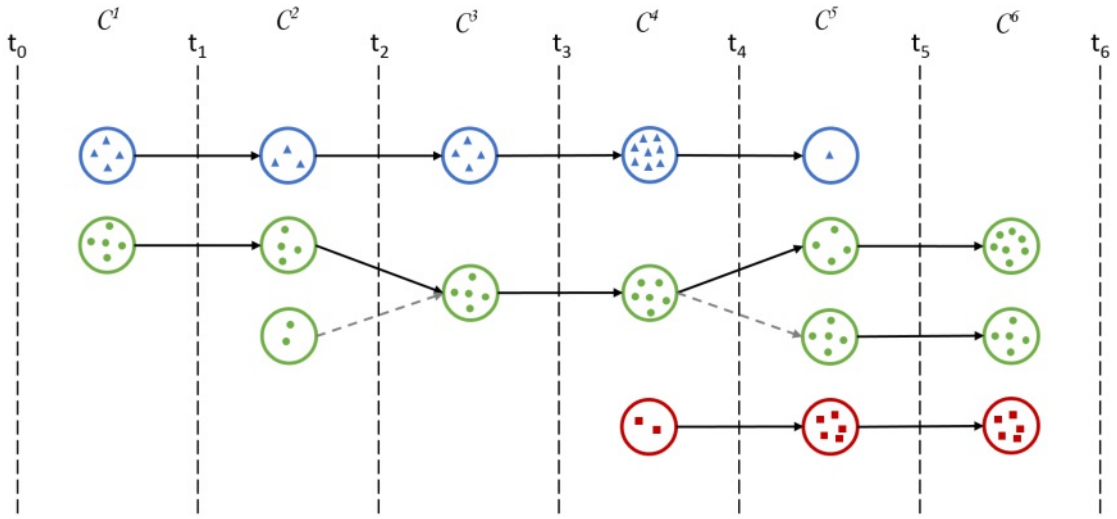


Figure 7.2: Example for dynamic cluster maps. Re-cording cluster maps dynamically over several time windows shows that clusters can appear, disappear, fusion and split over time (adapted from [LWS+18a]).

In contrast to signature and many rule-based IDS, unsupervised or semi-supervised clustering approaches operate independent from the structure of log data. Thus, approaches such as the bio-clustering (Ch. 3) and the incremental clustering (Ch. 4) are able to process any textual log data, to group similar log lines into a collection of clusters, i.e., a cluster map, and furthermore to detect anomalous log lines in form of outliers. However, cluster maps resulting from these algorithms usually only give a static view of the data. In general, locating outliers in these maps or single lines that contain significant words like "error" is not adequate for a thorough analysis of the system and neither is the presence or absence of certain lines sufficient to indicate problems, but rather the dynamic relationships and correlations between lines have to be considered [XHF+09].

Furthermore, static cluster maps cannot be used as permanent templates for a computer system. Due to the fact that any system generating log lines is continuously subject to changes, cluster maps generated during preceding time windows often turn out to consist of highly different structures. It is therefore necessary to incorporate dynamic features that span over multiple cluster maps. Figure 7.2 demonstrates the dynamic evolution of cluster maps over time. Cluster evolution analysis investigates transitions between clusters over time [HZHL16].

Existing cluster evolution techniques [SNTS06, CSZ+09, CKT06] rely on the principle that the same elements are observed and clustered over time. However, log lines are non-recurring objects, i.e., a log line occurs exactly at one single point in time and that same line is never observed again. Hence, it is not possible to simply match log lines with each other over time without previous efforts such as analyzing their similarity. Clustering groups similar log lines, but the structure and message content of lines within

clusters do not necessarily have to be homogeneous. Furthermore, log lines within clusters from different time windows may have structurally changed due to system events or modifications, for example, software updates that changed the syntax of log messages. While fuzzy string matching algorithms exist that alleviate these issues, their extensive computational complexity in combination with the immense amount of log lines distributed in numerous clusters, makes it nontrivial to determine transitions between clusters.

Finally, anomaly detection always relies on some kind of metric that determines whether a specific instance such as a log line, group of log lines or point in time is anomalous or not. Predefined limits are frequently used to trigger alarms for these metrics. However, they are not always an appropriate solution in an unsupervised setting. Reason for this is the fact that different systems usually show highly different behavior and also the behavior of a single system changes over time. A self-learning procedure should therefore be able to dynamically adjust to any environment it is placed into and adapt the limits for triggering alarms on its own.

Therefore, there is a need for dynamic log data anomaly detection that does not only retrieve lines that stand out due to their dissimilarity with other lines, but also identifies spurious line frequencies and alterations of long-term periodic behavior. We therefore introduce an anomaly detection approach containing the following novel features:

- A clustering model that is able to connect log line clusters from a sequence of static cluster maps and thereby supports the detection of transitions between these clusters,
- the definition and computation of metrics based on the temporal cluster developments and derived from aforementioned transitions between clusters,
- time series modeling and one-step ahead prediction for anomaly detection to detect contextual anomalies, i.e., outliers within their neighborhood.

7.1.1 Concept

This section uses an illustrative example to describe the concept of the anomaly detection approach that employs Cluster Evolution (CE) and time series analysis (TSA). For this, we consider log lines (see Fig. 7.3 (1)) that correspond to three types of events, marked with ◯, △ and □. The second layer (2) of Fig. 7.3 shows the occurrence of these lines on the continuous time scale that is split up by t_0, t_1, t_2, t_3 into three time windows. The third layer (3) of the figure visualizes the resulting sequence of cluster maps $\mathcal{C}, \mathcal{C}', \mathcal{C}''$ generated for each window. Note, in this example the clusters are marked for clarity. Due to the isolated generation of each map it is usually not possible to draw this connection and reason over the developments of clusters beyond one time window. The cluster transitions shown in the top (4) of the figure, including changes in position ($C_\triangle$ in $[t_1, t_2]$), spread ($C_\triangle$ in $[t_2, t_3]$), frequency ($C_\square$ in $[t_2, t_3]$) as well as splits ($C_\bigcirc$ in $[t_2, t_3]$), are thus overseen.

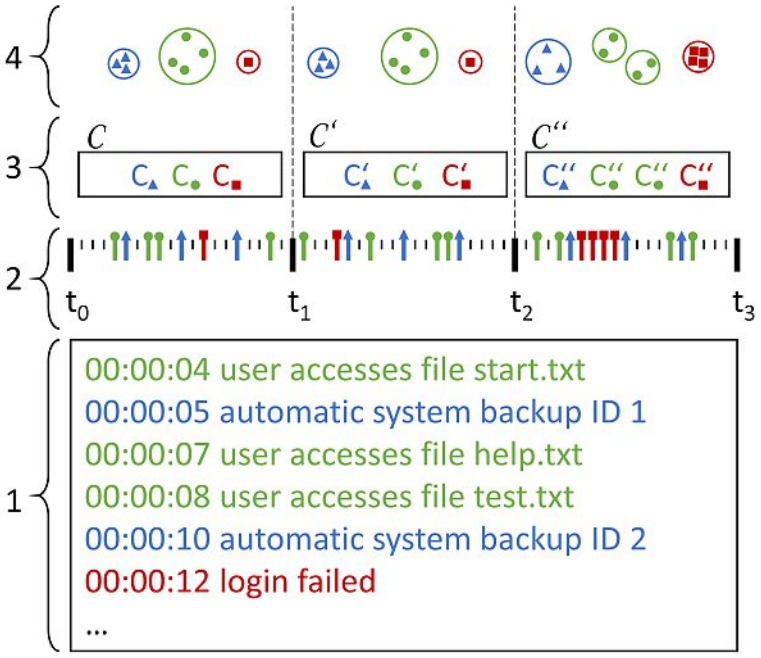


Figure 7.3: (1) Log file. (2) Log events occurring within time windows. (3) Static cluster maps for every time window. (4) Schematic clusters undergoing transitions (adapted from [LWS+18b]) [LWS+18a].

We therefore introduce an approach for dynamic log data analysis that involves CE and TSA in order to overcome these problems (Fig. 7.4). In step (1), the algorithm iteratively reads log lines either from a file or receives them as a stream. Our approach is able to handle any log format, however, preprocessing may be necessary depending on the log standard at hand. In our case, we use the preprocessing step (2) to remove any non-displayable special characters that do not comply to the standard syslog format defined in RFC5424 [Ger09]. Moreover, this step extracts the time stamps associated with each log line as they are not relevant for clustering. This is due to the fact that online handling of lines ensures that each line is processed almost instantaneously after it is generated.

Step (3) involves grouping log lines within each time window according to their similarity, resulting in a sequence of cluster maps. It is not trivial to determine how clusters from one map relate to the clusters from maps created during their preceding or succeeding time windows. Clustering the lines constituting each map into the neighboring maps (4) establishes this connection across multiple time windows and allows to determine transitions (5). A cluster from one time window evolves to another cluster from the following time window if they share a high fraction of common lines. More sophisticated case analysis is also able to differentiate advanced transitions such as splits or merges.

Several features of clusters are computed (6) and used for metrics that indicate anomalous

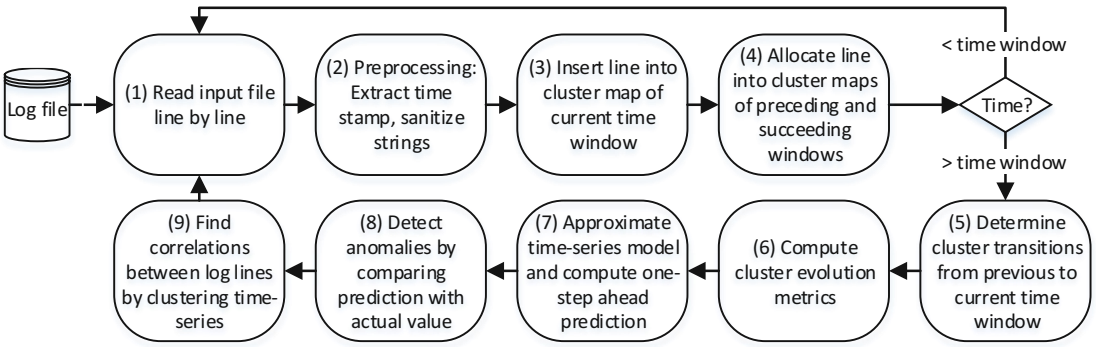


Figure 7.4: Flowchart of the dynamic clustering and anomaly detection procedure (adapted from [LWS+18a]) [LWS+18b].

behavior. As computations of these metrics follow the regular intervals of the time windows, we use TSA models (7) to approximate the development of the features over time. The models are then used to forecast a future value and a prediction interval lying one step ahead. If the actual recorded value occurring one time step later does not lie within these limits (8), an anomaly is detected. Figure 7.5 shows how the prediction limits (dashed lines) form 'tubes' around the measured, for example, cluster sizes. Anomalies appear in points where, for example, the actual cluster size lies outside that tube.

Finally, the time series of the cluster properties are also grouped according to their pairwise correlations. An incremental algorithm groups the time series similarly to the clustering of log lines. Carrying out this correlation analysis in regular intervals allows to determine whether two time series that used to correlate with each other over a long time suddenly stop doing that or whether new correlations between clusters appear, which are indicators of anomalous events (9).

7.1.2 Cluster evolution

This section describes in detail how online CE is performed on log lines. The approach is introduced stepwise, starting with a novel clustering model that establishes connections between cluster maps. Subsequently, we explain the process of tracking individual clusters and determining their transitions.

Clustering model

Considering only the lines of a single time window, we employ the incremental clustering approach introduced in Ch. 4. Repeatedly: The first line always generates a new cluster with itself as the cluster representative, a characteristic line for the cluster contents. For every other incoming line the most similar currently existing cluster is identified by comparing the Levenshtein distances between all cluster representatives and the line at hand. The processed line is then either allocated to the best fitting cluster or forms a

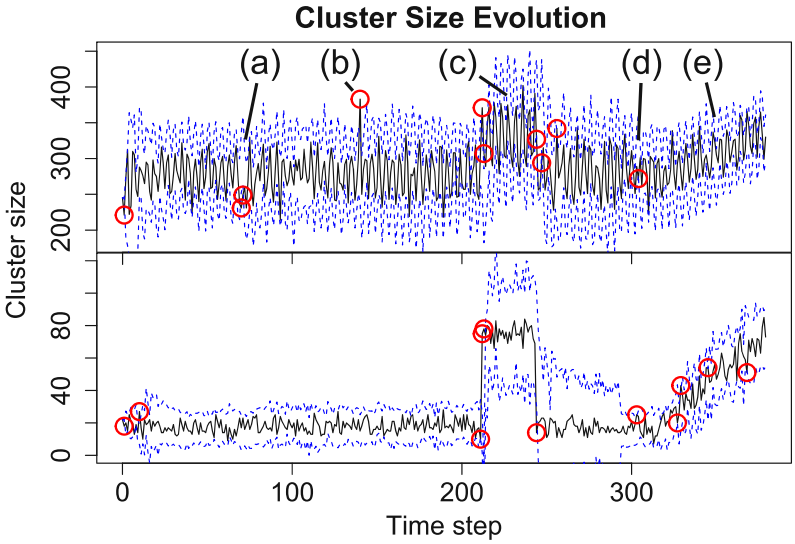


Figure 7.5: Time series representing the sizes of two evolving clusters (black solid lines) with prediction intervals (blue dashed lines) that form a 'tube' and detected anomalies (red circles). Top: A cluster affected by all anomalies caused by (a) incorrect periodicity, (b) sudden frequency increase, (c) long-term frequency increase, (e) slow frequency increase. (d) is a false positive. Bottom: A cluster not affected by periodic events, i.e. the size over time does not show oscillating patterns (adapted from [LWS+18a]) [LWS+18b].

new cluster with itself as the representative if the similarity does not exceed a predefined threshold t.

This clustering procedure is repeated for the log lines of every time window. The result is an ordered sequence of independent cluster maps $\mathcal{C}, \mathcal{C}', \mathcal{C}'', \ldots$. While the sequence itself represents a dynamic view of the data, every cluster map created in a single time window only shows static information about the lines that occurred within that window. The sequence of these static snapshots is a time series that only provides information about the development of the cluster maps as a whole, e.g., the total number of clusters in each map. However, no dynamic features of individual clusters can be derived.

It is not trivial to determine whether a cluster $C \in \mathcal{C}$ transformed into another cluster $C' \in \mathcal{C}'$ due to the fact that a set of log lines from a different time window was used to generate the resulting cluster. The reason is the nature of log lines that are only observed once at a specific point in time, while other applications employing CE may not face this problem as they are able to observe features of the same element over several consecutive time windows.

In order to overcome the problem of missing links between the cluster maps, we propose the following model: Every log line is not only clustered once to establish the cluster map

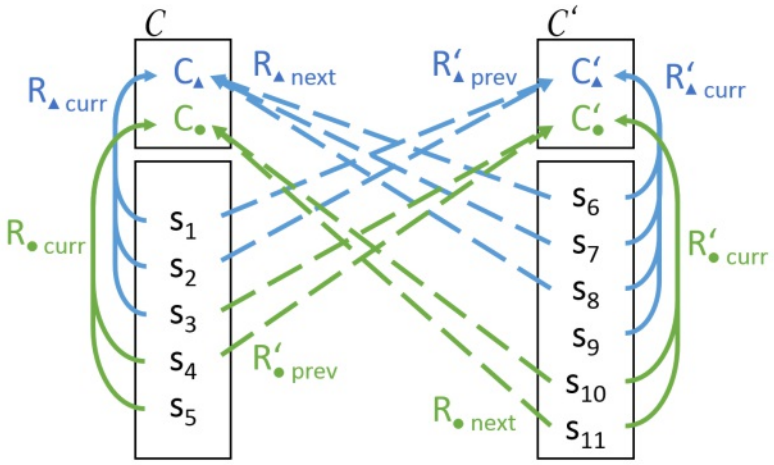


Figure 7.6: Solid lines: Construction of cluster map. Dashed lines: Log lines allocated to neighboring map [LWS+18b, LWS+18a].

in the time window in which it occurred, but is also allocated to the cluster maps created in the preceding and succeeding time windows. These two cases are called construction and allocation phase, respectively. The construction phase establishes the cluster map as previously described and each cluster stores the references to the lines it contains. The allocation phase allocates the lines to their most similar clusters from the neighboring cluster maps. This is also carried out using the incremental clustering algorithm, with the difference that no new clusters are generated and no existing clusters are changed, but only additional references to the allocated lines are stored. Note, lines do not necessarily have to be allocated.

Figure 7.6 shows the phases for two consecutive cluster maps. The solid lines represent the construction of the cluster maps $\mathcal{C}$ and $\mathcal{C}'$ by the log lines $s_1, ..., s_{11}$ that occurred in the respective time window, e.g., clusters $C_\triangle$ and $C_\bigcirc$ store references to the lines in $R_{\triangle curr}$ and $R_{\bigcirc curr}$ respectively, and $C'_\triangle$ and $C'_\bigcirc$ store their references in $R'_{\triangle curr}$ and $R'_{\bigcirc curr}$. The dashed lines represent the allocation of the lines into the neighboring cluster maps. Clusters in $\mathcal{C}$ store references to allocated log lines from the succeeding time window in $R_{\triangle next}$ and $R_{\bigcirc next}$. Analogously, clusters in $\mathcal{C}'$ store references to allocated log lines from the preceding time window in $R'_{\triangle prev}$ and $R'_{\bigcirc prev}$. Note that in the displayed example, s_3 was allocated to $C_\triangle$ in $\mathcal{C}$ but to $C_\bigcirc$ in $\mathcal{C}'$. Further, s_5 and s_9 are not allocated at all. The following section describes how this model is used for tracking individual clusters over multiple time windows.

Tracking

For any cluster $C \in \mathcal{C}$ and any other cluster $C' \in \mathcal{C}'$, a metric is required that measures whether it is likely that C transformed into C', i.e., whether both clusters contain logs from the same system process. An intuitive metric that describes the relatedness of C

and C' is their fraction of shared members. As previously mentioned, it is not possible to determine which members of each cluster are identical and it is therefore necessary to make use of the previously introduced clustering model that contains references to the neighboring lines. There exists an overlap metric based on the Jaccard coefficient for binary sets introduced in [GDC10] that was adapted for our model by formulating it as in Eq. (7.1).

$$overlap(C, C') = \frac{\left|\left(R_{curr} \cap R'_{prev}\right) \cup \left(R_{next} \cap R'_{curr}\right)\right|}{\left|R'_{curr} \cup R'_{prev} \cup R_{next} \cup R_{curr}\right|} \tag{7.1}$$

Note, the sets of references R_{curr} and R'_{prev} both correspond to log lines that were used to create cluster map $\mathcal{C}$ and can thus be reasonably intersected, while R_{next} and R'_{curr} both reference log lines from cluster map $\mathcal{C}'$. The overlap lies in the interval $[0, 1]$, where 1 indicates a perfect match, i.e., all log lines from one cluster were allocated into the corresponding other cluster, and 0 indicates a total mismatch.

Clusters can also be tracked over multiple time windows by applying the same idea to C' and C'', C'' and C''', and so on. In a simplistic setting where clusters remain very stable over time, this is sufficient for tracking all log line clusters separately. However, in realistic scenarios with changing environments clusters frequently undergo transitions such as splits or merges which negatively influence the overlap and may indicate anomalies. Therefore, in the following the tracking of clusters is extended with a mechanism for handling transitions.

Transitions

Clusters are subject to change over time. There exist internal transitions that only influence individual clusters within single time windows, and external transitions that affect other clusters as well [SNTS06]. We consider the cluster size denoted by $|C|$ as the most important internal feature as it directly corresponds to the frequency of log lines allocated to cluster C. Formally, a cluster C grows in size from one time step to another if $|C'| > |C|$, shrinks if $|C'| < |C|$ and remains of constant size otherwise. Alternative internal features derived from the distribution of the cluster members are their compactness measured by the standard deviation, their relative position as well as their asymmetry, i.e., their skewness.

Clusters from different time windows are affected by external transitions. In the following, θ is a minimum threshold for the overlap defined in Equation (7.1) and θ_{part} is a minimum threshold for partial overlaps that is relevant for splits and merges. In general, partially overlapping clusters yield smaller overlap scores, thus $\theta_{part} < \theta$. We take the following external transitions into account:

1. Survival: A cluster C survives and transforms into C' if $overlap(C, C') > \theta$ and there exists no other cluster $B \in \mathcal{C}$ or $B' \in \mathcal{C}'$ so that $overlap(B, C') > \theta_{part}$ or $overlap(C, B') > \theta_{part}$.

2. Split: A cluster C splits into the parts $C'_1, C'_2, ..., C'_p$ if all individual parts share a minimum amount of similarity with the original cluster, i.e., $overlap(C, C'_i) > \theta_{part}$, for all $i \in \{1, 2, \ldots, p\}$, and the union of all parts matches the original cluster, i.e., $overlap(C, \bigcup C'_i) > \theta$. There must not exist any other cluster that yields an overlap larger than θ_{part} with any of the clusters involved.

3. Absorption: The group of clusters $C_1, C_2, ..., C_p$ merge into a larger cluster C' if all individual parts share a minimum amount of similarity with the resulting cluster, i.e., $overlap(C_i, C') > \theta_{part}$, $i \in \{1, 2, \ldots, p\}$, and the union of all parts matches the resulting cluster, i.e., $overlap(\bigcup C_i, C') > \theta$. Again, there must not exist any other cluster that yields an overlap larger than θ_{part} with any of the clusters involved.

4. Disappearance or Emergence: A cluster C disappears or a cluster C' emerges if none of the above cases holds true.

By this reasoning it is not possible that a connection between two clusters is established if their overlap does not exceed θ_{part}, which prevents partial clusters that do not exceed this threshold from contributing to the aggregated cluster in the case of a split or merge. In order to track single clusters it is often necessary to follow a specific 'path' when a split or merge occurs. We suggest to pick paths to clusters based on the highest achieved overlap, largest cluster size, longest time that the cluster exists or combinations of these.

Evolution metrics

Knowing all the interdependencies and evolutionary relationships between the clusters from at least two consecutive time windows, it is possible to derive in-depth information about individual clusters and the interactions between clusters. Definite features such as the cluster size that directly corresponds to the frequency of the log lines within a time window are relevant metrics for anomaly detection, however do not necessarily indicate anomalies regarding changes of cluster members.

A more in-depth anomaly detection therefore requires the computation of additional metrics that also take the effects of cluster transitions into account. Toyoda and Kitsuregawa [TK03] applied several inter-cluster metrics in CE analysis that were adapted for our purposes. For example, we compute the stability of a cluster by $s = \left|R'_{prev}\right| + |R_{curr}| - 2 \cdot \left|R'_{prev} \cap R_{curr}\right|$, where low scores indicate small changes of the cluster and vice versa. For a better comparison with other clusters, a relative version of the metric is computed by dividing the result by $\left|R'_{prev}\right| + |R_{curr}|$. There exist numerous other metrics where each take specific types of migrations of cluster members into account, such as growth rate, change rate, novelty rate, or split rate [LWS+18a].

A simple anomaly detection tool could use any of the desired metrics, compare them with some predefined thresholds and raising alarms if one or more of them exceed these thresholds. Even more effectively, these metrics conveniently form time series and can thus be analyzed with TSA methods.

7.1.3 Time series analysis

The time series derived from metrics such as the cluster size are the foundation for analytical anomaly detection. This section describes the application of TSA methods to model the cluster developments and perform anomaly detection by predicting future values of the time series.

Model

Time series are sequences of values $y_0, y_1, y_2, \ldots$ associated with specific points in time $t = 0, 1, 2, \ldots$. For our purposes, a time step therefore describes the status of the internal and external transitions and their corresponding metrics of each cluster at the end of a time window. These sequences are modeled using appropriate methods such as autoregressive integrated moving-average (ARIMA) processes. ARIMA is a well-reasearched modeling technique for TSA that is able to include the effects of trends and seasonal behavior in its approximations [CC08].

Clearly, the length of the time series is ever increasing due to the constant stream of log messages and at one point its handling will become problematic either by lack of memory or by the fact that fitting an ARIMA model requires too much runtime. As a solution, only a certain amount of the most recent values are stored and used for the model as older values are of less relevance. The specific number of considered values depends on the available amount of resources and can be defined by the user.

Forecast

With appropriate estimations for the parameters, an extrapolation of the model into the future allows the computation of a forecast for the value directly following the last known value. By applying this procedure recursively it is possible to predict for arbitrary horizons into the future. In our approach an ARIMA model is fitted in every time step and we are interested only in predictions one time step ahead rather than long-term forecasts.

The smoothness of the path that a time series follows can be highly different. Therefore, neither a threshold for the absolute nor the relative deviation between a prediction and the actual value is an appropriate choice for anomaly detection. Assuming independent and normally distributed errors, the measured variance of previous values is therefore used to generate a prediction interval which contains the future value with a given probability. Using the ARIMA estimate $\hat{y}_t$, this interval is computed by Eq. (7.2), where $\mathcal{Z}_{1-\frac{\alpha}{2}}$ is the

quantile $1 - \frac{\alpha}{2}$ of the standard normal distribution and s_e is the standard deviation of the error, $s_e = \sqrt{\frac{1}{n-1}\sum(y_t - \bar{y}_t)^2}$.

$$I_t = \left[\hat{y}_t - \mathcal{Z}_{1-\frac{\alpha}{2}} s_e, \hat{y}_t + \mathcal{Z}_{1-\frac{\alpha}{2}} s_e\right] \tag{7.2}$$

Correlation

Some types of log lines appear with almost identical frequencies during certain intervals, either because processes that generate them are linked in a technical way so that a log line always has to be followed by another line, or processes just happen to overlap in their periodical cycles. Either way, time series of these clusters follow a similar pattern and they are expected to continue this consistent behavior in the future. The relationship between two time series y_t, z_t is expressed by the cross-correlation function [CC08], which can be estimated for any lag k as shown in Eq. (7.3), where $\bar{y}$ and $\bar{z}$ are the arithmetic means of y_t and z_t, respectively. Using the correlation as a measure of similarity allows to group related time series together.

$$CCF_k = \begin{cases} \frac{\sum_{t=k+1}^{N}(y_t-\bar{y})(z_{t-k}-\bar{z})}{\sqrt{\sum_{t=1}^{N}(y_t-\bar{y})^2}\sqrt{\sum_{t=1}^{N}(z_t-\bar{z})^2}} & \text{if } k \geq 0 \\ \frac{\sum_{t=1}^{N+k}(y_t-\bar{y})(z_{t-k}-\bar{z})}{\sqrt{\sum_{t=1}^{N}(y_t-\bar{y})^2}\sqrt{\sum_{t=1}^{N}(z_t-\bar{z})^2}} & \text{if } k < 0 \end{cases} \tag{7.3}$$

Detection

For every evolving cluster, the anomaly detection algorithm checks whether the actual retrieved value lies within the boundaries of the forecasted prediction limits calculated according to Eq. (7.2). An anomaly is detected if the actual value falls outside of that prediction interval, i.e., $y_t \notin I_t$. Figure 7.5 shows the iteratively constructed prediction intervals forming 'tubes' around the time series. The large numbers of clusters, time steps and the statistical chance of random fluctuations causing false alarms often make it difficult to pay attention to all detected anomalies. We therefore suggest to combine the anomalies identified for each cluster development into a single score. At first, we mirror anomalous points that lie below the tube to the upper side applying Eq. (7.4).

$$s_t = \begin{cases} y_t & \text{if } y_t > \hat{y}_t + \mathcal{Z}_{1-\frac{\alpha}{2}} s_e \\ 2\hat{y}_t - y_t & \text{if } y_t < \hat{y}_t - \mathcal{Z}_{1-\frac{\alpha}{2}} s_e \end{cases} \tag{7.4}$$

With the time period τ_t describing the number of time steps a cluster is already existing we define $\mathcal{C}_{A,t}$ as the set of clusters that contain anomalies at time step t and exist for at

least 2 time steps, i.e., $\tau_t \geq 2$. We then define the anomaly score a_t for every time step as in Eq. (7.5).

$$a_t = 1 - \frac{\sum_{C_t \in \mathcal{C}_{A,t}} \left(\left(\hat{y}_t + \mathcal{Z}_{1-\frac{\alpha}{2}} s_e \right) \log\left(\tau_t\right) \right)}{|\mathcal{C}_{A,t}| \sum_{C_t \in \mathcal{C}_{A,t}} \left(s_t \log\left(\tau_t\right) \right)} \tag{7.5}$$

When there is no anomaly occurring in any cluster at a specific time step, the anomaly score is set to 0. The upper prediction limit in the numerator and the actual value in the denominator ensure that $a_t \in [0, 1]$, with 0 meaning that no anomaly occurred and scores close to 1 indicating a high significance for an anomaly. Dividing by $|\mathcal{C}_{A,t}|$ and incorporating the cluster existence time τ_t ensures that anomalies detected in multiple clusters and clusters that have been existing for a longer time are weighted higher in the anomaly scores. The logarithm is used to damp the influence of clusters with comparatively large τ_t.

Finally, we detect anomalies based on changes in correlations. Clusters which correlate with each other over a long time during normal system operation should continue to do so in the future. In case that some of these clusters permanently stop correlating, an incident causing this change must have occurred and should thus be reported as an anomaly. The same reasoning can be applied to clusters which did not share any relationship but suddenly start correlating. Therefore, after the correlation analysis has been carried out sufficiently many times to ensure stable sets of correlating clusters, such anomalies are detected by comparing which members joined and left these sets.

7.1.4 Illustrative application scenario

This section describes an illustrative attack scenario to introduce the detection capabilities of the CE and TSA based anomaly detection approach.

Attack scenario

In order to identify many clusters, we pursue high log data diversity. For this, we propose the following illustrative scenario that adapts an approach introduced in [SSFF14]: A MANTIS Bug Tracker System[2] is deployed on an Apache Web server. Several users frequently perform normal actions on the hosted website, e.g., reporting and editing bugs. At some point, an unauthorized person gains access to the system with user credentials stolen in a social engineering attack. The person then continues to browse on the website, however following a different scheme, e.g., searching more frequently for open issues which simulates suspicious espionage activities. Such actions do not cohere with the behavior of the other users and we therefore expect to observe corresponding alterations in the developments of the log clusters. Due to the fact that only the probabilities for

[2] https://www.mantisbt.org/

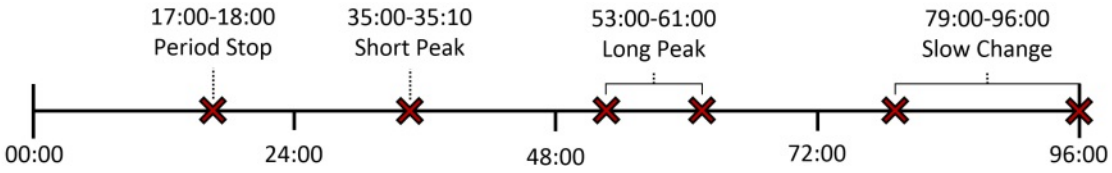


Figure 7.7: Injected anomalies on a timeline [LWS+18a].

clicking on certain buttons are changed, we expect that the log lines produced by the attacker will be clustered together with the log lines describing normal behavior and that this causes an increase in the measured cluster size. In addition, an automatized program that checks for updates in regular intervals is compromised by the attacker and changes its periodic behavior. In this case, we expect that the changes of the periodic cycles are also reported as anomalies. Figure 7.7 shows the six attacks on a timeline. The injected attacks include one missing periodic pulse, two sudden increases of cluster size with different length and one slowly increasing cluster size.

Results

Figure 7.5 shows the cluster size developments of two log line clusters, the one-step ahead prediction limits forming tubes around the curves and the anomalies that are detected whenever the actual size falls outside of this tube. The present types of anomalies in the plot are: (a) a periodic process skipping one of its peaks, (b) a spike formed by a rapid short-term increase in line frequency, (c) a plateau formed by a long-term frequency increase, (d) a false positive and (e) a slowly increasing trend. The curve in the top part of the figure corresponds to a cluster affected by all injected anomalies. While anomalies (a)-(c) are appropriately detected, anomaly (e) is not detected in this cluster, because the model adapts to the slow increase of frequency that occurs within the prediction boundaries thereby learning the anomalous behavior without triggering an alarm. We intentionally injected (e) in order to show these problems that occur with most self-learning models. These issues can be solved by employing change point analysis methods that detect long-term changes in trends [KFE12]. The bottom part of the figure corresponds to a cluster containing only log lines that are specifically affected by anomalies (c) and (e). Accordingly, the anomalies manifest themselves more clearly and the high deviations from the normal behavior makes their detection easier. The fact that each of the numerous evolving clusters are specific to certain log line types is a major advantage of our method. In particular, more than 300 evolving clusters representing more than 90% of the total amount of log lines were identified.

The anomaly score aggregated over all evolving clusters that exist for at least 20 time steps, where each time step is 15 minutes long, is displayed in Fig. 7.8. The figure clearly shows that the anomaly score increases at the beginning and end of every attack interval. This corresponds to the fact that our algorithm detects changes of system behavior, but almost immediately adapts to the new state. Only returning from this anomalous state

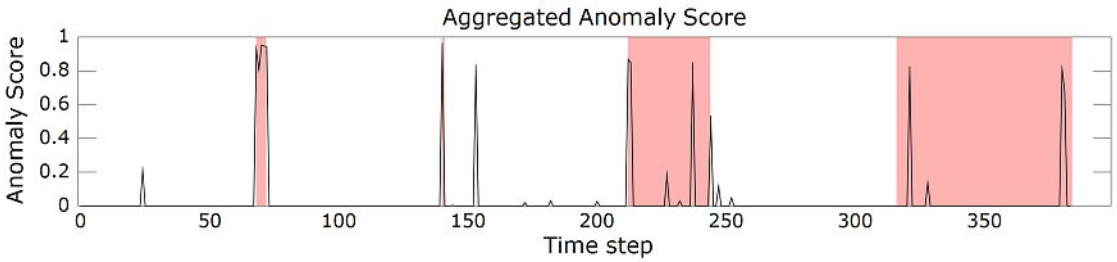


Figure 7.8: The aggregated anomaly score displayed as a time series and correctly increasing when the system behavior changes (red shaded intervals) [LWS+18a, LWS+18b].

to the normal behavior is again detected as an anomaly.

7.2 AECID: A self-learning anomaly detection approach based on light-weight log parser models

Major parts of the remaining section have been published in [WSSF18]. Research on IDS seems – due to rapidly changing technologies and system design paradigms – to be a never-ending story. Signature-based approaches, i.e., black-listing methods, are still the de-facto standard applied today for some good reasons: they are essentially easy to configure, can be centrally managed, i.e., do not need much customization for specific networks, yield a robust and reliable detection for known attacks and provide low false positive rates. Nevertheless, there are, solid arguments to watch out for more sophisticated anomaly-based detection mechanisms, which should be applied additionally to black-listing approaches for the reasons explained as follows:

- The exploitation of new zero-day vulnerabilities are hardly detectable by black-listing-approaches. Simply, there are no signatures to describe the indicators of an unknown exploit.
- Attackers can easily circumvent the detection of malware, once indicators are widely distributed. Simply re-compiling a malware with small modifications will change hash sums, names, IP addresses of command and control servers and the like – in the worst case, rendering all these data which is used to describe indicators useless.
- Eventually, many sophisticated attacks use social engineering as an initial intrusion vector. Here no technical vulnerabilities are exploited, hence, no indicators on a blacklist can appropriately describe malicious behavior.

Especially the latter requires smart anomaly detection approaches to reliably discover deviations from a desired system's behavior as a consequence of an unusual utilization through an illegitimate user. This is the usual case when an adversary manages to steal user credentials and is using these actually legitimate credentials to illegitimately

access a system. However, an attacker will eventually utilize the system differently from the legitimate user, to reach his target, for instance running scans, searching shared directories and trying to extend his influence to surrounding systems at either unusual speed, at unusual times, taking unusual routes in the network, issuing actions with unusual frequency, causing unusual data transfers at unusual bandwidth. This causes a series of events within an infrastructure which are picked up by anomaly-based approaches and used to trigger off alerts.

In this section, we have a close look on AECID[3] (Automatic Event Correlation for Incident Detection) that applies anomaly detection for intrusion detection. AECID specifically monitors semantically rich and verbose log data and applies. In particular the contributions of this section are:

- We discuss the design principles of a modern scalable anomaly detection system to be applied in large-scale distributed systems.
- We outline the AECID approach, which is an actual implementation based on the aforementioned design principles.
- We describe a successful proof of concept of the AECID approach in a network of cyber physical systems (CPS).

7.2.1 The AECID approach

In this section we illustrate the system architecture and design of AECID and describe its two main components the *AMiner*[4] and *AECID Central*.

Figure 7.9 depicts the system architecture of AECID. AECID is designed to allow the deployment in highly distributed environments; in fact, due to its lightweight implementation, an AMiner instance can be installed, as sensor, on any relevant node of a network; AECID Central is the component responsible of controlling and coordinating all the deployed AMiner instances.

AMiner

The AMiner operates similarly to an HIDS sensor. It runs on every host and network node that is monitored, or on a centralized logging storage which collects the log data generated by the monitored nodes. Each AMiner instance interprets the log messages acquired from the node it is deployed on, following a specific model, called *parser model*, generated ad-hoc to represent the different events being logged on that particular node. For this purpose, the AMiner applies the highly efficient tree-based parser introduced in Sec. 6.1, which allows to parse log lines with $O(\log(n))$. Furthermore, a tailored rule set recognizes the events that are considered legitimate on that system; an AMiner instance

[3] https://aecid.ait.ac.at/ [last accessed 12/12/2019]
[4] https://github.com/ait-aecid/logdata-anomaly-miner [last accessed: 07/02/2020]

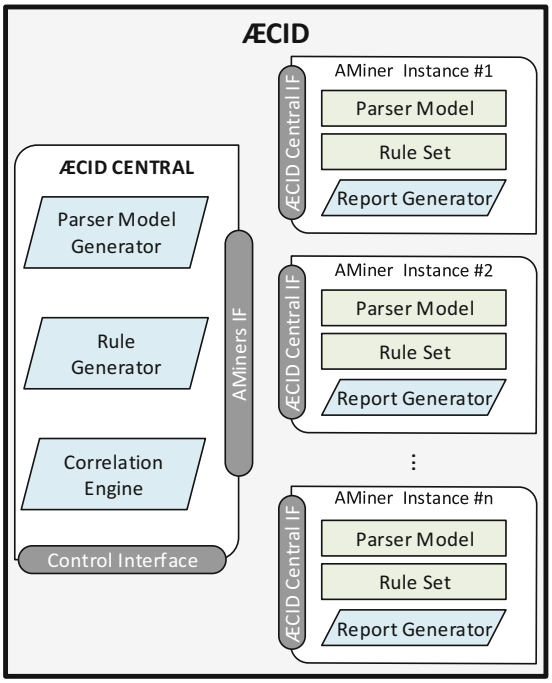


Figure 7.9: AECID architecture [WSSF18].

checks every parsed log line against this rule set and reports any mismatch. Additionally, each AMiner instance comprises a report generator that produces a detailed record of parsed and unparsed lines, alerts and triggered alarms. The reports can be sent either via the AECID Central Interface to the AECID Central, or through additional interfaces (e.g., via e-mail or message queue) to system administrators or to a security information and event management (SIEM) tool. The parser model in combination with the rule set, characterize the normal system behavior, i.e., describe the type, structure, and content of log lines representing events allowed to occur on the monitored system. Every log message violating this behavioral model represents an anomaly.

AECID central

While the AMiner performs lightweight operations such as parsing log messages and comparing them against a set of existing rules, AECID Central provides more advanced features, and therefore requires more computational resources than a single AMiner instance. One of the main functions executed by AECID Central is to learn the normal system behavior of every monitored system, and consequently configure the AMiner instance, running on that system, to detect any logged abnormal activity. To do this, AECID Central analyzes the logs received from each AMiner instance, generates a tailored parser model (*Parser Model Generator*[5] function) and a specific set of rules

[5]https://github.com/ait-aecid/aecid-parsergenerator [last accessed: 07/02/2020]

(*Rule Generator* function), and sends them to the AMiner instance, which adopts them to examine future log messages. For generating log parsers, AECID applies the approach introduced in Ch. 6. AECID Central provides the different AMiner instances with self-learned parser models and rule sets, and adapts them, when the network infrastructure and/or the user behavior change. Hence, AECID Central needs to control and configure all the deployed AMiner instances; these operations are performed through the *AMiner Interface*. Moreover, a *Control Interface* allows a system administrator to communicate with AECID Central, adjust its settings, and configure the deployed AMiner instances. Additionally, AECID Central leverages a *Correlation Engine* that allows to analyze and associate events observed by different AMiner instances, with the purpose of white-listing events generated by complex processes involving diverse network nodes. If the log data collected within a network infrastructure includes records of communication events between network devices (e.g., headers observed via *tcpdump*), AECID can operate as NIDS, and therefore be utilized as hybrid IDS.

Detecting anomalies

The AMiner interprets every incoming log message according to a specific parser model, which characterizes the events observed on the device or network component it monitors. The parser model represents a path entropy model that efficiently describes the whitelisted, i.e. permitted, log lines. It describes the log model of the monitored system as a graph, specifically an ordered tree (see Figure 6.1). The goal of using the parser model is to filter out as much redundant information as possible before a detailed analysis of the log line is performed. The parser model allows to efficiently extract all the information contained in a log line, while retaining only a minimum amount of data. Thus, every branch of the graph includes fixed segments, which represent constant strings that always occur at the same position of the log line, and variable segments, which represent strings that differ from line to line.

There are two ways for the AMiner to reveal anomalies within the log messages: (i) by observing deviations of the log lines from the parser model, (ii) by identifying log lines which do not follow certain predefined rules.

Thanks to the acquired knowledge on the normal system behavior, formalized through the corresponding parser model, the most advantageous way to reveal anomalies is to detect significant deviations of the logged events from the normal system behavior model. Usually, an information system operates only in a limited number of system states. The events in these states, which occur while the system runs normally, i.e. when the system is not in maintenance, error or recovery mode, define the model for the normal system behavior. Given these log data records that reflect the normal state, a log message that does not match any available path in the parser model graph is to be considered anomalous, because it represents an unexpected system event. Thus, the AMiner considers a log line non-anomalous only if it matches one entire path of the parser model graph. Every deviation from the graph's paths indicates an anomalous event. The anomaly can be caused either by a technical failure, by maintenance activities, or by an

unused system function that has been activated by an attack. The AMiner whitelists all paths described by the graph defined by the parser model, and raises an alert every time a log line cannot be fully parsed.

Another way to detect anomalies is by defining white-listing rules, which are configured to allow only specific system event types and/or associated parameters. The AMiner extracts all paths occurring in a log line and the associated parsed values as shown in Fig. 6.4. White-listing rules allow only specific values for a certain log line element, or specific combinations of value pairs. For example, a whitelist may only allow a certain list of IP addresses in `/model/services/ntpd/msg/ipv4/ip`. If a non-whitelisted IP address is detected, an alert or an alarm is raised, and consequently an e-mail message automatically sent to system administrators for notification. The same concept can be applied for a combination of values; for example to allow that specific user names only appear together with certain IP or MAC addresses. Furthermore, it is also possible to learn the probability distribution with which values of a certain path should occur and raise an alarm if the distribution changes. Finally, rules may also permit a range or a list of values.

Signature based IDS normally analyze log lines individually, however, malicious network behavior often manifests in a sequence of multiple log lines. Only by correlating such a sequence of events the anomaly becomes apparent. For this reason, the AMiner detects anomalies based on statistics. Two examples for anomalies detectable with such statistical methods are: First, a specific event which normally occurs 5 to 7 times per hour, will trigger an alert if it suddenly occurs 10 times in one hour. In this case, the AMiner will detect changes in the distribution of path occurrences. Second, assuming that the occurrence of a path follows a normal distribution over a predefined time interval, a fluctuation of the mean and of the standard deviation will indicate an anomaly. This demonstrates that the AMiner is able to detect not only anomalous single events, but also anomalous event frequencies.

Furthermore, the AMiner features TSA approach proposed in Sec. 7.1. Similarly, to the detectors based on statistics, the TSA allows to reveal anomalies that relate to malicious behavior, which generates log lines that look like normal behavior, but, for example, occur with an anomalous frequency.

Rule generator

Another self-learning feature of AECID is the rule generation. Similarly to the parser model generator, the rule generator can be activated for a specific AMiner instance via the control interface. Once enabled, the selected AMiner instance will forward the parsed lines to the rule generator running on AECID Central. Based on the parsed values the rule generator creates candidates for rules. It defines lists or intervals of values that are allowed to occur in a specific path, or it defines rules enforcing that values of different paths are only allowed to occur in specific combinations, e.g., specific usernames are only allowed to occur in combination with specific IP or MAC addresses. The rule generator

proposes also rules that analyze the probability distribution with which certain values occur.

Once the rule generator defines a rule candidate, the candidate has to be verified. One way to accomplish this is to run a binomial test as shown in [FSSF15], to evaluate if a tested rule candidate is stable or not. Once a rule candidate is verified and therefore considered stable, the rule generator pushes it to the AMiner, which includes it into its rule set.

Correlation engine

The correlation engine implemented in AECID Central allows to detect network-wide anomalies by correlating events observed by different AMiner instances. This makes it possible to detect deviations within complex processes that involve different services, and therefore produce log messages on components monitored by different AMiner instances. Consider the example depicted in Fig. 7.10; it shows the normal access chain of the log-in procedure to, for example, a web shop. If a user logs into a web shop, certain log lines will be produced in a specific order by the firewall, the web server, the database server and the web server again, including specific values for the paths of the parser. AECID is able to recognize such event sequences, and to generate corresponding models. This allows AECID to automatically derive relevant correlation rules and verify through them if the system behavior is aligned with the generated model. All AMiner instances monitoring the services involved in the process will, in fact, forward to AECID Central the log events for which the correlation rule is being evaluated (even if the single events are not individually considered anomalous). AECID Central will then analyze the sequence of events collected from the different AMiner instances and verify their alignment to the model. If the illustrated event chain mentioned above is violated, because the database is accessed without a previous access to the web server or the firewall is being bypassed, AECID Central will recognize an inconsistency and therefore trigger an alarm. This is an example that demonstrates how AECID does not only detect anomalies that manifest in deviations from the normal system behavior of a single device, but also complex anomalies that can only be detected when analyzing events occurring in distributed nodes of the network. It is important to notice that the correlation rules can also be applied to a single AMiner instance to analyze complex processes running on one single node (or when operating on a centralized log store which contains events collected from multiple nodes).

Detectable anomalies

An effective anomaly detection method recognizes different types of anomalies with certain levels of confidence. In the following, we list the main categories of anomalies which AECID reveals.

The simplest type of anomaly is represented by *anomalous single events*. On the one hand, these can be so-called *outliers* representing rarely occurring events, which appear so seldom that they are not part of the normal system behavior model. On the other hand,

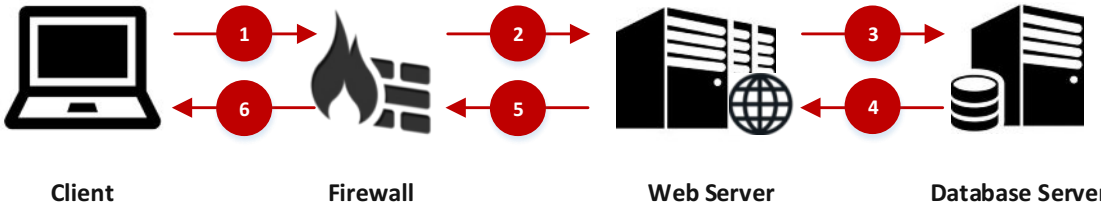


Figure 7.10: The figure shows the log-in process to, for example, a web shop: (1) the client tries to log into the web shop on a web server, (2) a connection through the firewall occurs, (3) the web server checks credentials through a database query, (4) the database query returns some result, (5) a response through the firewall: access acceptance or denial, (6) client receives the response [WSSF18, FWABK17].

these anomalies can be violations of permitted parameter values or value combinations; for example, a server access through an unknown (not whitelisted) user agent. In case of black-listing approaches, user agents that are not allowed need to be added (one-by-one) to the blacklist, and hence imply a high risk of incompleteness.

Anomalous event parameters are point anomalies such as IP addresses, port numbers or software versions that are not whitelisted and therefore are not part of the normal system behavior. This type of anomalies includes, for example, events that occur outside of business hours, or are triggered by accounts of employees who are on vacation.

Anomalous single event frequencies are events usually considered normal, which occur with an anomalous frequency. For example, in case of data theft, an anomalously high number of database accesses from a single client would be recorded in the log data, triggering an anomaly.

Anomalous event sequences are anomalies discovered by observing the dependency between related events. Such dependency can be formalized by defining correlation rules. A correlation rule describes a series of events that have to occur in an ordered sequence, within a given time window, to be considered non anomalous. To detect more complex anomalous processes, which may involve different systems on a network, multiple log lines need to be examined. After a particular log line type (recording a conditioning event) is observed, another specific log line (recording the expected implied event) has to occur within a predefined time slot, otherwise, an alert is raised. Additionally, such correlation rules should be definable so that once a given (conditioning) event occurred, the algorithm checks if in a predefined time window previous to such event, another specific (implied) event has occurred.

7.2.2 System deployment and operation

This section illustrates the different deployment topologies of AECID and presents the phases of its operational workflow.

The simplest way to deploy AECID is by employing only one AMiner instance installed

on a single node. This setup allows to exclusively monitor events produced and logged by that single component. Alternatively, if a log collection mechanism is in place on the node, which acquires log messages from other remote nodes in the network (e.g., via a syslog server), the AMiner can work in a centralized fashion, and analyze events occurring on distributed systems. The drawback of this configuration is that the parser models, as well as the set of rules utilized by the AMiner for the detection of anomalies, are statically defined and need to be manually configured. The lack of an intelligent component (AECID central) implies, in fact, that the administrators have to: (i) define the parser model describing the structure of the events being logged by the monitored node, and (ii) have a thorough understanding of every event (occurring on every monitored node) that has to be considered legitimate, and instantiate a corresponding set of white-listing rules. This solution is applicable in case of small-scale systems, whose computational power is not sufficient to run AECID central, which perform highly recurrent operations, and are therefore simple to characterize manually.

If the infrastructure to be monitored comprises a large number of distributed nodes, with little resources and small computational power at their disposal, AECID can be deployed following a star topology. In this setup, an AMiner instance is installed on every distributed node, while a more powerful node hosts AECID Central. Every AMiner is connected to AECID Central and exchanges information regarding parsed lines and discovered anomalies with it.

Figure 7.11 illustrates the three stages required to initialize the AECID system when deployed in this topology. When the AMiner instances are installed for the first time on the nodes, they are not able to parse any of the log lines generated on the node, because no parser model is defined yet; thus, they forward every unparsed line to AECID Central, which learns the structure of the log messages received from each node and automatically builds a dedicated parser model for each service generating logs on each node. Along with the parser models, AECID Central builds the behavioral model of the services running on every connected node, and generates a corresponding set of white-listing rules per node, which describe the model, i.e., the normal behavior. The time it takes AECID Central to build a stable model of the normal behavior depends on the complexity of the data provided by the logging mechanism. Furthermore, AECID is capable of adapting the model during runtime.

In a second step, AECID Central forwards the derived parser models and rule-sets to the respective AMiner instances; now the AMiner instances can follow the received parser models, analyze the incoming log messages and identify any suspicious event by checking the adherence to the obtained rules.

The third step represents the fully operational system; the AMiner instances continue sending any unparsed log line to AECID Central for further inspection, and receive from AECID Central updates on the enabled parser models and set of rules. If correlation rules are defined, AECID Central (through its correlation engine) analyzes the relevant log messages from the involved AMiner instances, as described in the previous section.

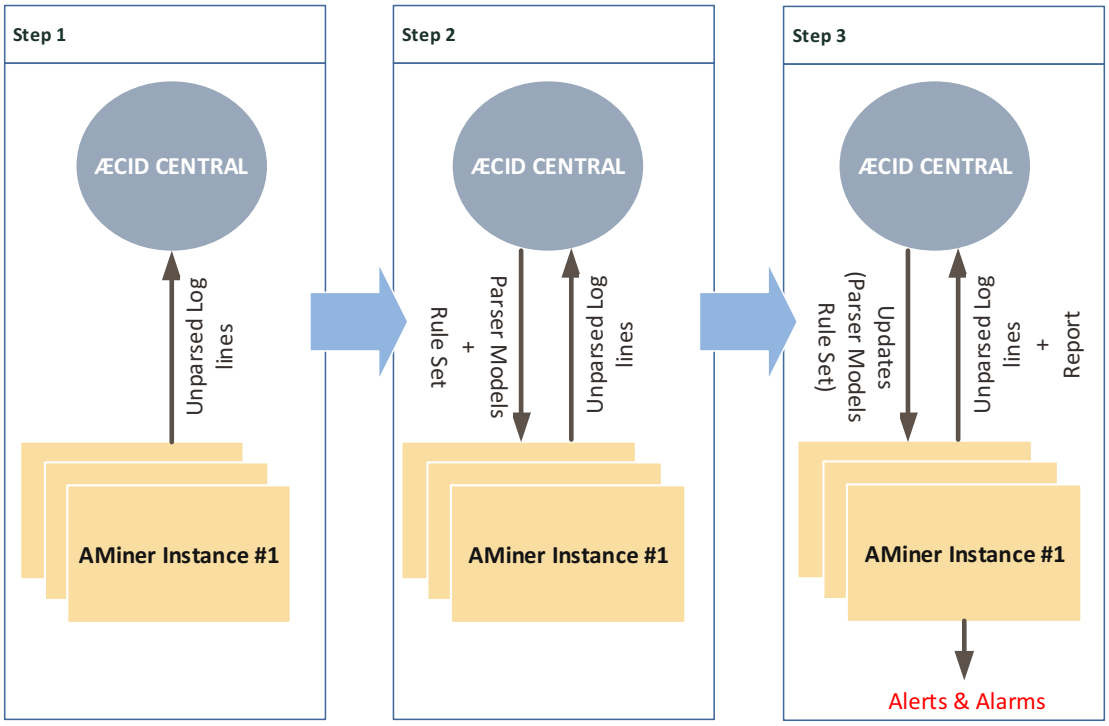


Figure 7.11: AECID initialization process [WSSF18].

Whenever an anomaly is revealed, the AMiner instance generates a notification report and, if necessary, sends it by email to a pre-configured list of recipients. Alerts and alarms are also reported to AECID Central, where they can be aggregated and visualized.

Finally, in case sufficient resources and computational power are available on a single node, AECID can be deployed in its full-fledged setup on a stand-alone machine. In this scenario, AECID Central and the AMiner instance operate on the same node. The advantage of this topology, compared to the first one, lies in the fact that the administrators neither need to manually determine the parser models nor to define the set of rules, because AECID Central automatically generates them following the three-step approach described above and depicted in Figure 7.11.

7.2.3 Application scenarios

The multi-layer light-weight detection approach presented in this section introduces a number of benefits that make its adoption attractive for a series of application scenarios beyond standard anomaly detection. This section explores some of the most promising use cases, in which the employment of AECID, stand-alone or in conjunction with other security solutions, would be highly advantageous.

AECID can be adopted to analyze events logged by systems running on different layers

of the OSI model. If applied on network traffic information, AECID can identify and keep track of the links established between different systems in the network, and perform a *network interaction graph analysis*. Observing which network nodes interact with each-other at which frequency, would allow AECID to build a 'communication-behavior' model, and to promptly identify any divergence from such a model, which could indicate internal or external malicious attempts to access network systems.

Similarly, AECID could be employed to analyze events recorded on the application layer, particularly when users authenticate on the numerous services deployed in an enterprise network. *Authentication interaction graph analysis* can be performed using AECID, to monitor which users authenticate to which services with what frequency. The 'authentication-behavior' model established by AECID in this scenario would allow to reveal any unusual authentication attempt, pinpointing potential intrusions, illegitimate access to critical resources, or erroneous authentication.

Moreover, AECID could serve as additional security layer, besides signature-based and other black-listing solutions, such as firewalls. In this setup, AECID would improve the overall detection capability by allowing the identification of previously unknown threats, and the verification of suspicious triggered alerts. The false positive rate (FPR) as well as the number of false negatives (FN) would decrease effectively and a higher level of security could be achieved. The alarms triggered by AECID could then be fed into a Security Information and Event Management (SIEM) system, which would correlate them with the events generated by other sensors [SLW+20, SWF18].

A further promising application area is CPS. CPS of the future will be the backbone of *Industry 4.0*, and will operate following a self-adaptation paradigm, which foresees that the components of a system are capable of configuring, protecting and healing themselves when certain internal and/or external conditions demand to [MMW+17]. The process of self-adaptation follows four principal phases: *monitor*, *analysis*, *plan*, and *execute*. Critical events, observed in the systems (in the monitor phase) and opportunely examined (in the analysis phase), would trigger specific changes in the system configuration (through the execute phase), which would follow suitable adaptation policies (evaluated in the plan phase). In the context of cyber security this approach could allow CPS being targeted by security threats to timely identify the indicators of an attack and swiftly react to contain its effects, reducing its impact. In this scenario, employing AECID in the monitoring and analysis phase, would be an asset. Thanks to their light-weight nature, AMiner instances could be installed on numerous low-power components deployed across the CPS, which would neither be able to run any traditional IDS directly nor have connections with feasible bandwidths to allow a continuous data stream to a centralized log store. By recording system events they would allow to have an accurate and comprehensive overview of the security situation of the entire CPS in real-time. The anomalies identified by AECID Central would then be evaluated and, in line with predefined security policies, trigger configuration changes in the monitored CPS, that contain the effect of the detected threat. A detailed description of this application scenario and a proof of concept applying AECID in this scenario is described in [SSK+18].

Finally, the system behavior model built by AECID Central enables AECID to recognize critical security events previously unseen, which may potentially indicate the occurrence of a *zero-day* attack. Integrating AECID with a cyber threat intelligence (CTI) management system would allow security operation centers to greatly improve their incident handling capability. Indicators of compromise (IoC), obtained by inspecting the anomalies revealed by AECID, could in fact be combined and correlated with the intelligence gathered from multiple data sources by CTI management solutions (such as the tool proposed in [SSS+17]). This correlation is fundamental to interpret the IoCs, confirm the occurrence of an attack, and prepare possible mitigation strategies. Additionally, this integrated framework would allow to dynamically reconfigure any deployed monitoring system in order to center their focus towards those critical assets vulnerable to the discovered threat. Eventually, this solution also supports the definition of attack signatures, which would be used to update any black-listing security solutions deployed in the infrastructure, and guarantee a higher level of protection. A proof of concept and detailed description of this application scenario can be found in [LSW+19].

7.3 Complex logfile synthesis for rapid sandbox-benchmarking of security and computer network analysis tools

The final section of this chapter proposes a rather different application case that uses log data analysis for the generation of realistic test data for evaluation of IDS. Major parts of the remaining section have been published in [WSSS16, WSSF15, WS16].

Despite the fact that much progress has been made in the application of log data analysis for performing anomaly detection, there seems to be one key element missing that is required to facilitate the wide adoption of these techniques. There is no appropriate solution available that is able to pinpoint the specific requirements of a network infrastructure, including its hosts, to a log analysis solution. However, this is essential to evaluate if a certain anomaly detection system is capable of detecting attacks customized specific to this given infrastructure. The underlying problem is that due to the high degree of interconnection of distributed systems as well as their application- and domain-specific customization and configuration, there are hardly two networked systems that work exactly the same way. Therefore, one can argue that each and every system is unique, either as a result of its configuration, its application domain and/or the way it is utilized.

Consequently, it is hard for system operators to optimize anomaly detection solutions and configurations to their specific infrastructure and their specific requirements. Today, there are only unsatisfying options for said system operators to determine and optimize the configuration of log analysis and anomaly detection solutions. They can either infer some conclusions (i) from quite general evaluations in testbeds, or (ii) test with somewhat simplified data only which mostly do not reflect the real-world properties sufficiently; or (iii) run tests on their productive systems - which enables the most realistic evaluation

of an anomaly detection system and its configuration, however at the same time might expose the infrastructure to dangerous or unstable situations. The latter is especially true in case automatic decisions are derived based on the output of a log analysis solution, such as the reaction of an intrusion prevention system due to discovered security violations, or network performance optimizations due to identified resource allocation problems.

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

The remaining section provides the following contributions:

- We propose an approach for generating realistic log data for testing log analysis and anomaly detection solutions. Furthermore, we introduce a concept for a testbed that applies the approach for generating realistic test log data to automatically evaluate, compare and optimize anomaly detection solutions for specific infrastructures.
- Log data analysis approach: We introduce and describe in detail a novel approach to analyze real log data and capture the unique characteristics of an infrastructures normal behavior. In particular, the approach makes use of log line clustering to discover and describe common events reflected by log lines, and Markov chains to model the correlation and interdependencies of those.
- Log data generation: Once we got an understanding about the structure and properties of short sequences of log data from a real system, we apply a new approach to generate large volumes of semi-synthetic log data that follows the properties, such as the sequence of log lines in terms of log event types, of the data analyzed before. The advantage here, compared to a simple 'record & replay'-system is that the complexity of the output data can be controlled during the generation and therefore, data to evaluate different kinds of systems can be produced. Additionally, this approach enables us to introduce variations into the produced set (such as time stamps, IP addresses, system names etc.) - similarly to the real world.

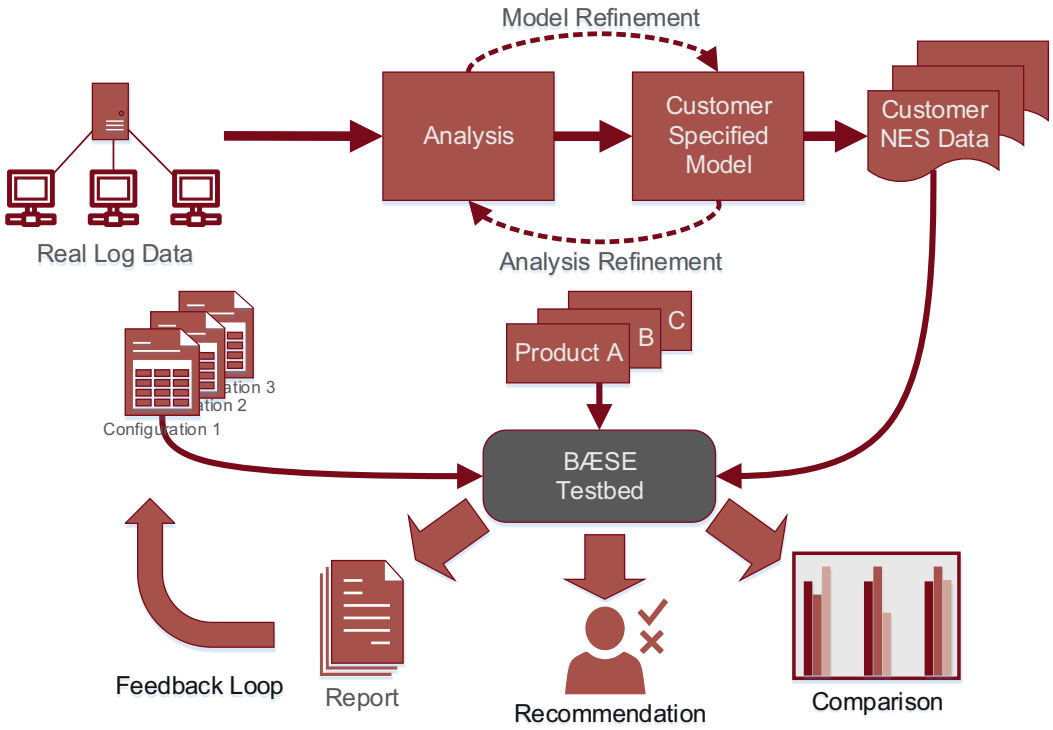


Figure 7.12: Testbed concept (adapted from [WSSF15, WS16, WSSS16]).

- Finally, to motivate and demonstrate the application of the proposed approach, we define a step-by-step enrollment process for IDS following standards from the International Organization for Standardization (ISO) [ISO15] and the National Institute of Standards and Technology (NIST) [SM07]. Based on this set-up process, we provide a qualitative evaluation by arguing which steps of the evaluation process can be simplified and optimized by applying our approach and highlight its benefits.

7.3.1 Testbed concept

The following section describes the concept of the BAESE – 'Benchmarking and Analytic Evaluation of IDS in Specified Environments' – testbed, first published in [WSSF15]. Figure 7.12 depicts the concept of the BAESE testbed. The BAESE approach consists of two major building blocks:

(i) Generation of semi-synthetic network[6] event sequence (NES) data [WSSS16] that bases on properties and characteristics of a specified network.

[6]Note, in the remaining chapter the term network includes the network infrastructure and its hosts.

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 7.1: This table summarizes the three common types of test data: synthetic, real and semi-synthetic. Also their advantages and disadvantages are pointed out [SSFF14].

(ii) Using generated NES data as input to various IDS for comparing, rating and evaluating their capabilities to detect pre-modeled attacks with respect to different configurations by applying the BAESE testbed.

The remaining section primarily focuses on the first part, because NES data generation applies log data analysis and therefore, approaches introduced in previous sections support this task.

NES data generation

The model for generating NES data takes a small part of captured log data from a computer network as input. Therefore, NES data can be classified as semi-synthetic test data (see Tab. 7.1). Note, the approach is not limited to textual log data and can potentially be applied to other sequential data available in textual format. The BAESE testbed processes the input using methods from statistics, probability theory and machine learning to obtain properties and characteristics of a considered network, including log line content and time intervals between log line occurrences. The acquired information is used to build new, semi-synthetic log lines. Finally, a network specific model is defined that uses, for example, Markov chain simulation [Nor98] to build NES data of any size and complexity. Iterative and interactive refinement of the analysis and the model allows to generate NES data of varying degrees of detail and to integrate anomalies into NES data.

Figure 7.13 depicts two procedures for generating NES data. The first one builds on clustering and applies the approaches for bio clustering (Ch. 3) or incremental clustering (Ch. 4) and generating character-based templates (Ch. 5), while the second one uses the parser generator approach (Ch. 6). Hence, the first procedure implements a character-based approach, and the second one applies a token-based approach.

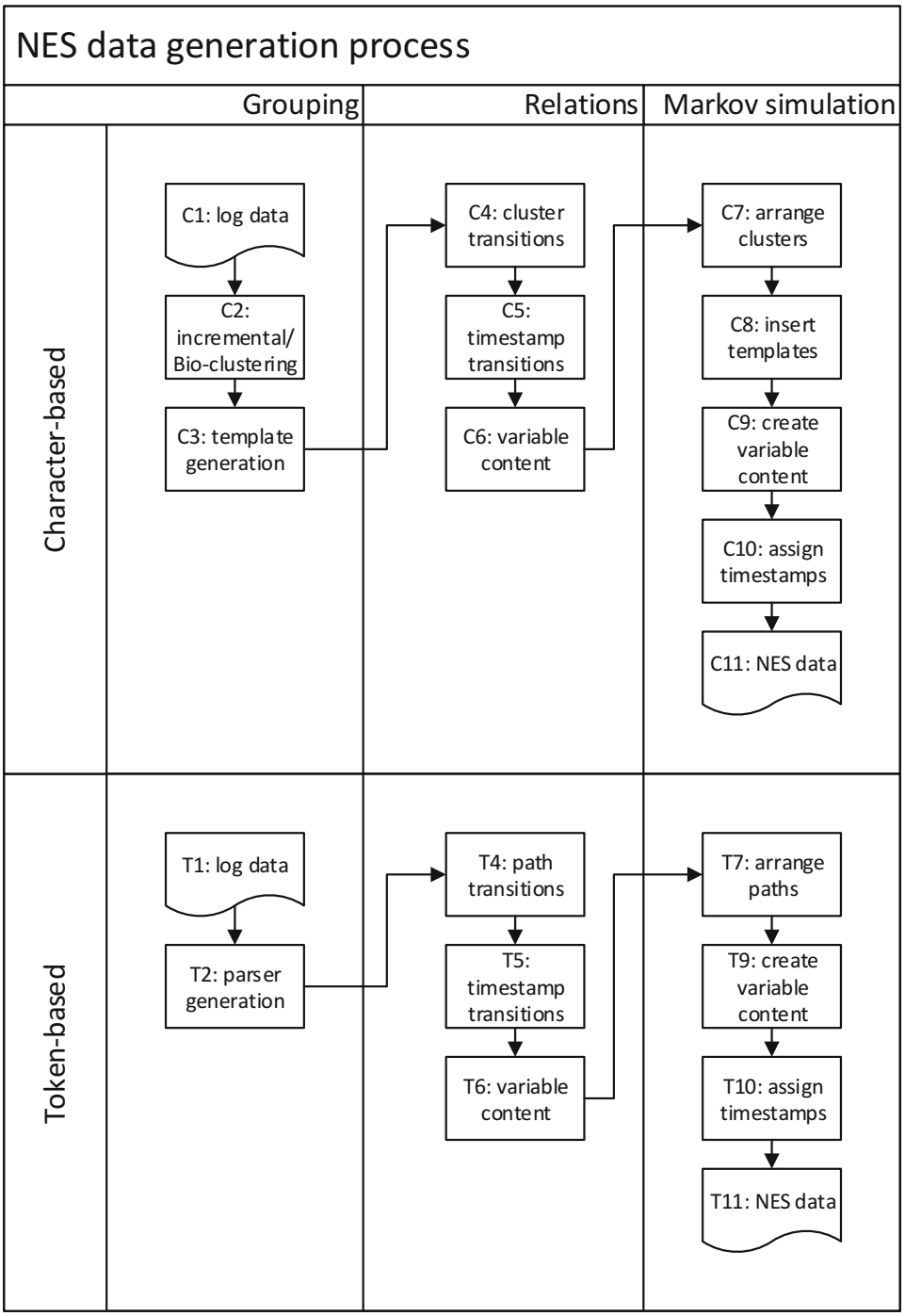


Figure 7.13: This figure depicts the process flow of the NES data generation process. The process comprises three steps: (i) *grouping* similar log lines, (ii) determine their *relations*, (iii) create NES data applying a *Markov chain simulation* using information obtained in the previous steps. The upper swimlane represents a character-based (C) and the lower one a token-based (T) procedure. Related steps share the same number.

The first procedure initially applies clustering (see Fig. 7.13 step C2) to group similar log lines. As mentioned both, the bio-clustering (Ch. 3), as well as the incremental-clustering (Ch. 4) approach can be used for this purpose. During the clustering the algorithm accounts for how often succeeding log lines have been assigned to each observed cluster. This information allows to calculate transition probabilities between all clusters (C4), i.e., how likely a line assigned to cluster A is succeeded by a line assigned to cluster B and so on. This information will later be used to build the transition matrix for a Markov chain simulation (C7). Similarly, the approach collects information on how long it takes until the monitored system generates the next log line (C5). Therefore, for each cluster a list containing the deltas between the current log line's timestamp and the timestamp of the succeeding log line is stored. This information will later be used to generate accurate and realistic timestamps (C10).

After the clustering, the approach generates cluster templates (C3) to identify static and variable parts in the log lines of each cluster. Additionally, for each variable in the cluster templates it stores a list containing all observed values of the variable (C6). The algorithm considers clusters that consist of only a single line separately. For the sake of simplicity, we assume it assigns those lines to a separate cluster, without computing a template. Furthermore, the approach accumulates the information on their transitions (C4). This step completes the analysis of the input data.

Next, a customer specified model (Fig. 7.12) that applies a Markov chain simulation is built. This model generates NES data of any size based on the properties of the real log data that have been extracted during the analysis. As transition matrix the Markov chain simulation uses the transition probabilities between the different clusters (C4) observed during the analysis. Eventually, the Markov chain simulation provides an ordered list of references to the different clusters (C7). The length of this list is equal to the intended number of lines of the NES data, and is therefore specified by the user. Afterwards, these references are replaced by the templates of the clusters (C8). Then, the variable parts of the templates are filled with content (C9). Therefore, for each variable an empirical distribution function is generated based on the observed values of the variables. For each variable, the corresponding empirical distribution function is used to specify its content. After this step, the NES data consists of a list of semi-synthetic log lines that have been generated based on the properties of the original input log data. Finally, timestamps are assigned to the log lines (C10). Again, an empirical distribution function is applied to specify the time span between consecutive log lines.

The second procedure initially generates a parser (T2) for the input log data, using the parser generator approach proposed in Ch. 6. Since the parser already includes information on static and variable parts of the log lines, no templates have to be generated. Furthermore, this procedure implements a token-based approach, because the parser generator splits log lines into tokens at specified separators. Similarly to the first procedure, during generating the parser the content of each variable (T6) is collected in a list. Besides the path frequencies, i.e., the probabilities by which certain branches occur in the log data (see Sec. 6.2.3), the parser generator learns the transition probabilities

between the different paths (T4) in the parser tree and assigns an ID to each path. Note, a path is defined as the sequence of nodes from the root node to a leaf node. Same as in the first procedure, the deltas between the timestamps of consecutive log lines are stored (T5).

After the analysis, again a Markov chain simulation represents the main building block of the specified model for generating NES data. The Markov chain simulation obtains its transition matrix from the transitions frequencies between the different paths (T7) observed during the analysis. The contents for the variable parts (T9) and the timestamps (T10) are generated the same way as in the first procedure.

A detailed description of the process for generating NES data is provided in [WSSS16]. It describes the first procedure that applies clustering. However, the version in [WSSS16] applies SLCT [Vaa03], a token-based clustering algorithm, instead of a character-based one.

BAESE testbed

Figure 7.12 outlines, that the BAESE testbed consists of three types of entities: (i) NES data, (ii) anomaly detection products, and (iii) configurations for these anomaly detection products. The remaining section discusses the operating principle of the BAESE testbed, using an example (see Fig. 7.14) that incorporates all anomaly detection algorithms developed in course of this thesis and the NES data generation procedures depicted in Fig. 7.13.

Considering the anomaly detection approaches discussed in this thesis, there occur configurations of different complexities. While the bio-clustering (Ch. 3) and the incremental clustering (Ch. 4) basically require only a single input parameter (the similarity threshold), more complex anomaly detection systems such as the time series analysis (Sec. 7.1) and AECID (Sec. 7.2) require either more model related input parameters or complex configuration files. The time series analysis demands besides the similarity threshold for the clustering, for example, parameters for the ARIMA model and a size for the time window in which log lines are clustered. AECID, requires a configuration file that includes defined parsers and detectors. Hence, there are two ways to use the BAESE testbed to simplify the configuration of AECID. First, it is possible to compare several pre-defined configuration files and optimize one for a specific network. Second, the BAESE testbed is used to configure certain components such as the parser by applying AECID-PG (Ch. 6). For the latter, the different thresholds θ, which the parser generator requires as input would be determined using the BAESE testbed.

Next, attacks or malicious behavior have to be integrated into the NES data to enable an in-depth evaluation of the different anomaly detection systems. The most straightforward option to change the system behavior is to simply delete certain lines, or lines generated by a certain service, such as a database, in the NES data. The latter could also be achieved by changing the transition probabilities to zero for clusters related to a specific service for a certain time period. An attacker might switch off, for example, database

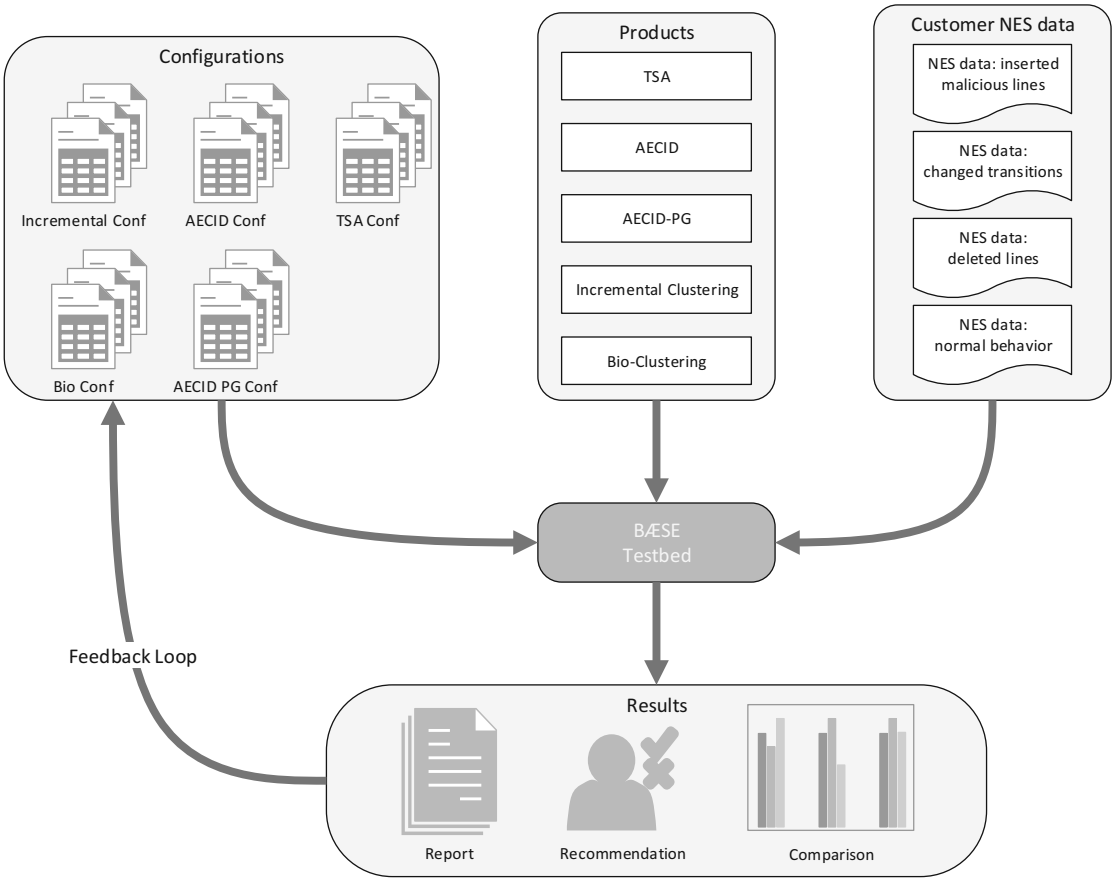


Figure 7.14: This figure shows a specific example of the BAESE testbed. It decpicts how the anomaly detection algorithms (*products*) developed in course of this thesis and different configurations (*configurations*) can be tested, compared and optimized using different types of *NES data*, generated with one of the procedures proposed in Fig. 7.13.

logging while tampering the database [FSSF15]. Altered system behavior like this could be detected by time series analysis and detectors of AECID. Another way to integrate malicious system behavior into NES data would be modifying parts of the transition probabilities for a certain time period. This would simulate anomalous system behavior related to uncommon user authentication actions, system process behavior, or web request behavior that could be observed in course of attacks tailored to a specific network and lead to atypical user actions [LSW+19, LSW+20]. Finally, log lines observed during attacks using known malware and exploits can be integrated into the NES data. These attacks would especially challenge anomaly detection systems that apply outlier detection, such as the bio-clustering and the incremental clustering. For example, it is possible to disclose log lines generated by cross site scripting attacks (XSS) or SQL injections, as done for the evaluation of the incremental clustering approach (cf. Sec. 4.2).

Finally, the BAESE testbed applies the different anomaly detection algorithms and their configurations on the NES data which contains attacks and malicious user behavior. Therefore, the BAESE testbed inserts attacks into the NES data. The ground truth defined by the customer specified model that describes the normal network behavior, supports the BAESE testbed to determine which log lines do not fit the model and thus need to be considered anomalous. In case of time series analysis, it indicates in which time windows anomalous behavior occurred. The ground truth builds the foundation to determine true positives (TP), false positives (FP), true negatives (TN) and flase negatives (FN). These classifications describe if an anomaly detection algorithm predicted an anomaly correctly (TP), raised a false alarm (FP), correctly not predicted an alarm (TN), ignored, i.e., not detected, an anomaly (FN). Based on these categories essential statistics are computed to rate and evaluate anomaly detection algorithms.

The most commonly used and therefore most suitable for comparison of different algorithms are true positive rate (TPR) Eq. (7.6), false positive rate (FPR) Eq. (7.7), precision (how many selected items are relevant) Eq. (7.8), recall (how many relevant items are selected) Eq. (7.6), a measure for sensitivity that is equal to the TPR and F_β-Score (most commonly $\beta = 1$) Eq. (7.9), which is the harmonic mean of precision and recall [Pow11].

$$\text{TPR} = \text{Recall} = \frac{TP}{TP + FN} \tag{7.6}$$

$$\text{FPR} = \frac{FP}{FP + TN} \tag{7.7}$$

$$\text{Precision} = \frac{TP}{TP + FP} \tag{7.8}$$

$$F_\beta\text{-Score} = (1 + \beta^2)\frac{\text{Precision} \cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}} \tag{7.9}$$

Another way to compare algorithms and variations of configuration provide the receiver operating characteristic (ROC) curves. A ROC curve plots the FPR of an algorithm on the x-axis and its TPR on the y-axis. This means the closer a point is to the coordinates $(0, 1)$, which refer to a TPR of 1 (100%) and a FPR of 0 (0%), the better the results the algorithm yields. Therefore, several algorithms and/or configurations can be compared in a single ROC plot. Besides visual comparison, ROC curves also provide the possibility to calculate the area under curve (AUC) (see Eq. (7.10)), which enables value-base comparison. The AUC is the area limited by the curve defined through the linear connection of the points $(0, 0)$, (FPR,TPR) (representing the algorithm) and $(1, 1)$, and the x-axis. The larger the AUC the better the detection capabilities regarding TPR and FPR of the evaluated algorithm [Pow11].

$$\text{AUC} = \frac{\text{TPR} - \text{FPR} + 1}{2} \tag{7.10}$$

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

Table 7.2: Steps of the roll-out process of an IDS [ISO15, SM07, WSSS16].

Hence, the BAESE testbed can provide visual reports in form of plots of the F-Score and the ROC curves, and value-based feedback in form of mentioned statistics. The statistics allow to rate and rank the different algorithms and their configurations. Based on this rankings it is also possible to automatically provide recommendations on which algorithms and configurations are the most effective ones.

Finally, a feedback loop (see Fig.7.12) can be implemented to automatically alter configuration parameters, which, for example, enables optimization of configuration parameters of an algorithm. Thus, to achieve this optimization techniques such as grid and random search can be applied. While, grid search alters numerical parameters with equal steps, random search chooses parameters randomly within a pre-defined interval. State of the art research shows that random search is superior to grid search regarding parameter optimization [BB12].

7.3.2 Roll-Out of an IDS

In this section, we define the roll-out of an Intrusion Detection System (IDS) within a medium or large-scale enterprise IT environment in a step-by-step set-up process to show how much effort is required to achieve this. The two standards [ISO15] published by the International Organization for Standardization (ISO) and [SM07] published by the National Institute of Standards and Technology (NIST) serve as starting point. Other reports discussing roll-out of IDS following similar approaches are [Sny08, IMT10, Yak08]. Referring to this procedure, we point out which steps can be simplified and optimized applying the BAESE approach. Thus, this section provides a qualitative evaluation of the BAESE testbed and points out the challenges in the set-up process of advanced anomaly detection systems such as AECID, proposed in Sec. 7.2, or the TSA approach introduced in Sec. 7.1.

The roll-out of an IDS can be structured as shown in Tab. 7.2. According to the standards

[ISO15, SM07], the three main parts of the set-up process are selection, deployment and operation. Furthermore evaluation is part of all these three steps. Table 7.2 also summarizes which criteria is considered when and which actions are performed.

In the roll-out process the **EVALUATION** of IDS is the biggest challenge. Reasons for this on the one hand are that no standardized methodologies for testing IDS exist and on the other hand there are no standard test environments for IDS available [SM07]. Hence organizations depend on vendor brochures, white-papers and product demonstrations, which are usually not objective and therefore insufficient as well as on third-party reviews of individual products and comparisons of multiple products. Since every network infrastructure is different also tests in lab environments are insufficient to rate the performance of IDS in specific environments. This circumstances either force organizations to perform evaluations on their running productive systems, which might expose the infrastructure to dangerous and unstable situations, or require them to run simplified tests in staged environments [ISO15, SM07]. The BAESE approach aims to fill this gap. Therefore, as shown in Sec. 7.3.1 the BAESE approach allows to generate high quality NES data, which enables detailed simulation of an organization's network infrastructure which then can be exploited for extensive evaluations.

Selection

Since there exist various IDS, the first step of enrolling a product is **SELECTION**. Therefore, on the one hand the criteria summarized in Tab. 7.2 and on the other hand the system environment, IDS security policies and financial costs build a basis for selecting an appropriate IDS candidates. Decisions based on system environment, IDS security policies and financial costs as well as on the type of the required IDS and the provided technical support have to be made by ICT security experts. The other criteria account for evaluation methods as mentioned in the beginning of this section. In this context, our approach allows to evaluate the performance of IDS applying techniques described in Sec. 7.3.1. In opposite to testing an IDS directly in an organization's network infrastructure the BAESE approach also allows to evaluate the scalability of a product, for example, by rescaling the time differences between consecutive log lines, which simulates a larger volume of network traffic. The scalability of a product is important, because otherwise in case an organization's network infrastructure grows, a new IDS solutions has to be selected. BAESE can be also applied for testing capabilities of a product. For example, to evaluate the detection capability of an IDS the system it monitors has to be attacked. Section 7.3.1 discusses how cyber attacks and malicious user behavior can be simulated using the BAESE approach. This is important since it is not advisable to verify the effectiveness of an IDS in a network environment by self-attacking it.

Deployment

After selecting an IDS the **DEPLOYMENT** process starts. First, an architecture of the IDS implementation has to be designed, which includes specifying the locations of sensors and also interactions with other system components are taken into account. Further,

both standards [ISO15, SM07] recommend a staged deployment, i.e. deploying an IDS first only for a small part of a network and then expanding it incremental, which makes it easier for the staff to acquire insights into new products. Furthermore, component tests simplify the evaluation of new products and lower the risk of problems during the deployment phase. BAESE enables performing off-line component tests with highly realistic NES data. Also the personnel has to be trained to get familiar with new IDS solutions. NES data makes it possible to accomplish this training within a sandbox environment and outside a running productive system, but still under highly realistic conditions. A major point of deployment is identifying a feasible configuration of the deployed IDS, i.e. the configuration that addresses the highest risks of the organization. The configuration also should be part of the selection phase, because it strongly influences the performance and effectiveness of an IDS. Since the configuration heavily depends on the network infrastructure – every network is different – it cannot be evaluated in a laboratory environment or based on published tests and comparisons of vendors. Therefore, generated NES data offers the advantage, that on the same highly realistic data set several configurations can be tested and compared easily and fast, which enhances and accelerates the configuration process tremendously. BAESE, for example, foresees a feedback loop that autonomously alters configuration parameters.

Operation

After the deployment, the **OPERATION** phase follows. In this context operation among other things covers maintenance, updating, alert handling and alert response, and also tuning the IDS as well as altering its configuration. Therefore periodic verification and evaluation has to be performed to continuously optimize the IDS. Updates, for example, can implicate altering the configuration. Therefore, it is possible to test the updated software first with generated NES data, which allows to adopt the configuration immediately. Since also the monitored infrastructure usually underlies frequent changes, it is possible to periodically generate new NES data files with the BAESE approach easily and fast, to continuously verify and evaluate the performance and effectiveness of the deployed IDS. If required, the IDS and configuration can be tuned and optimized accordingly. Also threats are evolving over time. Therefore, BAESE allows off-line testing of the detection capabilities of the applied IDS by simulating new threats in NES data files.

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

CHAPTER 8

Conclusion and future work

The thesis focused on log data analysis approaches that enable online anomaly detection even with limited resources. Thus, the goal was to develop algorithms that are able to efficiently process large amounts of data while simultaneously taking into account as much information as possible. Under these conditions, it turned out that incremental algorithms are an effective method.

RQ1: How can semi-/unsupervised machine learning techniques be applied to semi-/unstructured textual log data to enable online anomaly detection?

First, the thesis considered clustering, which is a well-established unsupervised machine learning method that enables outlier detection and provides a good overview on the events occurring in log data. However, traditional clustering approaches lack of performance when it comes to processing large amounts of semi- and unstructured data such as log lines. The thesis, provides two types of incremental log clustering approaches that allow to cluster large data sets, even infinitely growing data sets, and enable online outlier detection by implementing a density-based single pass clustering algorithm that processes each log line once and does not require a pre-defined number of clusters.

RQ2: To what extent is it possible to describe the content of log data during normal system operation?

While clustering allows to group similar log events and indicate rare events that refer to anomalies, it does not provide a satisfying solution for describing a log line cluster's content. Therefore, the thesis provides a novel approach for generating character-based log templates by solving the problem of generating multi-line alignments for any string. Furthermore, this approach eliminates disadvantages of token-based template generators that are, for example, not able to handle similar but not equal strings properly and often cover large parts of log lines with wildcards that represent variable log line content. Furthermore, we demonstrated that character-based templates show a significantly better coverage of log lines than token-based ones. Although the coverage depends on many

factors, such as data complexity and similarity used for clustering, the approaches we proposed improved coverage by up to 20% compared to token-based approaches. In our experiments, we reached a coverage between 70% and 95% applying our character-based template generator to different data sets.

RQ3: How can log line parsing be optimized to enable online processing of log lines with minimal information loss when analyzing large amounts of data with limited resources?

State of the art log line parsers apply lists of signatures or regular expressions. Due to the large amount of data and number of different event types occurring in modern networks' log data, this approach is inefficient and consumes large amounts of resources. Furthermore, to be able to process log lines online, large parts of log lines are neglected during parsing or only specific parts are looked at, which implies a loss of information. Additionally, because of frequent changes in hard- and software and consequently also in the produced log data, defining and maintaining such parsers is time consuming. To mitigate these problems, the thesis designed a novel parser generator that provides tree-like parsers, which significantly reduces complexity of parsing.

RQ4: How can online log analysis algorithms be applied?

Finally, the thesis presents three different applications that benefit from the developed algorithms. The first one introduces a TSA approach that applies the incremental clustering approach. The second one is a log-based anomaly detection system that applies the tree-like parser. The final one uses all of the proposed algorithms to generate semi-synthetic log data and autonomously evaluate, compare and optimize IDS.

The last chapter of the thesis provides a first glimpse on how online log analysis algorithms can be applied. Besides improving implementation of the proposed approaches, future work includes providing mechanisms that make the algorithms adaptive. This means, they should be capable of including changes in the system behavior into their model. For example, the clustering should be able to remove old clusters and include new ones. Similarly, the parser generator should be able to adapt the parser tree over time. However, this raises the major question, when new log lines that do not match the current model of normal system behavior should be considered part of these model or anomalous. One solution could be linking the clustering and the parser generator. In this case, for example, if a new cluster reaches a certain size, the character-based template generator could be applied to generate a template that afterwards is integrated into the parser tree.

List of Figures

List of Tables

Bibliography

[AGM+90] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[BE67] Leonard E Baum and John Alonzo Eagon. An inequality with applications to statistical estimation for probabilistic functions of markov processes and to a model for ecology. *Bulletin of the American Mathematical Society*, 73(3):360–363, 1967.

[Ber06] Pavel Berkhin. A survey of clustering data mining techniques. In *Grouping Multidimensional Data*, pages 25–71. Springer, 2006.

[BG16] Anna L Buczak and Erhan Guven. A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Communications Durveys & Tutorials*, 18(2):1153–1176, 2016.

[BGRS99] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *International Conference on Database Theory*, pages 217–235. Springer, 1999.

[Can98] James Cannady. Artificial neural networks for misuse detection. In *National Information Systems Security Conference*, volume 26, pages 443–456, 1998.

[CBK09] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(3):1–58, 2009.

[CC08] Jonathan D Cryer and Kung-Sik Chan. *Time series analysis: with applications in R*. Springer, 2008.

[Cer69] Vinton Gray Cerf. ASCII format for network interchange: RFC20. *Internet Engineering Task Force*, 1969.

[Chr06] Peter Christen. A comparison of personal name matching: Techniques and practical issues. In *Sixth IEEE International Conference on Data Mining-Workshops (ICDMW'06)*, pages 290–294. IEEE, 2006.

[CKT06] Deepayan Chakrabarti, Ravi Kumar, and Andrew Tomkins. Evolutionary clustering. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 554–560, 2006.

[CSP12] Anton Chuvakin, Kevin Schmidt, and Chris Phillips. *Logging and log management: The authoritative guide to understanding the concepts surrounding logging and log management.* Newnes, 2012.

[CSZ+09] Yun Chi, Xiaodan Song, Dengyong Zhou, Koji Hino, and Belle L Tseng. On evolutionary spectral clustering. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 3(4):1–30, 2009.

[DC15] Sizhong Du and Jian Cao. Behavioral anomaly detection approach based on log monitoring. In *International Conference on Behavioral, Economic and Socio-Cultural Computing (BESC)*, pages 188–194. IEEE, 2015.

[Edg10] Robert C Edgar. Search and clustering orders of magnitude faster than BLAST. *Bioinformatics*, 26(19):2460–2461, 08 2010.

[EES10] Mohamed E Elhamahmy, Hesham N Elmahdy, and Imane A Saroit. A new approach for evaluating intrusion detection system. *CiiT International Journal of Artificial Intelligent Systems and Machine Learning*, 2(11):290–298, 2010.

[FHH+11] Ru Fang, Hui-I Hsiao, Bin He, C Mohan, and Yun Wang. High performance database logging using storage class memory. In *IEEE 27th International Conference on Data Engineering (ICDE)*, pages 1221–1231. IEEE, 2011.

[Fir20] Fireeye Mandiant Services. Special report m-trends 2020. Technical report, FireEye, 2020.

[FKS+12] Peter Frühwirt, Peter Kieseberg, Sebastian Schrittwieser, Markus Huber, and Edgar Weippl. Innodb database forensics: reconstructing data manipulation queries from redo logs. In *Seventh International Conference on Availability, Reliability and Security (ARES)*, pages 625–633. IEEE, 2012.

[FLWL09] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Ninth IEEE International Conference on Data Mining (ICDM'09)*, pages 149–158. IEEE, 2009.

[FSSF15] Ivo Friedberg, Florian Skopik, Giuseppe Settanni, and Roman Fiedler. Combating advanced persistent threats: From network event correlation to incident detection. *Computers & Security*, 48:35–57, 2015.

[FWABK17] Ivo Friedberg, Markus Wurzenberger, Abdullah Al Balushi, and Boojoong Kang. From monitoring, logging, and network analysis to threat intelligence extraction. In Florian Skopik, editor, *Collaborative Cyber Threat Intelligence: Detecting and Responding to Advanced Cyber Attacks at the National Level*, Auerbach book, pages 69–127. CRC Press, 2017.

[GCTMK11] Ana Gainaru, Franck Cappello, Stefan Trausan-Matu, and Bill Kramer. Event log mining tool for large scale hpc systems. In *European Conference on Parallel Processing*, pages 52–64. Springer, 2011.

[GDC10] Derek Greene, Donal Doyle, and Padraig Cunningham. Tracking the evolution of communities in dynamic social networks. In *2010 International Conference on Advances in Social Networks Analysis and Mining*, pages 176–183. IEEE, 2010.

[Ger09] Rainer Gerhards. The syslog protocol: Rfc 5424. *IETF Trust: Reston, VA, USA*, 2009.

[Ger10] Rainer Gerhards. rsyslog: going up from 40k messages per second to 250k. In *Linux Kongress*, 2010.

[GF13] Wael H Gomaa and Aly A Fahmy. A survey of text similarity approaches. *International Journal of Computer Applications*, 68(13):13–18, 2013.

[GL12] Rainer Gerhards and Chris Lonvick. Rfc 6587-transmission of syslog messages over tcp, 2012.

[GLP11] Mohammadreza Ghodsi, Bo Liu, and Mihai Pop. Dnaclust: accurate and efficient clustering of phylogenetic marker genes. *BMC Bioinformatics*, 12(1):271, 2011.

[GTDVMFV09] Pedro Garcia-Teodoro, Jesus Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & Security*, 28(1-2):18–28, 2009.

[GU16] Markus Goldstein and Seiichi Uchida. A comparative evaluation of unsupervised anomaly detection algorithms for multivariate data. *PloS one*, 11(4):e0152173, 2016.

[H+98] David Heckerman et al. A tutorial on learning with bayesian networks. *Nato Asi Series D Behavioural And Social Sciences*, 89:301–354, 1998.

[HA93] Stephen E Hansen and E Todd Atkins. Automated system monitoring and notification with swatch. In *LISA*, volume 93, pages 145–152, 1993.

[HA04] Victoria J Hodge and Jim Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

[HDX+16] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. Logmine: Fast pattern recognition for log analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1573–1582. ACM, 2016.

[HH10] David Harris and Sarah Harris. *Digital design and computer architecture*. Morgan Kaufmann, 2010.

[Hir75] Daniel S Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[HS88] Desmond G Higgins and Paul M Sharp. Clustal: a package for performing multiple sequence alignment on a microcomputer. *Gene*, 73(1):237–244, 1988.

[HS98] Liisa Holm and Chris Sander. Removing near-neighbour redundancy from large protein sequence collections. *Bioinformatics (Oxford, England)*, 14(5):423–429, 1998.

[Hua08] Anna Huang. Similarity measures for text document clustering. In *Proceedings of the Sixth New Zealand Computer Science Research Student Conference (NZCSRSC2008)*, volume 4, pages 9–56, 2008.

[HZH+16] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. An evaluation study on log parsing and its use in log mining. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 654–661. IEEE, 2016.

[HZH+17] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. Towards automated log parsing for large-scale log data analysis. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[HZHL16] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 207–218. IEEE, 2016.

[HZZL17] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE*

International Conference onWeb Services (ICWS), pages 33–40. IEEE, 2017.

[IMT10] Paul Innella, Oba McMillan, and David Trout. *Managing Intrusion Detection Systems in Large Organizations, Part One*. Symantec, 2010.

[ISO15] ITU-T Recommendation ISO. Iso/iec 27039. *Information technology–Security techniques–Selection, deployment and operations of intrusion detection systems*, 2015.

[Jam80] James P. Anderson. Computer security threat monitoring and surveillance. Technical Report 17, James P. Anderson Company, Fort Washington, Pennsylvania, April 1980.

[Jar89] Matthew A Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association*, 84(406):414–420, 1989.

[JHHF08] Zhen Ming Jiang, Ahmed E Hassan, Gilbert Hamann, and Parminder Flora. An automated approach for abstracting execution logs to execution events. *Journal of Software: Evolution and Process*, 20(4):249–267, 2008.

[JM09] Daniel Jurafsky and James H Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Pearson International Edition. Prentice Hall, 2009.

[JMK14] Seyyed-Mohammad Javadi-Moghaddam and Stefanos Kollias. A fuzzy similarity measure for xml documents. *International Journal of Information Technology and Computer Science (IJITCS)*, 13(2):9–17, April 2014.

[KFE12] Rebecca Killick, Paul Fearnhead, and Idris A Eckley. Optimal detection of changepoints with a linear computational cost. *Journal of the American Statistical Association*, 107(500):1590–1598, 2012.

[KFE14] Satoru Kobayashi, Kensuke Fukuda, and Hiroshi Esaki. Towards an nlp-based log template generation algorithm for system log analysis. In *Proceedings of the 9th International Conference on Future Internet Technologies*, page 11. ACM, 2014.

[KIM+14] Tatsuaki Kimura, Keisuke Ishibashi, Tatsuya Mori, Hiroshi Sawada, Tsuyoshi Toyono, Ken Nishimatsu, Akio Watanabe, Akihiro Shimoda, and Kohei Shiomoto. Spatio-temporal factorization of log data for understanding network events. In *IEEE Conference on Computer Communications (INFOCOM)*, pages 610–618. IEEE, 2014.

[KKTM05] Kazutaka Katoh, Kei-ichi Kuma, Hiroyuki Toh, and Takashi Miyata. Mafft version 5: improvement in accuracy of multiple sequence alignment. *Nucleic Acids Research*, 33(2):511–518, 2005.

[KL10] Marius Kloft and Pavel Laskov. Online anomaly detection under adversarial impact. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 405–412, 2010.

[Lev66] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, volume 10, pages 707–710, 1966.

[LJG02] Weizhong Li, Lukasz Jaroszewski, and Adam Godzik. Tolerating some redundancy significantly speeds up clustering of large protein databases. *Bioinformatics*, 18(1):77–82, 01 2002.

[LLC] L. A. N. S. LLC. Operational data to support and enable computer science research.

[LP85] David J Lipman and William R Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985.

[LRLLT13] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, January 2013.

[LSW+19] Max Landauer, Florian Skopik, Markus Wurzenberger, Wolfgang Hotwagner, and Andreas Rauber. A framework for cyber threat intelligence extraction from raw log data. In *International Workshop on Big Data Analytics for Cyber Threat Hunting (CyberHunt 2019) in conjunction with the IEEE International Conference on Big Data 2019*, pages 1–10. IEEE, 2019.

[LSW+20] Max Landauer, Florian Skopik, Markus Wurzenberger, Wolfgang Hotwagner, and Andreas Rauber. Visualizing syscalls using self-organizing maps for system intrusion detection. In *Proceedings of the 6th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP,*, pages 349–360. INSTICC, SciTePress, 2020.

[LSWR20] Max Landauer, Florian Skopik, Markus Wurzenberger, and Andreas Rauber. System log clustering approaches for cyber security applications: A survey. *Computers & Security*, 92:101739, 2020.

[LTPM17] Laetitia Leichtnam, Eric Totel, Nicolas Prigent, and Ludovic Mé. Starlord: Linked security data exploration in a 3d graph. In *IEEE Symposium on Visualization for Cyber Security (VizSec)*, pages 1–4. IEEE, 2017.

[LWS+18a] Max Landauer, Markus Wurzenberger, Florian Skopik, Giuseppe Settanni, and Peter Filzmoser. Dynamic log file analysis: an unsupervised cluster evolution approach for anomaly detection. *Computers & Security*, 79:94–116, 2018.

[LWS+18b] Max Landauer, Markus Wurzenberger, Florian Skopik, Giuseppe Settanni, and Peter Filzmoser. Time series analysis: unsupervised anomaly detection beyond outlier detection. In *International Conference on Information Security Practice and Experience*, pages 19–36. Springer, 2018.

[Miz13] Masayoshi Mizutani. Incremental mining of system log format. In *2013 IEEE International Conference on Services Computing (SCC)*, pages 595–602. IEEE, 2013.

[MMW+17] Angelika Musil, Juergen Musil, Danny Weyns, Tomas Bures, Henry Muccini, and Mohammad Sharaf. Patterns for self-adaptation in cyber-physical systems. In *Multi-Disciplinary Engineering for Cyber-Physical Production Systems*, pages 331–368. Springer, 2017.

[Mou04] David W. Mount. *Bioinformatics: Sequence and Genome Analysis*. CSHL Press, 2004.

[MP17] Vlado Menkovski and Milan Petkovic. Towards unsupervised signature extraction of forensic logs. In *Benelearn 2017: Proceedings of the Twenty-Sixth Benelux Conference on Machine Learning, Technische Universiteit Eindhoven, 9-10 June 2017*, page 154, 2017.

[MPB+18] Salma Messaoudi, Annibale Panchella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. A search-based approach for accurate identification of log message formats. In *Proceedings of the 26th IEEE/ACM International Conference on Program Comprehension (ICPC'18)*. ACM, 2018.

[MRS17] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2017.

[MZHM09] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1255–1264. ACM, 2009.

[MZHM12] Adetokunbo Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering*, 24(11):1921–1936, 2012.

[NJCY14] Xia Ning, Geoff Jiang, Haifeng Chen, and Kenji Yoshihira. Hlaer: a system for heterogeneous log analysis. 2014.

[Nor98] James R Norris. *Markov Chains*. Number 2. Cambridge university press, 1998.

[Not07] Cédric Notredame. Recent evolutions of multiple sequence alignment algorithms. *PLoS Computational Biology*, 3(8):e123, 2007.

[NW70] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[Okm09] Anton Okmianski. Rfc 5426-transmission of syslog messages over udp. *IETF*, 2009.

[OS07] Adam Oliner and Jon Stearley. What supercomputers say: A study of five system logs. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 575–584. IEEE, 2007.

[PG07] Jimin Pei and Nick V Grishin. Promals: towards accurate multiple sequence alignments of distantly related proteins. *Bioinformatics*, 23(7):802–808, 2007.

[PL88] William R Pearson and David J Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988.

[Pos80] Jon Postel. Rfc0768: User datagram protocol. *IETF*, 1980.

[Pow11] David M Powers. *Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation*. Bioinfo Publications, 2011.

[Rag13] Sriram Raghavan. Digital forensic research: current state of the art. *CSI Transactions on ICT*, 1(1):91–114, 2013.

[SC08] Ingo Steinwart and Andreas Christmann. *Support vector machines*. Springer Science & Business Media, 2008.

[Shi16] Keiichi Shima. Length matters: Clustering system log messages using length of words. *arXiv preprint arXiv:1611.03213*, 2016.

[Shl14] Jonathon Shlens. A tutorial on principal component analysis. *arXiv preprint arXiv:1404.1100*, 2014.

[SL91] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(3):660–674, 1991.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[SLW+20] Florian Skopik, Max Landauer, Markus Wurzenberger, Gernot Vormayr, Jelena Milosevic, Joachim Fabini, Wolfgang Prüggler, Oskar Kruschitz, Benjamin Widmann, Kevin Truckenthanner, et al. synERGY: Cross-correlation of operational and contextual data to timely detect and mitigate attacks to cyber-physical systems. *Journal of Information Security and Applications*, 54:102544, 2020.

[SM07] Karen Scarfone and Peter Mell. Guide to intrusion detection and prevention systems (idps). *NIST special publication*, 800(2007):94, 2007.

[SM08] Farzad Sabahi and Ali Movaghar. Intrusion detection: A survey. In *2008 Third International Conference on Systems and Networks Communications*, pages 23–26. IEEE, 2008.

[SNTS06] Myra Spiliopoulou, Irene Ntoutsi, Yannis Theodoridis, and Rene Schult. Monic: modeling and monitoring cluster transitions. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 706–711. ACM, 2006.

[Sny08] Joel Snyder. *Guide to Network Intrusion Prevention Systems*. PCWorld, 2008.

[SSFF14] Florian Skopik, Giuseppe Settanni, Roman Fiedler, and Ivo Friedberg. Semi-synthetic data set generation for security software evaluation. In *2014 Twelfth Annual International Conference on Privacy, Security and Trust*, pages 156–163. IEEE, 2014.

[SSK+18] Giuseppe Settanni, Florian Skopik, Anjeza Karaj, Markus Wurzenberger, and Roman Fiedler. Protecting cyber physical production systems using anomaly detection to enable self-adaptation. In *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, pages 173–180. IEEE, 2018.

[SSS+17] Giuseppe Settanni, Yegor Shovgenya, Florian Skopik, Roman Graf, Markus Wurzenberger, and Roman Fiedler. Acquiring cyber threat intelligence through security information correlation. In *2017 3rd IEEE International Conference on Cybernetics (CYBCONF)*, pages 1–7. IEEE, 2017.

[SW+81] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[SWF18] Florian Skopik, Markus Wurzenberger, and Roman Fiedler. synERGY: Detecting advanced attacks across multiple layers of cyber-physical systems. *ERCIM NEWS*, (114):30–31, 2018.

[SWZ15] Erich Schubert, Michael Weiler, and Arthur Zimek. Outlier detection and trend detection: Two sides of the same coin. In *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*, pages 40–46, Nov 2015.

[Tan11] Colin Tankard. Advanced persistent threats and how to monitor and deter them. *Network Security*, 2011(8):16–19, 2011.

[Tay86] William Ramsay Taylor. The classification of amino acid conservation. *Journal of Theoretical Biology*, 119(2):205 – 218, 1986.

[TBG+11] Narate Taerat, Jim Brandt, Ann Gentile, Matthew Wong, and Chokchai Leangsuksun. Baler: deterministic, lossless log message clustering tool. *Computer Science-Research and Development*, 26(3-4):285, 2011.

[Tea20] LogPAI Team. Logpai/logparser, 2020. https://github.com/logpai/logparser [last accessed 03/09/2020].

[TK03] Masashi Toyoda and Masaru Kitsuregawa. Extracting evolution of web communities from a series of web archives. In *Proceedings of the Fourteenth ACM Conference on Hypertext and Hypermedia*, pages 28–37, 2003.

[TL10] Liang Tang and Tao Li. Logtree: A framework for generating system events from raw textual logs. In *2010 IEEE 10th International Conference on Data Mining (ICDM)*, pages 491–500. IEEE, 2010.

[TLP11] Liang Tang, Tao Li, and Chang-Shing Perng. Logsig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, pages 785–794. ACM, 2011.

[TMP17] Stefan Thaler, Vlado Menkonvski, and Milan Petkovic. Towards a neural language model for signature extraction from forensic logs. In *2017 5th International Symposium on Digital Forensic and Security (ISDFS)*, pages 1–6. IEEE, 2017.

[Vaa03] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations*

& Management (IPOM 2003)(IEEE Cat. No. 03EX764), pages 119–126. IEEE, 2003.

[Vaa04] Risto Vaarandi. A breadth-first algorithm for mining frequent patterns from event logs. In *Intelligence in Communication Systems*, pages 293–308. Springer, 2004.

[Vac13] John R Vacca. *Managing information security*. Elsevier, 2013.

[VP15] Risto Vaarandi and Mauno Pihelgas. Logcluster - a data clustering and pattern mining algorithm for event logs. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 1–7. IEEE, 2015.

[WFHP16] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[WHL+20] Markus Wurzenberger, Georg Höld, Max Landauer, Florian Skopik, and Wolfgang Kastner. Creating character-based templates for log data to enable security event classification. In *Proceedings of the 2020 Asia Conference on Computer and Communications Security (AsiaCCS2020)*. ACM, 2020.

[Win90] William E Winkler. *String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage*. ERIC, 1990.

[WLSK19] Markus Wurzenberger, Max Landauer, Florian Skopik, and Wolfgang Kastner. AECID-PG: A tree-based log parser generator to enable log analysis. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 7–12. IEEE, 2019.

[WM12] Michael E Whitman and Herbert J Mattord. *Principles of information security*. Course Technology, Cengage Learning, Stamford, Conn., 4. ed., international ed edition, 2012.

[WOHN06] Iain M Wallace, Orla O'sullivan, Desmond G Higgins, and Cedric Notredame. M-Coffee: combining multiple sequence alignment methods with T-Coffee. *Nucleic Acids Research*, 34(6):1692–1699, 2006.

[WS16] Markus Wurzenberger and Florian Skopik. The baese testbed-analytic evaluation of it security tools in specified network environments. *ERCIM NEWS*, (107):51–52, 2016.

[WSFK16] Markus Wurzenberger, Florian Skopik, Roman Fiedler, and Wolfgang Kastner. Discovering insider threats from log data with high-performance bioinformatics tools. In *Proceedings of the 8th ACM CCS International*

Workshop on Managing Insider Security Threats, MIST '16, pages 109–112. ACM, 2016.

[WSFK17] Markus Wurzenberger, Florian Skopik, Roman Fiedler, and Wolfgang Kastner. Applying high-performance bioinformatics tools for outlier detection in log data. In *2017 3rd IEEE International Conference on Cybernetics (CYBCONF)*, pages 1–8. IEEE, 2017.

[WSFK21] Markus Wurzenberger, Florian Skopik, Roman Fiedler, and Wolfgang Kastner. Applying high-performance bioinformatics tools for outlier detection in log data. arXiv 2101.07113, 2021. Extended version of DOI: 10.1109/CYBConf.2017.7985760.

[WSL+17] Markus Wurzenberger, Florian Skopik, Max Landauer, Philipp Greitbauer, Roman Fiedler, and Wolfgang Kastner. Incremental clustering for semi-supervised anomaly detection applied on log data. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, page 31. ACM, 2017.

[WSS18] Markus Wurzenberger, Florian Skopik, and Giuseppe Settanni. Big data for cybersecurity. In Sherif Sakr and Albert Zomaya, editors, *Encyclopedia of Big Data Technologies*. Springer International Publishing, 2018.

[WSSF15] Markus Wurzenberger, Florian Skopik, Giuseppe Settanni, and Roman Fiedler. Beyond gut instincts: Understanding, rating and comparing self-learning idss. In *2015 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*, pages 1–1. IEEE, 2015.

[WSSF18] Markus Wurzenberger, Florian Skopik, Giuseppe Settanni, and Roman Fiedler. AECID: A self-learning anomaly detection approach based on light-weight log parser models. In *ICISSP*, pages 386–397, 2018.

[WSSS16] Markus Wurzenberger, Florian Skopik, Giuseppe Settanni, and Wolfgang Scherrer. Complex log file synthesis for rapid sandbox-benchmarking of security-and computer network analysis tools. *Information Systems*, 60:13–33, 2016.

[WZTS06] Jason T L Wang, Mohammed J Zaki, Hannu Toivonen, and Dennis Shasha. *Data Mining in Bioinformatics*. Springer Science & Business Media, March 2006.

[XHF+09] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 117–132, 2009.

[XW05] Rui Xu and Donald Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, 2005.

[Yak08] Edward Yakabovicz. *Intrusion detection system deployment recommendations*. TechTarget, 2008.

[Yer03] Francois Yergeau. Rfc 3629: Utf-8, a transformation format of iso 10646. *IETF*, 2003.

[YPZ12] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering*, pages 102–112. IEEE Press, 2012.

[ZMP+13] Farhana Zulkernine, Patrick Martin, Wendy Powley, Sima Soltani, Serge Mankovskii, and Mark Addleman. Capri: A tool for mining complex line patterns in large log data. In *Proceedings of the 2nd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, pages 47–54. ACM, 2013.

[ZZH17] Maosheng Zhang, Ying Zhao, and Zengmingyu He. Genlog: Accurate log template discovery for stripped x86 binaries. In *IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 337–346. IEEE, 2017.