TU WIEN Informatics

Ein halbüberwachter Ansatz zur Konfiguration und Optimierung von Machine-Learning basierten Anomalieerkennungs-Algorithmen

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Rahmen des Studiums

Computational Science and Engineering

eingereicht von

Viktor Beck, B. Sc.
Matrikelnummer 11713110

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber
Mitwirkung: Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Dr.rer.soc.oec. Florian Skopik
Dr. Dipl.-Ing. Markus Wurzenberger
Dr. Dipl.-Ing. Max Landauer

Wien, 8. August 2024

Viktor Beck | Andreas Rauber

TU WIEN Informatics

A Semi-supervised Approach for the Configuration and Optimization of Machine Learning based Anomaly Detection Algorithms

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computational Science and Engineering

by

Viktor Beck, B. Sc.
Registration Number 11713110

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber
Assistance: Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Dr.rer.soc.oec. Florian Skopik
Dr. Dipl.-Ing. Markus Wurzenberger
Dr. Dipl.-Ing. Max Landauer

Vienna, August 8, 2024

Viktor Beck | Andreas Rauber

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

Erklärung zur Verfassung der Arbeit

Viktor Beck, B. Sc.

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, 8. August 2024

Viktor Beck

Acknowledgements

I would like to express my deepest gratitude to the Austrian Institute of Technology and especially Florian Skopik, for providing me with this incredible opportunity and for the great support throughout my research. I am also deeply grateful to Max Landauer and Markus Wurzenberger for their prompt and insightful responses to my questions and for effectively guiding me through this process.

Special thanks go to my colleagues Wolfgang Hotwagner, Ernst Leierzopf and once again Max Landauer for providing a baseline for my evaluation.

I would like to thank Prof. Andreas Rauber for his guidance as my supervisor and for the professional support he provided throughout my thesis.

Finally, I would like to thank my parents, my brother and my friends for their unwavering support and encouragement.

This thesis was funded by the European Union under the European Defence Fund (GA no. 101103385 - AInception and GA no. 101121403 - NEWSROOM). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the granting authority can be held responsible for them.

Kurzfassung

Cyber-Bedrohungen entwickeln sich ständig weiter und neue Angriffstechniken werden rasch entwickelt. Anomalieerkennung (AE) in System-Logzeilen ist daher zunehmend wichtiger, da sie in der Lage ist, Angriffe bekannter, aber auch unbekannter Art zu erkennen. Die Konfiguration von AE-Algorithmen hängt stark von den Daten ab und umfasst die Auswahl von Merkmalen und die Festlegung von Parametern wie Schwellenwerten oder Fenstergrößen. Der Prozess ist folglich nicht trivial und erfordert oft manuelle Eingriffe von Experten, was Zugänglichkeit und Wirksamkeit von AE-Algorithmen einschränkt. Diese Arbeit stellt daher die Configuration-Engine (CE) vor, ein halbüberwachter Ansatz zur Automatisierung des Konfigurationsprozesses von AE-Algorithmen. Die CE wendet einen datenwissenschaftlichen Ansatz an, um Eigenschaften von Teilen von Logzeilen zu identifizieren. Dabei verwendet sie einen Parser, um in Zeilen sinnvolle statische und variable Tokens zu erkennen, die AE-Detektoren analysieren können. Das CE kategorisiert Variablen auf Grundlage ihrer Eigenschaften und ihres Verhaltens über die Zeit. Basierend auf den Anforderungen der vorliegenden AE-Detektoren legt die CE fest, welche Teile des Logs ein Detektor beobachten soll und bestimmt die entsprechenden Konfigurationsparameter. Diese Arbeit betrachtet 6 Detektoren des AMiners, einer fortgeschrittenen AE-Pipeline, die eine breite Palette von AE-Algorithmen umfasst. Zusätzlich enthält die CE einen Optimierungsansatz zur weiteren Verfeinerung von Konfigurationen.

Die Leistung wurde anhand punktueller und kollektiver Anomalien bewertet, die in einer Reihe von Apache Access- und Audit-Datensätzen auftreten. Bei kollektiven Anomalien lieferte das CE Konfigurationen, die eine durchschnittliche Präzision von über 0.95 für Apache- und über 0.9 für Audit-Datensätze für 5 der 6 Detektoren erreichten, während der Recall bei 1.0 lag. Damit konkurriert sie mit der Leistung der von drei verschiedenen Experten handgefertigten Konfigurationen, die die Grundlage für die Bewertung bildeten. Darüber hinaus verbesserte die Optimierung die Präzision von CE- und Expertenkonfigurationen in 29 von 32 Fällen für Apache-Daten und in 6 von 20 Fällen für Audit. Weiters können Konfigurationen als Dictionaries dargestellt und mittels Jaccard-Index auf Ähnlichkeit verglichen werden. Es zeigt sich, dass die Konfigurationen der Experten denen der CE signifikant unähnlich sind, während die des CE eine bemerkenswerte Ähnlichkeit über verschiedene Datensätze hinweg aufweisen. Dies spricht für eine effektive Übertragbarkeit der Konfigurationen auf verschiedene Datensätze desselben Typs. Die CE stellt einen signifikanten Fortschritt in AE dar, da es den Bedarf an Fachwissen und manueller Konfiguration reduziert und somit AE zugänglicher und effizienter macht.

Abstract

Cyber threats are continuously evolving, with new attack techniques developing rapidly. Anomaly detection (AD) in system log data is thereby an increasingly important task, as it is able to detect attacks of previously known but also unknown kind. The configuration of AD algorithms heavily depends on the data and includes complex feature selection and the definition of parameters such as thresholds or window sizes. This process is consequently not straightforward and often necessitates manual intervention by domain experts which restricts accessibility and effectiveness of AD algorithms. This work therefore introduces the Configuration-Engine (CE), a semi-supervised approach to automate the configuration process of AD algorithms. The CE applies a data science approach to identify properties of parts of log lines. Thereby, it uses a parser to recognize meaningful static and variable tokens in the log lines that AD detectors can analyze. The CE categorizes variables based on their characteristics and behavior over time. Based on the requirements of the AD detectors at hand, the CE specifies which log parts a detector should observe and determines the appropriate configuration parameters. This thesis considers a set of 6 different detectors of the AMiner, an advanced AD pipeline encompassing a wide range of AD algorithms. Additionally, the CE contains an optimization approach for further refinement of configurations.

The performance was evaluated considering point and collective anomalies occurring in a set of Apache Access and audit datasets. For collective anomalies the CE provided configurations that reached an average precision of over 0.95 for Apache and over 0.9 for audit datasets for 5 out of the 6 detectors, while maintaining a recall of 1.0 during detection. It thereby competes with the performance of handcrafted configurations by 3 different experts that formed the baseline for the evaluation. Additionally, the optimization improved the precision of both CE and expert configurations in 29 out of 32 cases for Apache data and in 6 out of 20 cases for audit. Moreover, the configurations can be represented as dictionaries and thus be compared for similarity using the Jaccard index. The experts' configurations are thereby significantly dissimilar to the ones of the CE. Meanwhile, the CE's configurations exhibit remarkable similarity to each other across various datasets, suggesting effective portability of CE configurations across different datasets of the same type. The CE represents a significant advancement in AD, reducing the need for domain expertise and manual configuration, making AD more accessible and efficient across different datasets and detection techniques.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

The landscape of cyber threats is constantly evolving, with novel attack methodologies emerging at a rapid pace. Current reports such as the ENISA Threat Landscape Report 2023 or the Crowdstrike 2024 Global Threat Report name ransomware as the top cybersecurity threat. The rise of AI-enabled disinformation and supply chain attacks are also major concerns, as are persistent DDoS threats and phishing or social engineering tactics used to gain initial access to systems. Moreover, attacks are becoming more targeted, with a focus on high-value sectors like manufacturing and industry. Especially, intrusions in cloud environments are strongly increasing [LCT+23, cro24].

The dynamic nature of these threats presents a serious challenge to organisations, governments and individuals alike. It necessitates a continuous state of vigilance and the implementation of robust security protocols. Once an unauthorised party gains access or control of a system, they can exfiltrate or manipulate sensitive data, implant malware that is able to corrupt or destroy computer infrastructures and disrupt critical services. These attacks can occur without detection by system administrators and the longer an attack persists unnoticed, the greater the potential damage. Consequently, early detection of potential intrusions is paramount to mitigate risks and minimise harm [cro24, WSSF18].

The majority of detection methods involve scanning for signatures such as hashes or IP addresses that correspond to known malware. As this approach only covers threats that have been monitored before, it is not possible to detect intrusions based on new and unknown techniques [LWS+23]. As a consequence, there is a growing trend towards the use of data science methods, particularly anomaly detection (AD), to address these challenges. These methods have gained popularity due to their ability to detect system states that deviate from normal system behaviour, thereby enabling the identification of both known and unknown intrusions [CBK09].

Intrusion detection systems (IDS) such as the AMiner [LWS+23] learn the normal behavior of a system based on its log data to uncover possible intrusions by identifying anomalies. The configuration of the AMiner tool requires customization based on the specific type of log files generated by the system which varies depending on the system's expected events and thus the appropriate detectors for analysis. This includes the determination of parameters and adjustment of thresholds. Therefore, the configuration of the detection tool is a non-trivial task that is carried out manually by a domain expert. The content of the configuration file determines which detectors the tool should employ and their corresponding parameter settings. The detection of anomalies presents significant challenges and is not always generic. Therefore, selecting the appropriate settings to adapt to the given context is critical to find anomalies while maintaining a low amount of false positives. To illustrate, an IDS with insufficient sensitivity may fail to detect meaningful anomalies, while one that is overly sensitive may generate numerous alerts for normal events along with intrusions. It would be unfeasible for administrators of large systems to distinguish between intrusions and non-hostile events [CBK09].

1.2 Aim of the Work

In order to address the above-mentioned challenges, this work presents the "Configuration-Engine" (CE) - a method for the automatic configuration of AD tools. The CE is a process that addresses the configuration problem for AD tools and generates configurations from the data it received as input for AD tools. In simple terms, it assesses what aspects of the data are worth investigating and passes this information to the detection system.

The main goal of this thesis is therefore the definition of the underlying process of the CE. The CE itself consists of a collection of configuration methods designed to effectively recognize various patterns in the data that represent some kind of learnable normal behavior. The further objective of this work is the definition and evaluation of such methods in order to show the effective applicability and usability of the CE through empirical testing on log data.

In summary, the following research question and associated sub-questions are stated:

1. To what extent can automated configuration methods improve the effectiveness of anomaly detection tools in identifying intrusions compared to manually created configurations, with respect to detection metrics such as precision and recall in audit and Apache log data?

    a) To what extent is it possible to use feedback from an anomaly detection tool to effectively optimize configurations regarding precision and recall with the restriction of using only (presumably) anomaly-free data?

    b) How similar are artificial configurations compared to manually created configurations with respect to the Jaccard similarity coefficient?

c) How similar are artificial configurations generated for different datasets collected from similar infrastructures compared to each other with respect to the Jaccard similarity coefficient?

Research question 1 is answered in Chapter 6. Questions 1a, 1b and 1c are indirectly answered in Chapter 5.

1.3 Scope and Limitations

The general focus of this work primarily emphasizes the data science aspect of attack detection, specifically focusing on statistical data analysis, rather than the related cybersecurity aspects which provide motivation for the undertaken research.

This work is limited to AD tools that take log data and a configuration (file) as input and output a report that lists all the anomalies found and, in some cases, what caused the anomaly.

Six different detectors of the AMiner [LWS+23] were chosen to show the applicability of the CE. Each of the detectors employs a distinct method in order to cover a wide range of the data's properties. Covering all available detectors of the AMiner would be beyond the scope of this thesis but, in theory, many of the used techniques are applicable to other detectors as well since they often address similar data characteristics but with a different approach.

The analyzed data was limited to Apache access and audit log data but the potential applicability of the CE goes beyond these types of data and can theoretically be applied successfully to any kind of event data for which detection of anomalies is reasonable. Thereby, a parser has to be applied that converts the data to a suitable format. For the used Apache access and audit log data the predefined parsers of the AMiner were used [LWS+23]. For new types of data where no parsers are already available one would have to create a new parser manually or semi-automatically through parser-generating algorithms [WLSK19, HZH+18].

As mentioned earlier, the logs were not generated by human users, but by a simulation. On the one hand, this is convenient as labelled log data AD datasets are very rare, but at the same time there may be patterns or properties of the data that differ from regular human behaviour. However, since the methods defined in this thesis should be generally applicable, we assume that the data from [LSW+20, LSF+22] is sufficient to demonstrate the usability of these methods.

From an implementation point of view another limitation is posed by the amount of data that can be processed at once. As some of the methods perform tedious statistical operations on the training data, the computational complexity strongly scales with the number of features and instances, limiting the total amount of training data one can analyze. The further implications of this limitation are described in Sec. 3.1.

1.4 Structure of the Work

Chapter 2 provides an introduction to AD, encompassing both a broad overview of the field and a detailed examination of the existing literature on the related topics that this work builds upon.

In Chapter 3 we prepare and explore the data. The character of the data and the requirements to the input parameters of the detectors motivate the parameter selection methods also described in this chapter. These methods represent the core components of the model. Their composition determines how the configurations for the detectors are constructed.

In Chapter 4 the general configuration approach is explained. We also describe the specific composition of the configuration process for each detector, consisting of the configuration methods explained in the previous chapter. Furthermore, we present a method for optimizing configurations retrospectively based on direct feedback from the detection tool and provide a step-by-step example to further clarify the process.

The evaluation of the whole process is carried out in Chapter 5. At first, we fine tune the hyperparameters of the model with the validation datasets in order to obtain the best possible setting for all datasets. With this setting we assess the performance and similarity of artificial configurations compared to manually created ones from domain experts. Additionally, the effectiveness of the optimization approach is evaluated by comparing the performance of configurations before and after the optimization.

The work is concluded in Chapter 6, where we also answer research question 1 and provide a future outlook.

CHAPTER 2

Literature Review

2.1 Anomaly Detection Overview

Anomaly detection is a critical task in data mining and machine learning aimed at identifying patterns that differ significantly from the expected behavior within a dataset that indicate potential fraudulent activities, system malfunctions or other phenomena worth investigating. Anomaly detection techniques include a variety of methodologies, ranging from conventional statistical techniques to highly sophisticated machine learning algorithms.

As in other machine learning disciplines we differentiate between supervised, unsupervised and semi-supervised approaches. Chandola et al. [CBK09] determine the grade of supervision by the availability of labelled anomalies:

1. **Supervised** anomaly detection methods rely on labeled data to train models that distinguish between normal and anomalous instances, offering precise classification but requiring labeled anomalies which are often rarely available. The requirement for labels makes supervised anomaly detection algorithms difficult for practical applications as no information about the anomalies' distinct characteristics are available beforehand.

2. **Unsupervised** techniques operate on unlabeled data, detecting anomalies based solely on data structure, making them suitable for scenarios that lack labeled anomalies. Consequently, they do not require training data as they simply assume that normal instances are much more frequently occurring than anomalous ones. Commonly, techniques like clustering and outlier analysis are used. Unsupervised methods are flexible and can uncover hidden insights without human supervision. However, they may struggle with interpretability, predictability, and complex patterns in the data [GU16].

3. **Semi-supervised** techniques also use labels but only for the normal class. Hereby, the model is able to learn just the normal behavior of the data but no abnormal behavior. Anomalous instances are identified by their deviation from the normal class. Semi-supervised anomaly detection has a much wider range of applications than supervised approaches. For many semi-supervised approaches it is possible to transform them into an unsupervised approach. Thereby, one uses a sample of the unlabelled data set as training data, assuming it represents normal behavior and that the test data contains a minimal number of anomalies.

Each method comes with a set of advantages and disadvantages and is chosen based on factors like data characteristics, anomaly nature and necessary trade-offs between performance and scalability. These factors are often dependent on the domain of application such as cybersecurity, finance, industrial systems, healthcare and many more [CBK09].

The challenges of anomaly detection include the interpretation of detected anomalies, the handling of imbalanced datasets and the adaptability of anomaly detection systems to dynamic environments. Extensive research is carried out in order to enable more robust and effective anomaly detection solutions tailored to diverse and novel applications and problems [CBK09].

Goldstein and Uchida [GU16] distinguish three types of anomalies:

- **Point anomaly:** A point anomaly consists of a single instance that is considered anomalous. For instance, a single value in a list of values that is significantly higher than the others may be considered as a point anomaly.
- **Collective anomaly:** A collective anomaly consists of a group of related data instances that deviates from the overall dataset, even though individual instances within the group may not be considered anomalous on their own. Anomalies of this kind are identified by observing relationships or patterns among multiple data points.
- **Contextual anomaly:** A contextual anomaly occurs when the behavior of a data point varies depending on its context. For example, a sudden raise of sales of a product right before Christmas may be seen as normal while it would be seen as abnormal compared to other days of the year.

Different detection methods address different anomaly types and hence require different approaches in evaluation. For this work point and collective anomalies are considered.

A comprehensive overview over system log analysis for anomaly detection algorithms is provided by He et al. in [HZHL16]. They differentiate between four main steps for anomaly detection on log data:

1. **Log collection**: Large-scale systems generate logs containing valuable information about system states and runtime events.

2. **Log parsing**: Unstructured log messages are parsed into event templates with constant parts and variable parameters.

3. **Feature extraction**: Extract valuable information from the log events as input for the anomaly detection algorithm.

4. **Anomaly detection**: The extracted feature vectors are passed to the anomaly detection algorithm to identify anomalies.

Additionally, the paper reviews and evaluates a set of state-of-the-art log-based anomaly detection methods. Amongst others, the methods include three of the most common unsupervised approaches - Log Clustering, Principal Component Analysis (PCA), and Invariant Mining - that all exhibit different advantages and disadvantages depending on the data.

One of the most prominent anomaly detection approaches for log data is provided by Du et al. in [DLZS17] where they propose DeepLog, a recurrent neural network (RNN) model that uses Long Short-Term Memory (LSTM) to detect anomalies in system logs. It models log data as a natural language sequence, allowing DeepLog to automatically learn log patterns. It has been shown to achieve high detection accuracy, particularly on the HDFS and BGL datasets. DeepLogs popularity comes from the fact that it was the first to detect sequential anomalies in log data using deep learning [LOSW23]. Similar approaches to DeepLog are LogAnomaly [MLZ+19], which also relies on LSTM RNN and LogRobust [ZXL+19], which focuses on robustness against unstable log data.

However, neural networks typically require structure and entail complex dependencies. Consequently, preparing log data for neural network ingestion and extracting relevant features is a challenging task. Furthermore, the variety of deep learning architectures, such as recurrent or convolutional neural networks, complicates model selection for specific use cases [LOSW23].

Another well-known anomaly detection technique is provided by Xu et al. [XHF+09]. They utilize PCA to transform message count vectors into subspaces where anomalies are detected if they are significantly distant from other samples. On the other hand, Lou et al. [LFY+10] employ Invariant Mining to identify anomalies by detecting event sequences that violate established linear relationships among log events.

The most common evaluation metrics for anomaly detection are precision, recall, false positive rate and F1-score which is the combination of precision and recall. Less common metrics include the accuracy, as it does not account for class imbalance in datasets and the area under the (precision-recall) curve (AUC), as well as the receiver operator characteristic (ROC) [LOSW23]. Throughout this work, mostly precision and recall are used but also the F1-score:

$$\text{Precision} = \frac{TP}{TP + FP}, \qquad \text{Recall} = \frac{TP}{TP + FN}, \tag{2.1}$$

$$\mathrm{F1} = 2 \cdot \frac{\mathrm{Precision} \cdot \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}}, \tag{2.2}$$

where TP is the number of true positives, FP the number of false positives and FN the number of false negatives.

2.2 Related Work

The foundation of this work is provided by Landauer et al. in [LWS+23]. This paper describes the so-called Logdata-Anomaly-Miner - in short, AMiner - a modular pipeline for intrusion detection on log data. The implementation of the AMiner[1] is used for the evaluation of this work. This anomaly detection tool employs a collection of algorithms used to detect anomalies in log data. Each of these detectors addresses different data characteristics with a variety of techniques such as text processing, time series analysis, association rule based approaches and more. The AMiner operates in two different states: learn mode and detection mode. In learn mode each detector learns in its own way by extracting relevant information from the log data, mostly by building frequency tables or associated arrays. The AMiner learns incrementally by updating the gathered information after each processed log line. In detection mode the AMiner outputs information about log lines whose behavior differ from the learned norm by exceeding certain thresholds or by detecting something unprecedented, respectively. For both states it is necessary to define a configuration that guides the detectors in learning, dictating what aspects of the data are worth investigating and how they should be investigated.

The AMiner is part of the Automatic Event Correlation for Incident Detection toolbox (AECID) which was developed by the Austrian Institute of Technology (AIT). This toolbox features a variety of applications for log-related issues with its core focus on log based intrusion detection [WSSF18].

The AMiner takes parameters extracted from log data as input to analyze the system's behavior forensically. Its efficiency comes from its incremental approach to data analysis, whereby each line is processed one after another. This allows for the possibility of running the AMiner in real-time, provided that the processing speed is sufficient to keep up with the rate at which the logs are generated. The relevant information of the log lines is obtained by a log parser. Parsing log data is not straightforward due to the diverse array of formats for logs, ranging from conventional key-value pairs or human readable text to JSON structures. This variety demands sophisticated log parsing mechanisms to effectively classify events and extract relevant information. Parser trees outperform traditional regular expression lists in runtime performance by processing each token in an event only once [LWS+23]. All standard parsers of the AMiner utilize tree structures. For the case that a standard parser does not fit the data, the AECID toolbox provides a parser generator for creating tree-based parsers from log samples [WLSK19].

[1] AMiner GitHub page `https://github.com/ait-aecid/logdata-anomaly-miner`; accessed 13-May-2024.

Anomaly detection in categorical data has received relatively little attention compared to quantitative data due to its challenging nature. Traditional techniques often involve identifying a representative pattern and measuring distances from this pattern to detect anomalies. However, this approach is difficult to apply to categorical data as identifying patterns and measuring distances in such data is more complex than in quantitative data [TH19]. In this work 5 out of the 6 chosen detectors operate on categorical data. One detector analyzes the frequency of the events.

The configuration methods described here, as well as the way mathematical definitions are handled, take inspiration from [LHW+21] - for instance, the usage of co-occurrence for variable combinations (see Chapter 3). It proposes the so-called Variable Correlation Detector (VCD) which is also present in the AECID toolbox and implemented for the AMiner. The process they describe in [LHW+21] utilizes a sequence of selection and filtering steps to disclose variable pairs with correlating values to detect anomalies. Through its complexity and its large number of adjustable parameters the VCD is not part of the the selection of detectors used in this work. It would be out of scope to automate more than only this detector's configuration process, yet other detectors, such as EntropyDetector and NewMatchPathValueComboDetector [LWS+23], are also important as they allow very effective detection of anomalies when configured accordingly. Still, the VCD would be an appropriate candidate for future research. Nevertheless, the process proposed in this thesis also includes selection and filtering steps, as the VCD, that prune the search space beforehand to improve both efficiency and efficacy of the core methods. Unlike other association rule techniques that utilize frequent itemset mining, the VCD does not omit infrequent variables which possibly hold valuable information for the detection of anomalies [LHW+21].

In general, the majority of association rule mining techniques are not well-suited for anomaly detection with log data, as they require a standardized format. Thus, they face the same problem as neural networks [LOSW23]. Additionally, the sheer volume and variability of log data in large-scale systems further complicate the application of association rule algorithms, as they may struggle to efficiently process vast and diverse datasets which is often the case with log data [Vaa03].

Wurzenberger et al. [WHLS24] describe the importance of the variable part of log lines. Many detection techniques solely focus on event type occurrences which are usually strongly structured and completely neglect the message part of a log line which is typically unstructured due to the absence of standardization. In this paper they therefore propose an unsupervised approach for analyzing the variable part of log lines which they call the variable type detector (VTD). This technique classifies the tokenized log line, also called variables, as different data types such as chronological, static, ascending, descending, continuous, unique or more. Anomalies are identified by detecting a significant change in the data type of a variable. The idea of classifying variables into types such as static or unique was taken from this paper.

One of the configuration methods of Sec. 3.4 utilizes similar techniques as the ones featured in [ESL23]. The paper reviews and evaluates six window size selection (WSS)

algorithms on different tasks in unsupervised time series data mining (TSDM). The authors highlight the importance of WSS in TSDM, as it is a crucial hyperparameter that can significantly impact the results. They categorize WSS strategies into two types: whole series based methods, such as Fourier transformations or autocorrelation methods, as well as subsequence based methods which compare local window statistics with global signal properties.

Another work worth mentioning is [ATD19] which proposes a novel method for adapting the parameters of anomaly detection algorithms in order to fit the model to changes in the data's behavior over time, also known as concept drift. The models are trained in an unsupervised way and therefore assume that the amount of true anomalies in the data is negligible for training, thus anomalies are detected as outliers. The detection of anomalies through the model depends on parameter settings that fit to the current data. The model's predictions are evaluated by a validation dataset. The parameters are then adjusted to new data based on the resulting performance metrics precision and recall. This approach exhibits similarities to the optimization approach, one of the steps of the Configuration-Engine described in Section 4.1.4. Contrary to the approach of [ATD19], this optimization takes a more semi-supervised perspective as we assume the data to be totally anomaly-free. Thus the approach cannot work with metrics like precision and recall for adjusting thresholds as true positives cannot exist. The adjustment of parameters here is thus solely based on the assessment of the amount of produced false positives.

In general, many parameter optimization techniques are based on evolutionary algorithms:

1. Generate an initial population of candidate solutions.
2. Evaluate the fitness of each candidate.
3. Select parents based on fitness.
4. Apply variation operators, such as crossover or mutation, to produce offspring.
5. Replace least fit individuals with the new offspring.

These steps are repeated until the termination criteria are met [BS93]. However, the assessment of fitness is not straightforward for the case of anomaly-free data. As mentioned above there are no true positives and minimizing false positives will always result in the trivial case of the minimum being zero. For the same reason it is not possible to formulate this as a mathematical optimization problem and therefore not meaningful to use common search methods such as GridSearch or LocalSearch. The optimization approach of Sec. 4.1.4 therefore takes a different approach.

The proposed methods in Section 3.4 are mostly feature selection methods. Feature selection is a common problem in anomaly detection as it is critical to select input features for the anomaly detection algorithm with a high sensitivity to anomalies, in

order to achieve high effectivity [KBD+08]. For instance, Kloft et al. [KBD+08] propose a method for automatic feature selection for network intrusion detection that determines optimal mixture coefficients for various sets of features by utilizing a generalized support vector data description [TD04] model. The problem is transformed into an optimization problem, whereby the solution is given as the minimal-volume description of the data. On the other hand, Pascoal et al. [PdOV+12] propose a feature selection method based on a mutual information metric. The features are divided into two subsets by highest and lowest mutual information. They then search for the partition that maximizes the t-test statistic (as a measure of separation) among subsets. They utilize robust statistics in order to handle the eventually contaminated training data (by true anomalies), thereby making also their algorithm robust. Since in this work we pursue a semi-supervised approach robustness of our model is not directly necessary. However, as even an anomaly-free dataset can contain atypical behavior of any kind it is still useful. Also, most of the methods in Sec. 3.4 apply some kinds of averaging operations and therefore exhibit some level of robustness. Both [KBD+08] and [PdOV+12] utilize interesting approaches, yet they are not applicable here due to the specific nature of the detectors that were chosen for the application of the CE.

Most publications assess their approaches using well-known datasets such as HDFS, BGL, Thunderbird, OpenStack or Hadoop which are publicly available on LogHub[2] [HZHL20, XHF+09]. LogHub maintains a collection of log datasets for machine learning driven research. These datasets became the standard evaluation playground for many anomaly detection algorithms in literature, yet most of their anomalies can be solely detected by analysis of event type sequences and do not require advanced techniques for detection. An event type is a predefined template that only fits the events of the defined type. Thereby, the parameters of the events itself are ignored [LSW23]. Nevertheless, these parameters, or features, hold valuable information with which effective anomaly detection is possible [KV03]. The AMiner detectors, we are mostly interested in, such as the EntropyDetector or the NewMatchPathValueComboDetector, operate on these features that are extracted from the event [LWS+23]. More suitable datasets for our case are AIT Log Data Set V1.0 [LSW+20] and AIT Log Data Set V2.0 [LSF+22] as their anomalies can often only be detected by an analysis of the event parameters. These datasets are therefore used for training, validating and testing the implementation of the Configuration-Engine. They are described more closely in Section 3.1.

[2]LogHub GitHub page `https://github.com/logpai/loghub`; accessed 29-May-2024.

CHAPTER 3

Data Analysis

3.1 Data

Log data represents a record of all occurrences within a system, application, or network device. When logging functionality is activated, the system automatically generates these logs. Each entry is marked with a precise timestamp and encompasses a variety of information:

- **User activity**: Information about user interactions and logins and actions performed within an application or system.
- **System performance**: Records regarding system startups, shutdowns, hardware failures and currently used resources.
- **Security events**: Documentation of login attempts, changes in access control, authentication and alerts triggered by built-in intrusion detection systems.
- **Errors and warnings**: Logs capturing errors, exceptions, warnings or debug data for the diagnosis and resolution of issues.
- **Network traffic**: Details concerning network access, connections and traffic flow.
- **Database transactions**: Information regarding data modifications or queries.
- **Application-specific events**: Events specific to an application.

For the evaluation in Chapter 5 we use audit and Apache Access log data. Apache Access logs record all requests made to the Apache web server. These logs include information about every request processed by the server. Audit logs, on the other hand, are more

comprehensive and focus on tracking system activities and user actions within a broader scope and are used primarily for security and compliance purposes.

A sample audit log line is provided in Listing 3.1. The information is represented as key-value pairs with mostly categorical values and the timestamp given in unix time ("1642760221.565").

Listing 3.1: Single audit log line.

```
type=USER_START msg=audit(1642760221.565:662): pid=14286 uid=0 auid=0 ses=98
    msg='op=PAM:session_open acct="root" exe="/usr/sbin/cron" hostname=?
    addr=? terminal=cron res=success'
```

Labelled log data is very rare and hard to find which poses a common problem for the evaluation of AD algorithms [LSW+20, LWS+23, LSF+22, CBK09]. The datasets used for training, validating, and testing the CE originate from the AIT Log Data Set V1.0 [LSW+20] and AIT Log Data Set V2.0 [LSF+22]. These datasets were generated synthetically due to the limited availability of labeled intrusion detection datasets. The scarcity of labeled log data for AD arises from the complexity of data sensitivity regarding legal and ethical considerations, resource intensiveness and unavailability of ground truth. The logs were sourced from diverse testbeds that were built at the Austrian Institute of Technology (AIT), each representing a small enterprise network. Simulated user behavior was generated over a duration of multiple days to introduce noise and at some points attacks were launched against the networks. A variety of log types were captured, though only the Apache access and audit logs are used for this project.

While the training of the AD tool has to be done with anomaly-free data, the validation and test sets require anomalies. Additionally, we are dealing with event data, thus time series data and have to maintain the temporal order of the original data. Due to these constraints, it is not meaningful to use classic k-fold cross-validation. Given the constraints, the test set starts at the exact time or log event when the first attack occurs in the data and ends with the last entry in the dataset. The entries until the attack constitute the training set.

One disadvantage of this approach is that the training and test sets remain constant within a given dataset which may result in overfitting. To address this, multiple datasets and log data types are employed for validation to increase model generalization. A selection of them was used for the creation of the model while the rest remained untouched until the final evaluation for pure testing to avoid data leakage.

Table 3.1 lists the datasets used for validation and testing and show some relevant numbers. The datasets, "russellmitchell", "fox" and "harrison" and "mail.onion.com" for Apache, were used during the validation phase where the model and its method were defined and implemented and are therefore not used for the performance assessment in Chapter 5, except for the hyperparameter tuning.

One can see, that from the datasets "mail.spiral.com", "mail.onion.com", "mail.insect.com", "mail.cup.com" only their Apache Access data is used. Given that their audit log data

Name	Training samples	Test samples	Point anomalies	Collective anomalies	V/T
Apache Access datasets					
russellmitchell	2884	8300	7696	7	V
fox	9058	413948	410841	16	V
harrison	19604	420790	415376	352	V
mail.onion.com	53004	28959	6429	19	V
shaw	8050	7696	5226	6	T
santos	6752	9032	7794	7	T
wardbeck	32454	9647	5299	9	T
wheeler	7848	433072	431560	53	T
wilson	25130	438743	428116	89	T
mail.spiral.com	65811	34634	7370	30	T
mail.insect.com	118549	50791	6973	30	T
mail.cup.com	115443	33091	6789	28	T
Audit datasets					
russellmitchell	1859	457	9	2	V
fox	2078	809	19	3	V
harrison	2454	376	24	3	V
shaw	2608	787	19	3	T
santos	1968	295	19	3	T
wardbeck	2645	274	19	3	T
wheeler	2693	148	14	3	T
wilson	2622	851	22	3	T

Table 3.1: Dataset statistics. “V/T” indicates whether the dataset was used for validation (V) or testing (T).

has a high scan volume they are simply too large for the CE. To illustrate, the smallest audit log file is from “mail.spiral.com” and is 11GB in size and contains over 350 million log lines. The information density is extremely low compared to the other datasets. Therefore, it is not meaningful to just use a portion of the data as not enough information can be extracted from a feasibly sized batch of the data to learn a useful normal model. All methods utilize all available log lines to extract information and apply transformations. The methods’ operations process information from all log lines simultaneously, which is not problematic for the other, smaller datasets, but is problematic in terms of RAM for large datasets. This poses a limitation to the CE. One potential solution would be to stream the data in a line-by-line or batch-wise , but that would entail a significant expense regarding the complexity of the implementation and the methods. As the implementation is not the focus of this work this was not further investigated.

3.2 Parsing

Before we can start analyzing the data a parser has to be applied to sort or extract the information from the logs into a set of features which we also call variables. The used log parser encompasses a tree-based method from [WLSK19]. In brief, this parser extracts the information of each of the log lines into a set of different variables and assigns names to each.

After applying the audit log parser of the AMiner [LWS+23], the log line from Listing 3.1 can be represented as an associative array (or simply a dictionary) with the variable names as keys.

Listing 3.2: Parsed log line.

```
{'/model': 'type=USER_START msg=audit(1642760221.565:662): pid=14286 uid=0
    auid=0 ses=98 msg=\'op=PAM:session_open acct="root" exe="/usr/sbin/cron"
    hostname=? addr=? terminal=cron res=success\'',
 '/model/type_str': 'type=',
 '/model/type/user_start': 'USER_START msg=audit(1642760221.565:662):
    pid=14286 uid=0 auid=0 ses=98 msg=\'op=PAM:session_open acct="root"
    exe="/usr/sbin/cron" hostname=? addr=? terminal=cron res=success\'',
 '/model/type/user_start/msg1_str': 'USER_START msg=',
 '/model/type/user_start/audit_str': 'audit(',
 '/model/type/user_start/time': '1642760221.565',
 '/model/type/user_start/colon_str': ':',
 '/model/type/user_start/id': '662',
 '/model/type/user_start/pid_str': '): pid=',
 '/model/type/user_start/pid': '14286',
 '/model/type/user_start/uid_str': ' uid=',
 '/model/type/user_start/uid': '0',
 '/model/type/user_start/auid_str': ' auid=',
 '/model/type/user_start/auid': '0',
 '/model/type/user_start/ses_str': ' ses=',
 '/model/type/user_start/ses': '98',
 '/model/type/user_start/msg2_str': ' msg=',
 '/model/type/user_start/msg2': "'op=PAM:session_open",
 '/model/type/user_start/fm/acct': ' acct="root"',
 '/model/type/user_start/opt': ' exe="/usr/sbin/cron" hostname=? addr=?',
 '/model/type/user_start/terminal_str': ' terminal=',
 '/model/type/user_start/terminal': 'cron',
 '/model/type/user_start/res_str': ' res=',
 '/model/type/user_start/res': "success'",
 '/model/type/user_start/opt/opt_seq': ' exe="/usr/sbin/cron" hostname=?
    addr=?',
 '/model/type/user_start/opt/opt_seq/exe_str': ' exe=',
 '/model/type/user_start/opt/opt_seq/exe': '"/usr/sbin/cron"',
 '/model/type/user_start/opt/opt_seq/hostname_str': ' hostname=',
 '/model/type/user_start/opt/opt_seq/hostname': '?',
 '/model/type/user_start/opt/opt_seq/addr_str': ' addr=',
 '/model/type/user_start/opt/opt_seq/addr': '?',
 '/model/type/user_start/fm/acct/acct_str': ' acct=',
 '/model/type/user_start/fm/acct/acct': '"root"'}
```

In the resulting dictionary of Listing 3.2 one can see how the variable names have a tree or path-like structure. Subsequently, this dictionary is transformed into a table where each row represents an event and each column corresponds to a variable along with its associated values. Thus, each value is assigned to a particular variable. In this format the data can be easily analyzed for further purposes. Note, that the table might be extremely sparse since an event is a composition of variables that do not necessarily occur in every event.

3.3 Detector Requirements

For the application a set of detectors of the AMiner is chosen that addresses a variety of different properties in the data.

In-detail information about the detector's specific settings is listed in the AMiner documentation[1]. We describe the selected detectors, their operating principles, the data characteristics they address and what parameters they require in the listing below. The detectors all require log data as input and, unless stated otherwise, require individual variables as input parameter. Some also require special input parameters [LWS+23].

1. **NewMatchPathValueDetector (NMPVD):** The detector triggers an alert whenever new and unknown (untrained) values of a variable are found. For both training and test phase it is therefore meaningful to present variables to the detector that contain a limited set of categorical values.

   - **Input**: event data, individual variables.
   - **Output**: anomalous events.

2. **NewMatchPathValueComboDetector (NMPVCD):** This detector is similar to the previous one, but examines combinations of variables instead of single variables. An alert is triggered whenever a new and unknown value combination for the corresponding variable combination is found. Therefore, one would want to pass variable combinations to the detector that form a limited set of value combinations.

   Especially from audit data the parser generates a large number of variables (usually over 300). As the number of possible variable combinations does no scale linearly it is important to consider the computational cost of the configuration process.

   - **Input**: event data, variable combinations.
   - **Output**: anomalous events.

3. **CharsetDetector (CSD):** The CharsetDetector inspects the set of characters of the values of a variable. During training this character set is extended whenever a

[1] AMiner documentation `https://aeciddocs.ait.ac.at/logdata-anomaly-miner/development/CONFIGURATION.html`; accessed 29-April 2024.

new character is found in a value of the given variable, while during testing the detector will trigger an alert. Consequently, the usage of variables with limited character sets seems meaningful.

- **Input**: event data, individual variables.
- **Output**: anomalous events.

4. **EntropyDetector (ED):** The detector learns the occurrence probabilities of consecutive character pairs of a variable. Averaging over all character pair probabilities of a value yields a critical value which is a measure of the likelihood of a value's occurrence. When learning is turned off the detector will trigger an alert whenever the likelihood of the occurrence of a value is below a chosen limit. Consequently, the behaviour of the critical values of variables in the training data is the important factor to consider.

   - **Input**: event data, individual variables, a lower limit for critical values.
   - **Output**: anomalous events, the critical values of the anomalous events.

5. **ValueRangeDetector (VRD):** The ValueRangeDetector generates ranges for numeric values, detecting values outside these ranges and extending ranges when it is learning. Variables with numeric values and limited ranges are therefore a beneficial choice for this detector.

   - **Input**: event data, individual variables.
   - **Output**: anomalous events.

6. **EventFrequencyDetector (EFD):** This detector assesses the occurrence of a variable (or its specific values) within a time window of a certain window size, thus the frequency f. Log lines are classified as anomalies whenever the frequency in the current time window exceeds or falls below a certain limit. The expected frequency f_{exp} within a time window is computed from the previous time windows in the training phase. The season is the periodicity of f. The detector's most important input parameters are the seasonality of the data, the number of time windows that should be considered for the computation of f_{exp}, the window size and a confidence factor controlling the allowed deviation from f_{exp}. Concluding the above, it is thus necessary to analyze the behavior of the occurrences of a variable over time.

   For this work the number of time windows is not investigated. It only makes sense to limit this value if the behavior of the data changes over time, which is not the case here as the datasets' records do not cover more than a few days. Additionally, the confidence factor is sufficiently generic to not necessitate significant changes based on the given data, as the confidence value of the detected anomalies is in every observed case very high (> 0.9) for both true positives (TP) and false positives (FP). Besides, the sensitivity can in some way already be controlled by the window size. Both the number of time windows and the confidence factor can therefore be fixed to a constant value.

- **Input**: event data; individual variables; seasonality of the data; window size; number of windows; a confidence factor that defines the range of tolerable deviation of measured frequency from expected frequency.
- **Output**: anomalous events, confidence value of anomaly.

The detectors listed above and their requirement for input parameters provide the motivation for the definition of a set of configuration methods. These methods should determine the detectors' effective parameter settings based on the training data.

Note, that all the detectors analyze the properties of values of variables except the EFD which is solely interested in their occurrence frequencies over time. For all detectors but the EFD it is therefore not meaningful to treat the data as time series data.

3.4 Parameter Selection Methods

This section features a selection of methods used to classify the characteristics of variables and determine other parameters that are necessary for the configuration of the chosen detectors. In other words, the methods transform information extracted from the data into the input parameters for the detectors. These methods represent the model's core components. Their composition determines how the configuration of a detector is constructed.

In general, an AD tool employs one or more detection algorithms. Since each detector requires different input parameters it has to be assessed individually for each method which parameter values are suitable. In many cases, a human operator with in-depth knowledge of the event data has to determine these parameters. The first step in finding the right parameters is often to assess which variables to choose. This configuration step can be automated by mapping the variable properties to the corresponding detection method. Consequently, one has to classify the variables into certain feature sets that suit the corresponding detector.

Some of the configuration methods explained in the sections below require thresholds or other parameters. The selection of suitable values for these thresholds falls within the scope of hyperparameter tuning and can be done by an educational guess or empirically by testing with different datasets or -splits, respectively. Search algorithms such as GridSearch or LocalSearch can theoretically also be applied. A thorough investigation of the hyperparameters is given in Section 5.4.

To effectively use mathematical formulations in this context a set of expressions is introduced in Table 3.2.

Note, that in this chapter the term "testing with data" or likewise refers to testing with the validation datasets - see Table 3.1.

Expression	Description
$\mathcal{V}$	the set of all variables (or simply the dataset)
$n_{\mathcal{V}}$	the total number of events (or the length of the dataset)
V	a set of variables for which applies $V \subseteq \mathcal{V}$
x, y, z	variables of $\mathcal{V}$
x_i	the value of the i-th occurrence of x, thus $x_i \in x$
$dim(u) : \mathbb{R}^n \to \mathbb{N}$	the length of a vector u of length n
$dim(x) : V \to \mathbb{N}$	total count of occurrences of x
$ValCount(x_i) : x \to \mathbb{N}$	total count of all occurrences of values $x_j \in x$ equal to x_i
$UniqueCount(x) : V \to \mathbb{N}$	total count of unique occurrences of x
$UniqueCount(x_i) : x \to \mathbb{N}$	total count of unique occurrences $\forall x_j$ with $j = 0, 1, ..., i$
$CharSet(x_i)$	set of all unique characters $\forall x_j$ with $j = 0, 1, ..., i$
$min(x_i) : V_{numeric} \to \mathbb{R}$	minimum value $\forall x_j$ with $j = 0, 1, ..., i$ for numeric variables in $V_{numeric}$
$max(x_i) : V_{numeric} \to \mathbb{R}$	maximum value $\forall x_j$ with $j = 0, 1, ..., i$ for numeric variables in $V_{numeric}$

Table 3.2: Definitions of mathematical expressions.

3.4.1 Static Occurrence

Variables that have the same value in almost every sample in which they occur are classified as "static" [WHLS24]. In other words, a variable is static if the number of unique occurrences is equal to 1. The set of static variables is defined as

$$V_{static} \coloneqq \{x \in \mathcal{V} \mid UniqueCount(x) = 1\}. \tag{3.1}$$

3.4.2 Random Occurrence

Variables are classified as "random" if there are different values for most occurrences. Since it is possible that individual values of a variable are occurring randomly, the occurrences of each unique value of a variable are counted. If this number is below a certain threshold the variable is considered as random:

$$V_{random} \coloneqq \{x \in \mathcal{V} \mid \exists x_i \in x : ValCount(x_i) < \theta\}. \tag{3.2}$$

It is meaningful to choose threshold θ as a number higher than 1 since a value occurring more than once indicates that it is not a random value. It would therefore be appropriate to set $\theta = 2$. In practice, it is possible that some variables occur with the same value in batches, meaning that the same values occur twice or more within a certain period and then never again. However, throughout this work θ is fixed to 2.

Randomness exhibits a very limiting condition since a variable with a value with only a single occurrence is already classified as random, even if the remaining values are

identical. One could therefore exchange randomness by non-stability (by occurrence) which exhibits a higher tolerance for outliers - see next section.

3.4.3 Stability

In general, the stability of a variable can be considered from multiple points of view. It is depending on the specific characteristic of the variable one is considering. We call a variable "stable" regarding that characteristic if the corresponding curve approaches a constant value within the training period. Figure 3.1 exemplarily shows the behavior of different values regarding the number of unique occurrences against the number of occurrences. One can see, the random variable (blue) has a new unique occurrence for every occurrence while the static one (orange) only has a single unique occurrence and is therefore constant. The line between them (green) shows the behavior of a variable that could be classified as stable as no new unique values occur at some point. Consequently, static variables are a subset of the stable variables. For many detectors there is some kind of stability involved since it implies some kind of learnable behavior for many detectors.

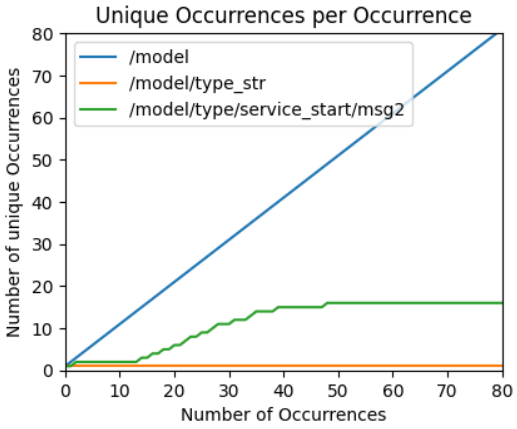


Figure 3.1: Unique occurrences per occurrence of a static (orange), stable (green) and random variable (blue).

To check whether a variable is stable, some kind of threshold curve is defined that acts as an upper limit for the curve $f(k)$ that represents the data characteristic we are interested in. $k(x) = k$ is the number of occurrences of a variable x with $k \in \mathbb{N}$. This threshold curve is not applied directly to $f(k)$, but to its derivative $f'(k)$. The derivative represents the change in $f(k)$ per occurrence. For a stable variable, this curve should therefore approach zero within the period of the training data.

For the detectors covered in this work, we are actually not interested in the magnitude of change but rather in the information whether a change has occurred or not. Consequently, we want $f'(k) \in \mathbb{R}$ to be an element of the binary space $\{0, 1\}$. Therefore, we define the

boolean conversion

$$g_b(k) = \begin{cases} 1 & \text{if } g(k) \neq 0, \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

for an arbitrary function $g(k)$. Thus, the function $f'_b(k)$ is 1 (or `True`) if a change in $f(k)$ occurred at occurrence k or 0 (or `False`) if no change occurred.

The concept of stability is based on the assessment of the mean values of the segments $s_m(f'_b(k))$ with $m = 0, 1, ..., n_s - 1$. n_s is the number of segments. To be precise, the m-th segment of the function $f'_b(k)$ is

$$s_m(f'_b(k)) = \begin{cases} f'_b(k) & \text{for } \frac{m}{n_s} dim(x) \leq k < \frac{m+1}{n_s} dim(x) \\ 0 & \text{otherwise.} \end{cases} \tag{3.4}$$

The set of stable variables is then defined as

$$V_{stable} := \left\{ x \in \mathcal{V} \mid \frac{\sum_l s_{ml}(f'_b(k))}{L_m} \leq \theta_m \quad \forall m \in \{0, 1, ..., n_s - 1\} \right\} \tag{3.5}$$

where each element of s_m is denoted as s_{ml} with $l = 0, 1, ..., L_m$. $L_m = dim(s_m)$ is the length of each segment m. L_m is not uniform if $dim(x)$ is not divisible by n_s. Therefore,

$$L_m = \begin{cases} q + 1 & \text{if } m + 1 \leq r \\ q & \text{if } m + 1 > r \end{cases} \tag{3.6}$$

with quotient $q = dim(x) : n_s$ and remainder $r = dim(x) \mod n_s$. This definition is based on the function "array_split"[2] from the NumPy library [HMvdW+20].

If each of the segment means is below the thresholds θ_m then the corresponding variable is classified as stable. The thresholds θ_m represent a discretized threshold curve that serves as an upper boundary for the change in each segment of $f(k)$. As $f'_b(k) \in \{0, 1\}$ this relation can be understood as the "relative change per segment". This is also convenient for the selection of the thresholds as we can define them in a relative way within the range $[0, 1]$. This implies that thresholds chosen suitably for one dataset are likely to be suitable for other datasets - in theory, also for other data types.

To illustrate this with an example, we choose $f = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 4]$ and $\theta = [1, 0.5, 0.1]$ for some variable x. Consequently, we have $f' = f'_b = [1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]$, thus $dim(x) = dim(f') = 11$, $n_s = 3$. Therefore, $q = 3$, $r = 2$ and $L = [4, 4, 3]$. The segments themselves are then $s(f'_b) = [[1, 1, 0, 1], [0, 0, 1, 0], [0, 0, 0]]$ and their mean values $\overline{s}(f'_b) = [0.75, 0.25, 0.0]$. As $0.75 \leq 1$, $0.25 \leq 0.5$, $0.0 \leq 0.1$ are all true we classify variable x as stable.

[2] NumPy documentation of "array_split" function `https://numpy.org/doc/stable/reference/generated/numpy.array_split.html`; accessed 18-July-2024

The selection of the thresholds θ_m can be done in a manifold of ways. The curve described by the function $f'(k)$ should, in case of a stable variable, behave similar to an exponential decay, i.e. approach 0. It is therefore meaningful to say $\theta_m = e^{-cm}$ with some constant c determining the magnitude of decay. In general, any (discretized) function with similar behavior to an exponential decay (e.g. $1/x$) could be used for θ_m.

The paragraphs above described stability generically so that it can be applied to any function $f(k)$ representing the behavior of a certain data characteristic. In the following we describe the specific types of stability based on different data characteristics:

Stability by Unique Occurrence

The first characteristic we are interested in is the behavior of the number of unique values over the training period as in Figure 3.1. Therefore, we take

$$f(k) = UniqueCount(x_k) \tag{3.7}$$

and its discrete derivative

$$f'(k) = UniqueCount(x_k) - UniqueCount(x_{k-1}). \tag{3.8}$$

Hereby, we can assess whether the count of unique values of a variable is stable. For a variable classified as stable by occurrence, it can be assumed that it has a limited set of unique values. If the thresholds are reasonably set one can say that stability by occurrence is a weaker condition than static occurrence (Sec. 3.4.1) but a stronger condition than non-random occurrence (Sec. 3.4.2) - see Fig 3.1. Note, that the change of the count of unique values per occurrence can be at most 1 so that $f'(k) \in \{0, 1\}$ and therefore $f'(k) = f'_b(k)$.

Stability by Character Set

Detectors like the CharsetDetector of the AMiner create a character set from the characters of each value of a variable. Subsequently, new and unknown characters in values can be identified as anomalies. The stability of the length of this character set can therefore be an indicator for whether to pass a certain variable to a detector or not. In a similar manner to above, the function and its derivative read as

$$f(k) = dim(CharSet(x_k)), \tag{3.9}$$

$$f'(k) = dim(CharSet(x_k)) - dim(CharSet(x_{k-1})). \tag{3.10}$$

Stability by Value Range

For the ValueRangeDetector one would want to pass variables with a limited range of numeric values. Consequently, we define a measure of stability based on the minimum-maximum range for variables containing only numeric values $V_{numeric}$. We have

$$f_{min}(k) = min(x_k), \tag{3.11}$$

$$f_{max}(k) = max(x_k) \tag{3.12}$$

and their derivatives

$$f'_{min}(k) = min(x_k) - min(x_{k-1}), \tag{3.13}$$
$$f'_{min}(k) = max(x_k) - max(x_{k-1}) \tag{3.14}$$

with the minimum and maximum functions defined in Table 3.2. In order to fit the stability relation of Eq. 3.5 we can add both functions together to get

$$f'(k) = |f'_{min}(k)| + |f'_{max}(k)| \tag{3.15}$$

where $|\cdot|$ denotes the absolute value. Thus, this yields a function that is 0 when no change occurred in the minimum or maximum value of the variable and > 0 otherwise.

3.4.4 Co-occurrence

Some detectors require variable combinations as input such as the NMPVCD which raises an alert whenever a new combination of values for a specified combination of variables is found. The previous configuration methods assess the characteristics of each variable individually and subsequently, select the ones that fit the corresponding characteristic. One could say these are brute force search methods. Still, these methods are feasible in terms of computational effort since log lines are usually held in a limited length. Hence, the number of variables remains small enough for many methods. Since combinations do not scale linearly we have to take computational cost into account. From combinatorics we know that the number of combinations is given by

$$C_{count}(n, k) = \frac{n!}{k! \cdot (n-k)!}. \tag{3.16}$$

For example, for audit log data it is easily possible to receive around $n = 350$ variables from the parser for all log lines of a dataset. Note, that a single audit log line contains far less variables (around 30). Also, this number strongly depends on the parser itself. For combinations of length $k = 2$ there are $C_{count}(350, 2) = 61075$ combinations or for $k = 3$ there are already $C_{count}(350, 3) = 7145775$. To reduce the number of variables, we filter out irrelevant characteristics for the detector. The following types of variables can be filtered from $\mathcal{V}$, the set of all variables, such that only relevant variables remain:

- **Random variables (Eq. 3.2)**: They lead to random value combinations which means that in the most occurrences of a combination containing one or more random variables is a new value combination. It is therefore hardly possible for the detector to learn a useful normal behavior from combinations containing random variables.
- **Static variables (Eq. 3.1)**: Assume a combination of a static variable and a variable of some other characteristic (e.g. also static). A combination with a static variable as input for a combinations based detector is equivalent to just feeding both of these variables separately to a detector that detects an anomaly whenever

a new unique value for a single specified variable is found - such as the NMPVD. This kind of detector has much less computational cost compared to detectors that evaluate combinations of variables. It is therefore not necessary to generate combinations of this kind.

The input for the actual procedure of finding combinations is then the set of variables

$$\mathcal{V}' = \mathcal{V} \setminus (V_{random} \cup V_{static}). \tag{3.17}$$

The combinations are selected by the assessment of co-occurrence. Two or more variables are co-occurring if they occur in the same event. Co-occurrence of two or more variables is thereby equivalent to the occurrence of the combination of these. A combination c is selected if it occurs at least θ_{abs} times ("abs" for "absolute"). The set of valid combinations $C \subseteq \mathcal{P}_2(\mathcal{V}')$ is therefore defined as

$$C \coloneqq \{c \in \mathcal{P}_2(\mathcal{V}') \mid Count(c) \geq \theta_{abs}\} \tag{3.18}$$

with $Count(c)$ as the total count of occurrences of combination c and $\mathcal{P}_2(\mathcal{V}')$ as the power set of $\mathcal{V}'$ for combinations of 2 variables. We limit combinations to 2 variables to further decrease the computational effort for this step. However, a later step allows combinations of more than 2 variables. The result of this step is a set of already suitable combinations.

The selection of the parameter θ is not trivial since the number of co-occurrences can be rather arbitrary for different variable combinations and datasets. It is therefore meaningful to choose a relative threshold θ_{rel}. Since there are several variables in a combination that can occur with different frequencies, the threshold value is defined in relation to the total occurrence of the most frequently occurring variable in a combination:

$$\theta_{abs} = \max\left(Count(v) \mid \forall v \in c\right) \cdot \theta_{rel} \tag{3.19}$$

with v as an arbitrary variable in combination c.

Testing has shown that C often consists of too many combinations which can overwhelm the AD tool in terms of computational cost. Especially when running in online mode, the tool has to be efficient enough to process more events per time interval than events are occurring. To address this issue, we can apply a simple graph theory method to merge related combinations. We use graph theory because combinations can be represented as nodes connected by edges in a graph. For instance, the combinations (x, y), (x, z), (y, z), (v, y), (v, w) can be represented as the graph in Fig. 3.2.

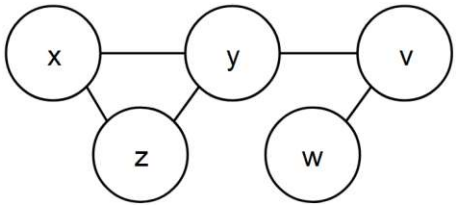


Figure 3.2: Variable combinations represented as connected nodes in a graph.

In the next step we merge all connected nodes. For the example this leads to the combination (v, w, x, y, z). This is possible as for the NMPVCD the set of 2-combinations and the merged combination are equivalent. For instance, assume the combinations (x, y), (x, z), (y, z) consisting of the variables x, y and z. For the NMPVCD these combinations are equivalent to the combination (x, y, z) and since we want to reduce the total number of combinations the latter is preferred. To show that these combinations are equivalent assume the following: The detector knows the triple value combination (x_1, y_1, z_1) from the variable combination (x, y, z) from training. In the test case, it receives the combination (x_1, y_2, z_1). This is an unknown combination and the detector triggers an alert. If this would be the three double combinations (x, y), (x, z), (y, z) instead with the known value combinations (x_1, y_1), (x_1, z_1), (y_1, z_1) then the combination (x, y) would trigger an alert instead because (x_1, y_2) is not known. In both cases the alert would output the same event.

3.4.5 Character Pair Probability

Anomalous values can be detected by assessing their probabilities of occurrence. This can be done in various ways, one of which is the evaluation of the probability of the occurrence of character pairs in an occurring value. Character pairs are consecutive fragments of a character sequence. Such fragments are also called n-grams for sequences of length n. The occurrence probability of such a character pair, or 2-gram, is calculated as follows: For all occurring values, firstly, each occurrence of each character is counted and secondly, for each occurring character the number of times this character occurs after each other character is counted. The probability of a new value can now be calculated by dividing the frequency of each 2-gram $f(a, b)$ by the frequency of the first character $f(a)$, hence $p(a, b) = f(a, b)/f(a)$. C is the set of distinct character pairs of x_i with b following a and character pair $(a, b) \in C$. $p(a, b)$ is therefore defined as the probability that character two will occur after character one. The computation of the probabilities of subsequent character pairs builds upon the previous frequencies. Subsequent values are thus dependent on their predecessors [LWS+23].

To exemplify this procedure, assume that some variable has two subsequent occurrences, “data” (x_0) and “cat” (x_1). The word “data” consists of the 2-grams “da”, “at”, “ta” and “cat” of “ca” and “at”. In order to take beginnings and endings of character sequences into account, a virtual character, e.g. “#”, is added there, such that we have “#data#” instead. The 2-gram “at” has frequency f(“a”,“t”) = 2 and the character “a” has frequency f(“a”) = 5. The probability of “t” occurring after “a” is consequently p(“a”, “t”) = 2/5 for this value. Note, that the occurrences of the characters of a value x_i are already taken into account for the computation of the character pair probabilities of x_i.

By taking the arithmetic mean of all character pair probabilities $p(a, b)$ of a value x_i, including the ones for the virtual characters, we receive a measure for the likelihood of

its occurrence which we call critical value $P(x_i)$:

$$P(x_i) = \frac{1}{|C|} \sum_{(a,b)\in C} p(a,b). \tag{3.20}$$

A detector that utilizes this approach is the EntropyDetector (ED) [LWS+23] which is based on "freq"[3]. The learning of this detector is therefore based on the generation of a frequency table for each character pair. For each occurring value that has a critical value below some defined threshold ϕ an alarm is triggered. We use symbol ϕ instead of θ to point out that this is a threshold which is directly passed to the AMiner as input and not a threshold used to adjust the configuration methods.

To determine which variables to select for this kind of detector, we calculate the mean of all critical values $\overline{P}(x)$ for each variable x, resulting in an overall mean probability measure for each variable. The ones with a mean critical value above a certain threshold are then selected as input for the detector. Variables selected by their character pair probabilities (CPP) of their values are defined as

$$V_{CPP} \coloneqq \{x \in \mathcal{V} \mid \overline{P}(x) \geq \theta_{CPP}\} \quad \text{with} \quad \overline{P}(x) \coloneqq \frac{1}{dim(x)} \sum_i P(x_i). \tag{3.21}$$

In general, the threshold θ_{CPP} can be understood as the minimum mean critical value and is to be chosen empirically by validating with data. It has been observed that a rather high value, in the range between 50% to 80%, yields better results than lower values.

In case of the ED, its configuration expects a specific threshold parameter that decides if a critical value belongs to an anomaly. This threshold parameter $\phi \in [0,1]$ is an indicator for how unlikely a value has to be in order to be detected as an anomaly. We calculate it by taking the minimum of all critical values $P(x_i)$ of selected variable x. From this minimum we also subtract a certain offset δ (in practice, some small non-zero value e.g. between 0.01 and 0.1 is appropriate) to have a buffer between the least likely values that were still considered as normal behavior, such that the same values or the ones with very similar critical values are not considered as anomalies when the training phase is over. Therefore,

$$\phi(x_i) \coloneqq \min_i P(x_i) - \delta. \tag{3.22}$$

As we take the minimum of all critical values as ϕ this will be strongly affected by potential outliers with very low critical values within the training data. However, since we assume this is normal behavior, this trade-off is accepted. Also, this corrects the potentially poor selection of a certain variable by lowering ϕ to a level where only very unlikely values will trigger an alert. In practice it is meaningful to limit ϕ to a certain range since it is possible that the minimum critical value for a certain variable can be arbitrarily low or high within the range of $[0,1]$.

[3]freq GitHub page `https://github.com/MarkBaggett/freq/tree/master`; accessed 16-July-2024.

3.4.6 Event Frequency

Detectors such as the EFD assess the frequencies of events. We define frequencies in this context as "occurrences per time interval". These detectors operate on time windows and thus, on a collection of events rather than the exact events. The number of events in a time window differs depending on the size of the chosen time interval and on the event frequencies in the data. It is therefore also possible to have no events within a certain time window or just one. A detector operating on time windows may therefore not know the exact event that caused an alarm but only the time interval in which it occurred. In case of the EFD of the AMiner, an alarm contains the log line at which the detector was triggered, which may not be the actual anomalous events. This circumstance also implies the requirement of a specific evaluation approach explained in Section 5.3.

Window size

Detectors like the EFD operate on time windows and require the size of these windows as input parameter. Unfortunately, it is not straightforward to define a time window that fits every scenario in the data but we can at least reduce the dependence of the window size parameter of the EFD from the specific nature of the data to have a more generically configured window size. Determining the size of the time windows is still one of the most challenging tasks in time series data mining [ESL23].

The idea to generalize the window size parameter is to somehow define the number of events that should be present in a single window. The window size, denoted by Δt_{window}, is therefore defined as the mean of the non-zero time differences between the occurrences of a variable times a factor n. We say non-zero time differences because events in log data are often logged with the exact same timestamp since single actions on computer systems can generate an arbitrary number of logs that are then counted as different events. This results in time differences between events being often equal to 0. We define these time differences as

$$\Delta t = \{t(x_i) - t(x_{i-1}) \mid i = 0, 1, ..., dim(x) : t(x_i) - t(x_{i-1}) \neq 0\}. \tag{3.23}$$

To this end, the time of the i-th occurrence of a value is defined as $t(x_i)$. Thus, the formula for the window size reads as

$$\Delta t_{window} = \overline{\Delta t} \cdot n. \tag{3.24}$$

This implies that the average number of events with unique timestamps within a time window of size Δt_{window} is n.

Seasonality

For the further analysis we have to resample the arbitrary timestamps of each variable x into equidistant intervals and count the occurrences per time interval. The interval length was chosen as 1 minute. Smaller time intervals might be unfeasible for the case

that the data was collected over a longer time period than just a couple of days. We call the resulting time series $T(t)$. For convenient notation we define it as a function of time t. For instance, $T = [3, 0, 1]$ means the variable x is occurring 3 times in the first minute ($T(t_1) = 3$) and once in the third ($T(t_3) = 1$).

Logs representing computer systems or user behavior often contain reoccurring events. As a consequence, we assume that the occurrence of each variable is, at least partially, behaving periodically. In many cases, this is done by trial and error or visual inspection which is not an option here. The first step is therefore to compute the seasonality (or periodicity) of the occurrence of variables. A common practice for the assessment of seasonality is finding the local extrema of the unbiased autocorrelation function

$$R(t) = \{R_k(T(t)) \mid k = 0, 1, ..., dim(T(t))\}, \tag{3.25}$$

$$R_k(T(t)) = \frac{1}{(n-k)\sigma^2} \sum_{i=1}^{n-k} (T(t_i) - \mu)(T(t_i + k \cdot \Delta t) - \mu) \tag{3.26}$$

with μ and σ^2 as the mean and variance of $T(t)$ and k as the number of lags Δt. Note, that $R(t) \in [-1, 1]$ as each point of the function is a Pearson correlation coefficient. Local maxima in $R(t)$ that are significantly greater (or smaller) than 0 indicate a reoccurring pattern in the data. The seasonality is therefore the time difference between the maxima. The identification of seasonality from a visual inspection of the autocorrelation plot is often straightforward. However, this is considerably more challenging when implemented through an automated process.

For the computation of the maxima we make use of the peak finding algorithm "find_peaks" from the SciPy library [VGO+20] which identifies local maxima within the 1D input array. It scans the array and locates points where the value is greater than that of its neighbors, signifying a potential peak. Subsequently, for each identified local maximum, the algorithm computes its prominence by determining the lowest contour line around the peak. This contour line is established by tracing the valley floor from the peak to the nearest higher peaks, ensuring that the prominence captures the peak's significance relative to its immediate surroundings. Peaks with prominence below a specified threshold are filtered out, resulting in the detection of only prominent peaks. Since $R(t) \in [0, 1]$ we can set the minimum prominence to a fixed value. For the evaluation in Chap. 5 we fix this threshold to 0.1, as a small non-zero value is sufficient to ensure a significant peak.

Based on the peak finding algorithm, we define the function $peaks(\cdot)$ that returns an array of (time) indices of the local maxima of its input array. Hence, the locations (in time) of the local maxima of the autocorrelation function $R(t)$ are

$$t_{peaks} = peaks(R(t)) = \{t_i \mid i = 1, 2, .., dim(R) : R_i \text{ is maxima}\}. \tag{3.27}$$

Given the algorithm's sensitivity to noise, it is necessary to assess whether the identified peaks occur regularly, meaning that the maxima occur in nearly equidistant time intervals.

The time differences between the maxima t_{peaks} are

$$\Delta t_{peaks} = \{t^{j+1}_{peaks} - t^{j}_{peaks} \mid j = 1, 2, ..., dim(t_{peaks}) - 1\}. \tag{3.28}$$

The regularity condition is controlled by the coefficient of variation $c_V = \sigma_{peaks}/\mu_{peaks}$ where σ_{peaks} and μ_{peaks} are the standard deviation and mean value of Δt_{peaks} with an upper limit for the coefficient of variation θ_{c_V} that has to be fixed to some small non-zero value. Testing has shown that $\theta_{c_V} = 0.1$ is sufficiently large to allow small deviations from the mean time difference between the maxima while always capturing the period if there is one (by visual inspection). Additionally, we introduce the constraint that the minimal number of peaks is $dim(t_{peaks}) \geq \gamma_{peaks}$. γ_{peaks} should be chosen as some value greater or equal 3 since it is the least amount of maxima (including the autocorrelation peak of the series with a lag of 0) necessary for indicating a reoccurring pattern in the autocorrelation function. We call the union of these two conditions regularity criterion. If $c_v < \theta_{c_V}$ and $dim(t_{peaks}) \geq \gamma_{peaks}$ we call the peaks regular and the mean of the time differences between the peaks the seasonality $s = \mu_{peaks}$ (of variable x).

Since a minimum amount of occurrences is necessary for a certain number of peaks $\geq \gamma_{peaks}$ a filter is applied to the set of all variables $\mathcal{V}$ beforehand. Variables with an absolute occurrence below $2\gamma_{peaks} - 1$ (peaks plus the valleys) are removed from $\mathcal{V}$ to form the input for the EF method:

$$\mathcal{V}' = \{x \in \mathcal{V} \mid dim(x) \geq 2\gamma_{peaks} - 1\}. \tag{3.29}$$

Since we are mostly interested in the largest local maxima representing the highest peaks of the autocorrelation we apply the peak finding algorithm in an iterative manner by "scanning" from top to bottom. In each iteration we compute the peaks of the autocorrelation that are larger than the minimum autocorrelation threshold θ_R. If the found peaks do not satisfy the regularity condition the threshold is lowered by a small increment (e.g. 0.05). The iteration stops when θ_R exceeds a lower limit or if the seasonality is found. We limit the threshold to $\theta_R \in [0, 1]$ since it is sufficient to only inspect the positive side of the autocorrelation function.

The data often exhibits too much noise which is unfavorable for the autocorrelation function. Consequently, a rolling mean is applied iteratively to the time series $T(t)$ in order to smooth the data. In each iteration, the time window of the rolling mean increases (e.g. with a percentage of the whole time range of the time series) and the peaks are computed (as explained in the paragraph before). If the peaks do not fulfill the regularity criterion a new iteration starts. We condense above mentioned steps into Algorithm 3.1 (j_{max} is the maximal number of smoothing iterations).

Visualizations of the time series, raw and smoothed by moving average, and their autocorrelation functions with marked maxima are given in Fig. 3.3. There one can see, how it becomes more straightforward to find the relevant maxima in the autocorrelation function after the occurrences per minute are smoothed by a rolling average. Note, that the marked peaks in Fig. 3.3b are only marked above a correlation coefficient of 0.1.

Algorithm 3.1: Computation of seasonality

Input : Timeseries T
Output : Seasonality s

for j *in range*(j_{max}) **do**
  Compute $R(T(t))$
  while $\theta_R \in [0, 1]$ **do**
    Compute t_{peaks}, Δt_{peaks}, μ_{peaks}, σ_{peaks}
    $c_V = \sigma_{peaks}/\mu_{peaks}$
    if $c_V < \theta_{c_V}$ *and* $dim(t_{peaks}) \geq \gamma_{peaks}$ **then**
      $s = \mu_{peaks}$
      return s
    end
    Subtract small increment from θ_R and update
  end
  Update T with smoothed T (with window size $j/j_{max} \cdot dim(T)$)
end

To conclude the above, the relevant adjustable parameters for this method are:

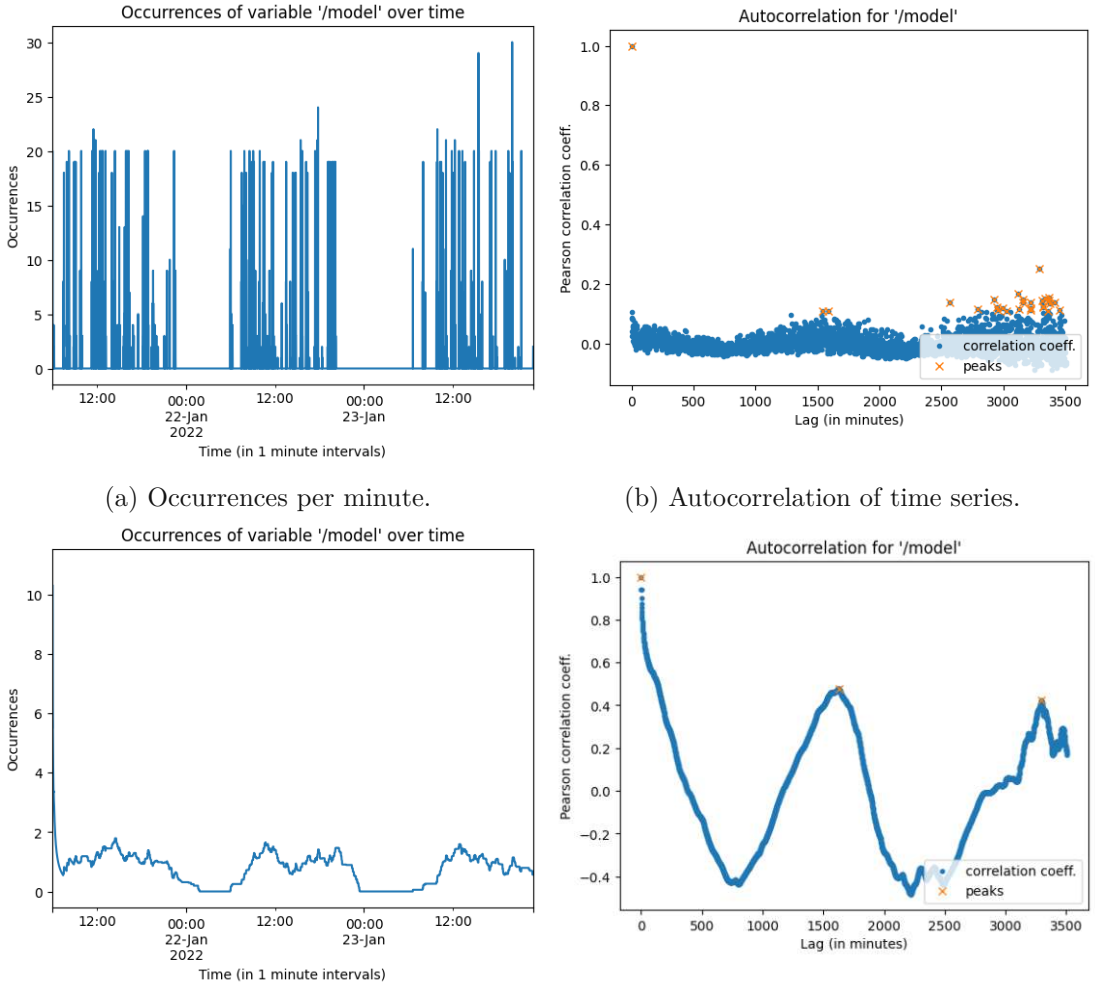
- the average number of values occurring within a time window n,
- the minimum prominence of the peak finding algorithm - fixed to 0.1,
- the upper limit for the coefficient of variation θ_{c_V} - fixed to 0.1,
- the minimum number of peaks necessary for indicating a reoccurring pattern γ_{peaks} - fixed to 3,

Only the defining parameter for the window size n is not fixed. It is therefore investigated in the hyperparameter tuning in Section 5.4.

Variables for which a seasonality and a window size > 0 is found are members of V_{EF}. Thus,

$$V_{EF} = \{x \in \mathcal{V}' \mid \exists s \text{ and } \exists \Delta t_{window} > 0\}. \tag{3.30}$$

Ermshaus et al. [ESL23] review a technique that also computes the period of a time series in a very similar way but apply it in a different context. They use the estimation of the period (or seasonality) as window size. An experiment with the seasonality as the window size turned out to result in unsatisfactory performance for the EFD as the window size is then mostly set too large. This was therefore not further investigated.



(a) Occurrences per minute.

(b) Autocorrelation of time series.

(c) Occurrences per minute smoothed.

(d) Autocorrelation of smoothed time series.

Figure 3.3: Plot of the time series and its autocorrelation function.

Chapter Summary

This chapter described the used datasets, log data in general and what implications log data brings for AD. The data is parsed into a convenient structure to allow a comprehensive analysis. We named the features of the parsed data variables where each occurrence of a variable yields a value.

We described the chosen detectors, that they require variables and other parameters as input and that they output log lines and additional information of the anomalies they detected.

A set of configuration methods was introduced that output what the corresponding detectors require as input. Each of these methods extracts certain information from

the data and defines a set that contains the variables that are suitable for the detector. Furthermore, they also output other parameters such as threshold or other specifications for the configuration process if the associated detector requires some. The configuration methods mostly require some parameters themselves, yet they are less dependent on the character of the data than the original parameters of the detectors and thus, they should not require adjustments. Whether they truly do not require adjustments is investigated in the hyperparameter tuning in Section 5.4.

CHAPTER 4

Modelling

4.1 Pipeline

In this chapter the core functionality and thus the relevant steps of the Configuration-Engine pipeline are explained - starting from data input to final output and every step that takes place inbetween. Since this approach is not AMiner [LWS+23] specific and potentially applicable to a variety of AD tools, it will be described in a more generic way. Figure 4.1 visualizes the process in a flow chart. The steps numbered 0 to 9 in the chart are explained in the following sections.

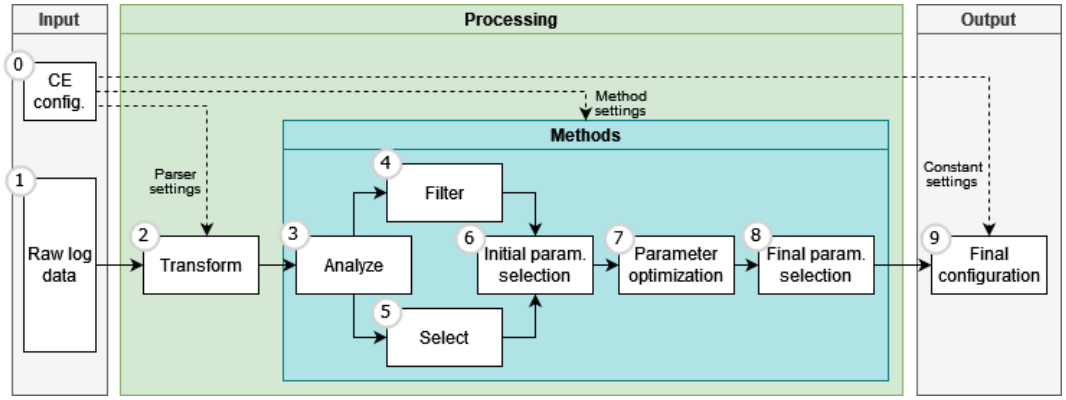


Figure 4.1: Flow chart of the Configuration-Engine pipeline.

4.1.1 Meta configuration (step 0)

In step 0 the configuration of the CE itself is defined. This meta configuration describes the mappings of certain characteristics of the data to the associated detectors. These characteristics, that are assigned to certain detectors in the meta configuration, trigger

the corresponding functions in the CE that extract the relevant information in the data corresponding to these characteristics. These functions are the methods described in Section 3.4. In the best case, this meta configuration remains untouched by users. To be more precise, once there is a meta configuration defined for a certain detector, it should work for any use case independent of the data and, at best, even the data type.

The characteristics relate closely to the detection methods of the detectors. The mapping of these properties to detectors is carried out for a selection of detectors of the AMiner. Table 4.1 shows which detector corresponds to which characteristic or vice versa.

Detector type	Characteristics / Methods
NewMatchPathValueDetector	stability-by-occurrence
NewMatchPathValueComboDetector	minimum-co-occurrence
CharsetDetector	stability-by-character-set
EntropyDetector	average-character-pair-probability
ValueRangeDetector	numeric + stability-by-value-range
EventFrequencyDetector	seasonal-event-frequency

Table 4.1: Mapping of data characteristics to detectors.

The meta configuration resembles the mappings in Table 4.1. Theoretically, every possible detector could be listed in the meta configuration. If an according characteristic cannot be found in the data the detector will not be listed in the final configuration for the AD tool.

General Mapping Strategy

Before a detector can be defined in the meta configuration one has to define the mapping to data properties. These properties are very specific for each detector and closely related to the method they are based on. However, the mapping process itself can be generalized for all detectors. It follows the following sequence:

1. Choose new detector.
2. Understand its detection method.
3. Define data characteristic from the detection method.
4. Expand characteristic to a measure of stability.

A simple example that incorporates these four steps is delivered by the NewMatchPathValueDetector (1). This detector triggers an alert whenever a new and unknown value of a certain class is found in a log line [LWS+23]. Consequently, we do not want to pass certain classes of values to this detector. Imagine a class of values that has a different value in every occurrence (e.g. random variables - Sec. 3.4.2). For this detector any

learning would be irrelevant with this kind of values, and it would trigger false positive alarms for every occurrence (2). The corresponding characteristic is therefore based on "unique occurrence" (3). The corresponding measure of stability is hence the asymptotic decrease of new unique occurrences of variables over time (4). In other words, the occurrence of new and unknown values has to stabilize after some time or some number of events.

Based on the defined methods of this mapping, one can then perform analysis on the data to classify the matching variables into the according feature sets.

4.1.2 Data Ingestion and Transformation (step 1-2)

The CE expects the input to be in the form of event data which in our case is given as log files (step 1).

For the subsequent steps the data has to be transformed into a two-dimensional matrix-like structure that consists of columns and rows. Event data is mostly given as text based lines. If the data is not already organized in a table, a parser has to be applied in order to transform the textual data into a suitable format. The parsing process is covered in Section 3.2 or [LWS+23]. Each row in the resulting table corresponds to an event (or instance) and each column corresponds to a variable (or feature) and its values.

4.1.3 Parameter Selection (step 3-6)

Apart from the training data, for a single detector instance the AD tools we are interested in expect 2 distinct forms of information from a user:

- delineations of where anomalies are anticipated to occur within the data, thus variables,
- specifications regarding the detectors that should be employed, thus specific parameters such as thresholds or window sizes.

Simply put, variables instruct the detector where to search for anomalies, while the other parameters, if given, dictate how the search should be conducted or how strict the criterion for the detection of an anomaly is.

In order to identify suitable variables for each method and also, to assess which method has to be applied or not, the data from the previous step is analyzed and categorized (step 3). In detail, a selection of methods is applied that classifies each variable into a set based on its properties. The further step is then the selection of variables (step 5) that fit the corresponding detection method or by filtering variables (step 4) that do not fit the corresponding detection method from the preceding selection or from the set of all possible variables such that only suitable variables remain.

As certain detection methods require specific input parameters, it is necessary to evaluate the value of each of these parameters. This is by nature closely related to the selection of

variables as these specific parameters often rely on the same principles as the variable selection process. For the case that a valid variable is already found for a certain detection method, we can subsequently carry out further analysis on the characteristics of the variable and its values to assess the value of a specific parameter (step 5). In some cases this might be the other way round and we choose variables for which a suitable specific parameter was successfully extracted from the data (as for the event frequency method in Section 3.4).

The question, whether an available detector should be used or not, is implicitly answered by the circumstance whether a valid parameter setting is found for this detector or not. A detector listed in the meta-configuration may therefore not necessarily be included in the configuration of the detection tool itself. On the other hand, this might generate a large number of detector instances in the configuration and lead to higher runtimes for the detection tool. However, throughout this work the runtimes of the AMiner remained within a feasible range of less than an hour and this was therefore not investigated any further.

The parameter selection procedure reduces the effort from selecting suitable parameters individually to selecting the suitable type of parameters, thus translating the problem to a meta level.

4.1.4 Parameter Optimization (step 7)

In the following, we investigate an approach to systematically optimize a configuration by the feedback obtained from the AD tool itself. The objective is to identify false positive (FP) sources in the configuration and to adapt or delete them in order to minimize their occurrence. These sources can either be variables, options and thresholds or other numerical parameters that were listed in the configuration of a detector. This optimization approach is therefore a measure to reduce the sensitivity of the detection tool to the training data. Thereby, it also reduces the likeliness of overfitting to the training data.

To get feedback from the AD tool, it has to be run with the given data and an initial configuration. The procedure's complexity originates from the assumption that the data does not contain anomalies. An AD tool that only runs on anomaly-free data cannot produce true positive alerts (TP). At first glance, it may appear that having a minimum of FP is always preferable. However, since this might lead to the detection tool being configured too insensible, simply aiming for a minimum is not adequate. A detector, that does not have the potential to trigger an alert, will clearly not produce any false-positives.

Each of the detectors requires a set of variables which can be obtained by the methods of Section 3.4 or by plugging in an already existing configuration. It is not guaranteed that a detector receiving these variables, firstly, holds the potential to identify TP and secondly, does not produce an unreasonable amount of FP alerts. The information of the former is not directly available since we assume anomaly-free data as input, but the latter can be examined more closely.

To get feedback of how the initial configuration performs regarding the amount of FP, the detection tool has to be run with the given data. For the sake of generalization, a cross-validation technique is used here. An adequate technique can be derived from the function "TimeSeriesSplit" from the Scikit-Learn Package [PVG+11]. This method splits the data K times by the time interval while maintaining the order of, first, the train and secondly, the test split. Contrary to TimeSeriesSplit we split the training data based on the number of events since most detection methods used here are not directly time dependent. In fact, testing has shown that in these cases splitting by the number of samples delivers better results. Other than that the method is equivalent to TimeSeriesSplit.

Goldstein and Uchida name two forms of output an AD tool can have: labels and scores [GU16]. However, some of the AMiner's detectors include additional information into their alerts for explanatory reasons which we make use of. From the alert report of the tool the following information for each detector instance is extracted:

1. the number of FP alerts,
2. which event caused the alert (timestamp, line index),
3. information of what caused each alert (some critical value).

It should be noted that detected anomalies are considered as point anomalies. The number of false positives FP and the timestamp t_n of event n can be used to determine if an action is necessary for the corresponding detector instance d. Based on this, two conditions are constructed that assess whether an action has to be taken or not:

1. The underlying idea is simply to check if FP is greater than a certain threshold θ_1 for each detector instance, but since the tool is run for K times we have an array of FP_k for each run k. The data splits are usually not equally sized. Therefore, we take a weighted mean [Coc77], denoted by the bar with the subscripted w, where the weights increase linearly with the size of the corresponding split. For some array v this is

$$\overline{(v)}_w = \frac{\sum_{k=1}^{K} w_k v_k}{\sum_{k=1}^{K} w_k}, \quad w_k = \frac{k}{K}. \tag{4.1}$$

   The condition is then defined as

$$\overline{(FP)}_w > \theta_1. \tag{4.2}$$

2. Similarly, we limit the amount of false positives per time interval. The time interval is defined as $\Delta t = t_n - t_0$ with t_0 and t_n as the timestamp of the first and last alert from the corresponding detector instance. Therefore,

$$\overline{(FP/\Delta t)}_w > \theta_2. \tag{4.3}$$

The information of which event and what caused it can be used to adapt the parameters of the detector instance. Since there is a manifold of detectors and different ways to adapt their different parameters and their possible combinations, the scope of this procedure was limited to the adaptation of numerical thresholds (which is why it is called "threshold optimization") - see Section 14. In conclusion, this decision process is described in Algorithm 4.1 with $d \in c$ as the detector instances in the configuration c:

Algorithm 4.1: Parameter Optimization

Input : Initial configuration c, number of data splits K, thresholds θ_1, θ_2, thresh. minimum ϕ^{min}, thresh. maximum ϕ^{max}

Output : Optimized configuration c'

```
Run AD tool with config. c for K different data splits;
foreach d_i in c do
    if (FP_i)_w > θ_1 or (FP_i/Δt_i)_w > θ_2 then
        if φ_i ∈ d_i then
            φ_i^new = adapted φ_i (threshold optimization);
            if not (φ^min < φ_i^new < φ^max) then
                Delete d_i;
            end
        end
        else
            Delete d_i;
        end
    end
end
```

For the case that the initial configuration was already well designed, this process will in many cases not lead to any increase in performance regarding the amount of produced false-positive alerts, given that the thresholds θ_1, θ_2 were not chosen too small. Thus no detector instances will be deleted from the configuration. However, since the initial selection of variables is provided by a potentially incomplete selection of methods or characteristics, or by a badly designed manually created configuration, it is not guaranteed that all the variables in the selected set are actually suitable.

Moreover, this method could be applied alternatively to the initial variable selection procedure by using all the variables of a dataset as the input for this approach. Since all the variables, that are determined as unsuitable by the condition above, are removed we obtain a set of potentially suitable variables (and thresholds). Also, no previous knowledge of the detection method itself is necessary, assuming the output of the corresponding tool can be analyzed accordingly. The configuration of a detector, that appears as a black-box to the user, can therefore still be optimized.

Threshold optimization

The information of what caused the alert can be used to adapt the thresholds of detector instances, given that they expect one (or more) as input parameter. To be precise, we expect that there is a critical value ϕ^* listed in the alert report of the detection tool that was either too high or too low compared to the threshold that was specified in the corresponding detector instance such that an alarm was triggered.

In order to adapt the given threshold of a detector instance, the critical value for each alarm, given that there is one, is extracted from the alert report. This is done for each of the K runs of the detection tool. We yield multiple critical values for a detector instance for each run k. Out of all of these the minimum is taken. The new adapted threshold for detector instance d_i is therefore,

$$\phi_i^{new} = \min_k(\min_n \phi^*_{i,k,n}) - \delta \tag{4.4}$$

with n as the number of critical values in a single run and offset δ. The offset here has the same functioning as in Section 3.4.5. Thus, to act as a buffer between this minimum critical value and similar critical values that might occur due to future data points. Testing has shown that this procedure can reduce the amount of false-positive alerts while it is still possible to detect the same number of true-positives as before the optimization.

Even tough this procedure is applicable to a wider range of different AD algorithms, it is in our case only relevant for the ED, as this is the only detector of the selection that returns such a critical value for its detected anomalies.

4.1.5 Finalization (step 8-9)

After the parameters are selected (step 3-6) and possibly optimized (step 7) they can be written into a configuration file or something likewise or passed directly to the AD tool (step 8-9). The meta configuration thereby dictates what other information is saved into the final configuration such as parser settings, how the events are handled or other trivial parameters.

4.2 Detector Configuration Assembly

The meta configuration for each detector is either based on a single method of the previously discussed methods in Section 3.4 or assembled as ensembles of multiple methods. In this section we explain the specific mappings of the methods to the detectors according to their requirements explained in Section 3.3. The output of these methods matches the input for the detectors regarding their parameters.

In our case, the mappings require two types of operations. For each detector we can

- **filter variables** from the set of given variables V. Only the remaining variables V' are then passed on to the subsequent operation.

- **select parameters** such as variables (or combinations) from V' (or power set $\mathcal{P}(V')$) or thresholds, window sizes, options, etc. that are passed to the detector.

Based on these operations we can define the mappings for the detectors. Note, that the order of the operations represents the order in which they are applied. The input for the first operation is the set of all variables $\mathcal{V}$. Thereby, we can filter variables that are unnecessary or lead to an unfeasible computational effort for certain methods before they are passed to that method. For the:

- **NewMatchPathValueDetector (NMPVD)** we:
  - **filter static variables**. Even tough these might be interesting for the evaluation, they blow up the configuration unnecessarily and are trivial to detect. Furthermore, the anomalies they cover (of the validation datasets) are also covered by stable variables.
  - **select stable variables (by occurrence)** because they contain a limited set of values.
- **NewMatchPathValueComboDetector (NMPVCD)** we:
  - **filter random variables** as they lead to random combinations which do not form a limited set of value combinations.
  - **filter static variables** as they form unnecessary combinations. The usage of static variables in combinations for this detector is equivalent to the usage of static variables in the NMPVD whose configuration process exhibits less computational effort.
  - **filter variables by minimum occurrence** to reduce the number of possible combinations and make sure that only variables with significantly many occurrences are combined. The choice for this measure is based on computational efficiency. Hereby, we determine the minimum amount of occurrences $0.005 \cdot n_{\mathcal{V}}$.
  - **select variable combinations based on co-occurrence** as only combinations of variables are relevant that co-occur at least a certain amount of times.
- **CharsetDetector (CSD)** we:
  - **filter static variables** for the same reason we filter the m for the NMPVD. Also, a change in their character set implies the occurrence of a new value which can already be detected by the NMPVD.
  - **select stable variables (by character set)** as their values' characters form a limited set.
- **EntropyDetector (ED)** we:

  - **filter static variables** as their critical values are also static. Similar for the CSD a change in their character pair probabilities implies the occurrence of a new value which can also be detected by the NMPVD.
  - **select parameters by character pair probability** as the values of the corresponding variables are on average rather likely to occur (depending on the choice of the threshold), and thus represent a pattern from which the ED is effectively able to learn a normal behavior. The probability threshold is set accordingly.

- **ValueRangeDetector** we:
  - **select stable variables (by value range)** as they are numerical and their minimum-maximum ranges are limited.

- **EventFrequencyDetector** we:
  - **filter variables by minimum occurrence** as for the existence of a certain amount of peaks in the frequency a certain minimal amount of occurrences is necessary. This step is also beneficial for the reduction of computational effort.
  - **select parameters by event frequency** as the behavior of the occurrence frequency of the variables determines the important parameters. In turn, the determination of suitable parameters for certain values implies the identification of these variables as suitable.

4.3 Example Showcase

The application of the configuration process is exemplarily demonstrated for two different detectors of the AMiner [LWS+23] with a small sample of 12 sample log lines that are used as input for the CE and subsequently, for the training of the detection tool. The first log line is given in Listing 4.1 below:

Listing 4.1: Example log line.

```
1 a b10 abc d1 c2
```

The contents of the line are chosen totally at random to demonstrate the applicability of the Configuration Engine to any kind of text-based event data independent of the contextual meaning. The rest of the data was constructed based on the first line in order to have an expressive example. Table 4.2 shows the parsed data.

Index	A	B	C	D	E
0	a	b10	abc	d1	c2
1	a	b0	xyz		c2
2		b12	abc	d1	
3	a		abc		c1
4	a	b0	xyz	d1	
5	a	b18	xy	d2	c2
6	a	b12	xy	d1	c1
7		b13	abc		c2
8	a	b2	abc		c2
9	a	b12	xyz	d1	
10	a	b8	xyz		c2
11		b11	xyz	d2	

Table 4.2: Training data that is assumed to be anomaly-free. The rows represent the log lines, the columns the extracted variables from the log lines. A, B, C, D, E are the variable names assigned by the parser. Note, that $\{A, B, C, D, E\} = \mathcal{V}$.

Then we have the meta configuration from step 0 in Section 4.1.1 with the mappings of the characteristics for each detector which is in this case a .yaml file:

Listing 4.2: Meta-configuration example.

```
ParameterSelection:
  NewMatchPathValueComboDetector:
    Variables:
      PreFilter:
        Static: {}
        Random: {}
      Select:
        Co-OccurrenceCombos:
          min_co_occurrence: 0.5
  EntropyDetector:
    Variables:
      PreFilter:
        Static: {}
      Select:
        CharacterPairProbability:
          thresh_cpp: 0.8
    SpecificParams:
      CharacterPairProbability:
        parameter_name: prob_thresh
        min: 0.1
        max: 0.7
        offset: 0.1
Optimization:
  SampleSplit:
    k: 3
    detectors: [EntropyDetector]
```

```
27      max_FP: 1
28      max_FP_per_minute: 0.05
29      weighted_split: true
30      thresh_optimization:
31        EntropyDetector:
32          name: prob_thresh
33          min: 0.1
34          max: 0.7
35          offset: 0.05
```

The meta-configuration is structured into levels denoted by indents. The first level is reserved for "ParameterSelection" (line 1) and "Optimization" (line 23) which are the two sub-processes of which the CE consists.

The second level in "ParameterSelection" (line 2-22) specifies the detectors that should be used. For each detector there are the options "Variables" (line 3-9 and 11-16) and "SpecificParams" (line 17-22) that define the type of parameters that the corresponding detector expects. Next, the mapping to variable properties is defined. In "Variables" this is one level deeper because it has to be specified first what happens with the variables that correspond to the chosen characteristics. The action "PreFilter" (line 4-6 and 12-13) filters the variables, that were classified into the listed characteristics, from the set of all variables. The other methods of the characteristics then use this set of variables as their input and not the set of all variables. The action "Select" (line 7-9 and 14-16) specifies the variables that should be selected for the configuration of the detection tool. There is also an option "PostFilter" that filters from the set of selected variables before they are written into the configuration (not used here). The lowest levels in the branches of "ParameterSelection" (line 9 and 16) are the meta-parameters of the functions and formulas from Section 4.1.3. For "Optimization" there are the options "Samplesplit" and "Timesplit" (the latter is not given here) which specify the splitting type which is either splitting by number of samples or by time interval.

Configuration of the NewMatchPathValueComboDetector

For each detector we will pre-filter the static and the random variables. This means that they will be removed from the set of all variables so that $\mathcal{V} \setminus (V_{static} \cup V_{random})$ remains for the following steps. From the data in Table 4.2 we get the following number of unique values per variable in Listing 4.3.

Listing 4.3: Unique values per variable.

```
1 {'A': 1, 'B': 8, 'C': 3, 'D': 2, 'E': 2}
```

Therefore, variable A is classified as static because $(UniqueCount(A) = 1$ as in Eq. 3.1. Variable B is classified random because $ValCount(\text{b10}) < 2$ as in Eq. 3.2.

Next, the process computes all possible combinations of variables. The input variables are C, D, E. Their possible pair combinations are given in Listing 4.4.

Listing 4.4: Possible pair combinations.

```
[('C', 'E'), ('C', 'D'), ('E', 'D')]
```

For each combination the CE computes the number of co-occurrences as in Listing 4.5:

Listing 4.5: Number of co-occurrences per variable.

```
('C', 'E') : 8
('C', 'D') : 7
('E', 'D') : 3
```

In Listing 4.2 the threshold "min_co_occurrences" ($= \theta_{rel}$) was set to 0.5, meaning that for each variable combination at least 50% of their values have to co-occur with the variable with the most occurrences. The combination (E, D) is therefore removed. The remaining two combinations can be merged together since they are connected (co-occurrence between variables present in both combinations is sufficiently high - see Sec. 3.4.4), thus we have (C, D, E).

These combinations are now written into the configuration file of the NMPVCD. The configuration file's relevant part can look like Listing 4.6.

Listing 4.6: Configuration of the NewMatchPathValueComboDetector.

```
Analysis:
-   type: NewMatchPathValueComboDetector
    id: id0_CoOccurrenceCombo
    variables:
    - C
    - D
    - E
```

Configuration of the EntropyDetector

Just as for the NMPVCD we remove the static variable A from the set of all variables before initializing the other methods. In general, this is not necessary, but a change of any kind in static values can already be detected by simpler detectors.

The next step is the computation of the character pair probabilities as in Section 3.4.5. The corresponding function counts the occurrences of each value and each occurring value pair. As an example, the frequency counts of the two consecutive values "abc" and "xyz" in variable C are given below. Note, that -1 is a virtual character that represents the delimiters of the character sequence (delimiters are only counted once per value). Listing 4.7 shows what we get for value "abc":

Listing 4.7: Frequency counts and critical values of value "abc" of variable "C".

```
Character pair counts:
{-1: {'a': 1}, 'a': {'b': 1}, 'b': {'c': 1}, 'c': {-1: 1}}
Total character counts:
{-1: 2, 'a': 1, 'b': 1, 'c': 1}
```

```
Critical values:
{'C': [1.0]}
```

Since every count is 1, the probabilities of the character pairs are uniformly 1. Their mean, and thus the critical value, is consequently also 1.

Subsequently, "xyz" extends the character pair probabilities of variable C as given in Listing 4.8.

Listing 4.8: Frequency counts and critical values of values "abc" and "xyz" of variable "C".

```
Character pair counts:
{-1: {'a': 1, 'x': 1}, 'a': {'b': 1}, 'b': {'c': 1}, 'c': {-1: 1}, 'x':
    {'y': 1}, 'y': {'z': 1}, 'z': {-1: 1}}
Total character counts:
{-1: 2, 'a': 1, 'b': 1, 'c': 1, 'x': 1, 'y': 1, 'z': 1}
Critical values:
{'C': [1.0, 0.875]}
```

The probability of the first character pair, (-1, x), is 0.5 because -1 occurred already as the beginning of a character sequence in "abc" and now in "xyz". The probabilities of the other character pairs are 1.0 since they occur the first time. Therefore, the critical value is $\frac{0.5+1+1+1}{4} = 0.875$.

Calculating the critical values for the other values of C yields Listing 4.9.

Listing 4.9: Critical values of variable "C".

```
{'C': [1.0, 0.875, 0.917, 0.9375, 0.85, 0.611, 0.69, 0.875, 0.889, 0.775,
    0.803, 0.824]}
```

The minimum critical value for C is $P(x_5 =$"xy"$) = 0.611$. With an offset of $\delta = 0.1$ the threshold for the EntropyDetector is $\phi = 0.511$ (Eq. 3.22).

The computation of the variables' mean critical values yields Listing 4.10

Listing 4.10: Mean critical values of variables.

```
{'B': 0.7625454545454545,
 'C': 0.83725,
 'D': 0.9127142857142856,
 'E': 0.90075}
```

Since $\theta_{CPP} = 0.8$, we discard variable B and write the remaining variables into the configuration of the EntropyDetector in Listing 4.11.

Listing 4.11: Configuration of the EntropyDetector.

```
Analysis:
-   type: EntropyDetector
    id: id1_CharacterPairProbability
    variables:
```

```
    - C
    prob_thresh: 0.511
-   type: EntropyDetector
    id: id2_CharacterPairProbability
    variables:
    - D
    prob_thresh: 0.7
-   type: EntropyDetector
    id: id3_CharacterPairProbability
    variables:
    - E
    prob_thresh: 0.728
```

Optimization

To optimize the configuration generated in the previous steps, we feed it into the detection tool along with the training data. Actually, a larger amount of data is necessary for meaningful results regarding the optimization approach, but this would be beyond the scope of this small example. Therefore, we assume the already used training data was sampled from a roughly three-times larger dataset such that splitting into $K = 3$ splits can deliver potentially meaningful results. Furthermore, we assume the alert reports for splits $k = 2$ and $k = 3$ of the detection tool are empty, but for the split $k = 1$ two anomalies were reported as given in Listing 4.12:

Listing 4.12: Alert report for split $k = 1$.

```
[{'Detector': EntropyDetector, 'Name': 'id1_CharacterPairProbability',
   'AffectedVariables': ['C'], 'AffectedValues': ['acb'], 'CriticalValue':
   0.24, 'Timestamp': '04/Nov/2023:07:26:48'},
 {'Detector': EntropyDetector, 'Name': 'id1_CharacterPairProbability',
   'AffectedVariables': ['C'], 'AffectedValues': ['bac'], 'CriticalValue':
   0.339, 'Timestamp': '04/Nov/2023:07:29:12'}]
```

The optimization procedure checks if the sensibility of the configuration was too high. Thereby, it has to calculate $\overline{(FP)}_w$ and $\overline{(FP/\Delta t)}_w$ (with $\Delta t_1 = 2.4$ minutes) for every detector instance that appears in the alert report as in the Algorithm 4.1 in Section 4.1.4.

Only "id1_CharacterPairProbability" is present in the alert report. For this detector instance we get:

$$\overline{(FP)}_w = \frac{\sum_{k=1}^{3} \frac{k}{3} \cdot FP_k}{\sum_{k=1}^{3} \frac{k}{3}} = \frac{1}{3}, \tag{4.5}$$

$$\overline{(FP/\Delta t)}_w = \frac{\sum_{k=1}^{3} \frac{k}{3} \cdot \frac{FP_k}{\Delta t_k}}{\sum_{k=1}^{3} \frac{k}{3}} = \frac{1}{3\Delta t_1} = 0.139. \tag{4.6}$$

$\overline{(FP)}_w$ is less than the false-positive threshold $\theta_1 = 1$, but $\overline{(FP/\Delta t)}_w$ is greater than $\theta_2 = 0.05$ the threshold for false-positives per minute. Consequently, the method checks if a threshold parameter is given in the configuration of the corresponding detector instance "id1_CharacterPairProbability". Since there is a threshold given, the optimization of the threshold "prob_thresh" is initialized. The minimum critical value that was reported for "id1_CharacterPairProbability" is 0.24 which is the new value for "prob_thresh" ϕ^{new}. This fulfills $\phi^{min} < \phi^{new} < \phi^{max}$ with $\phi^{min} = 0.1$ and $\phi^{max} = 0.7$. This means that the detector instance is not removed from the configuration, but its threshold "prob_thresh" is updated from $\phi = 0.511$ to $\phi^{new} = 0.24$.

Chapter summary

This chapter described the pipeline of the CE and each of its steps. It takes data as input, processes it through the methods explained in Section 3.4 and outputs the final configuration.

The CE itself is controlled by the meta configuration whose parameters we furthermore call hyperparameters. These are the same parameters that control the configuration methods of Section 3.4. The best possible hyperparameters are determined in the next chapter.

The processing includes the optimization step that prunes or adjusts instances from the configuration that were determined as unfitting. The optimization trains and runs the AMiner multiple times with the initial configuration on the training data and analyzes the feedback to determine if an instance causes too many FP.

Finally, a constructed example was shown to further clarify how each of the steps of the CE works.

CHAPTER 5

Evaluation

5.1 Evaluation Environment

All results were produced with the same evaluation environment. The system runs 64-bit Windows 10 with Ubuntu 20.04 via Windows Subsystem for Linux (WSL) and uses an Intel Core i7-8665U CPU with a base clock speed of 1.90 GHz and 16GB RAM.

The algorithm was programmed and executed in Python 3.8.10. Thereby, the most important tool was the Pandas library [pdt24, McK10] which provides powerful operations and data structures for easy and efficient manipulation of tabular data.

The implementation of the CE can be found at `https://github.com/ait-aecid/aminer-configuration-engine`.

5.2 Evaluation Pipeline

For validation and testing variations of the input are fed into an evaluation pipeline. Each variation of the input consists of a dataset and a meta configuration that holds the parameters for the configuration engine itself.

A single run of the evaluation pipeline consists of the following steps:

1. Split data into train and test (or validation) set.
2. Run CE with training data and meta configuration to produce a configuration file, or take a predefined configuration.
3. Run AD tool with training set and configuration.
4. Run AD tool with test set and configuration.

5. Evaluate by comparing attack labels (ground truth) to alerts (prediction).

This process represents a single run of the evaluation pipeline. By analyzing the output that was produced by different input variations, problems, limitations but also capabilities of the model can be assessed and used for refinement of the process in the validation phase. Application of the same procedure in the test phase shows the model's applicability across different datasets and data types - in this case, Apache access and audit log data (see Section 3.1) - and how different input parameters affect the performance.

5.3 Anomaly Definition

The definition of the absolute performance metrics such as the number of true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN) is not always straightforward in AD and depends on what kind of anomaly (point, collective or contextual) one is considering. Consequently, we have to compare the ground truth (the attack labels) to the prediction (the detected events), to compute the performance metrics [CBK09].

All detectors, except for the EFD, return the exact event index that triggered an alarm. Hence, we compare the line number (or event index) of the attacks and the line number of the detected event and can classify it accordingly into TP, FP, TN, FN. On the other hand, there are detectors that do not return the exact log line of the anomalous event because they operate on time windows. As a consequence, we do not have a unique identification for the anomalous event since there is the possibility of multiple events occurring at the same time or in the specified time period. An anomaly can, in this case, only be identified within a collection of subsequent events or a time frame but not for the exact event (except for the case that only the anomalous event is given in the frame).

Consequently, we treat the detected anomalies in two different ways:

1. We treat all detected anomalies as **point anomalies** and assume their independence from each other. Thereby, it is possible that a detector may identify a significant proportion of the anomalous lines associated with a single attack type, yet simultaneously fails to detect any lines belonging to other attack types.

2. We treat all detected anomalies as **collective anomalies**. For our case, we define a collective anomaly, or attack period, as a sequence of log lines that belong to a specific attack type and are subsequently occurring with no non-attack log lines in between. Attack periods separated by non-attack lines are treated as different collective anomalies or attacks, respectively. A collective anomaly is considered detected if at least one log line of the collective anomaly is detected.

In practice, a system administrator would in case of an alert from the detection tool start an investigation whether it was really a TP or a FP. Hereby, it is important that at

least a single log line of the collective anomaly is detected, under the assumption that the administrator is able to identify the true nature of the alert.

Another issue is posed by the EventFrequencyDetector which detects anomalies within time windows. Hereby, the detection of an anomalous log line can be delayed by up to the length of the time window the detector uses. Therefore, the detector might output a log line which is not part of an attack but is detected due to an attack.

This issue is accommodated by extending the attack period of all collective anomalies. The initial attack period T_{attack} is therefore extended by a certain amount of time δt with t_{start} and t_{end} as the first and last timestamp belonging to an attack:

$$T_{attack}(t_{start}, t_{end}) \rightarrow T'_{attack}(t_{start}, t_{end} + \delta t). \tag{5.1}$$

Consequently, this is not as precise as evaluating the exact event index, treating anomalies as point anomalies.

5.4 Hyperparameter Tuning

Most of the methods from Section 3.4 require input parameters besides the input data. We call these parameters hyperparameters since they are used to control the learning process of the AD tool. They are defined to be as independent of the data as possible, meaning that once an optimal hyperparameter setting is identified it should be suitable for any kind of dataset. This approach aims to minimize the number of settings the user has to adjust. The extent to which the parameters are truly independent of the data can be gauged by examining their performance across different datasets. The desired outcome is for the performance to be consistent across different datasets for the same parameters and at best also for different data types, speaking of Apache and audit log data.

The detectors of the AMiner [LWS+23] operate totally independent from each other. Hence, hyperparameters of methods applied to different detectors can also be tuned independently from each other. The performance of methods with multiple parameters is contingent upon the collective influence of all of them and thus must be addressed accordingly which would require methods like GridSearch, LocalSearch or likewise. For each evaluation the configuration methods, but also the AMiner, have to run for each dataset (16 in total counting both Apache and audit datasets) and for each different parameter. Therefore, the use of search methods is avoided by fixing as many adjustable parameters as possible to a constant such that the tuning can concentrate on the most crucial ones. An evaluation for every possible adjustment would go beyond the scope of this thesis.

To assess what constitutes satisfactory performance, the evaluation metrics of precision and recall (Eq. 2.1) are considered which are often used in the context of AD [LOSW23]. For the selection of the best parameter settings we favor precision over recall as is more important for a single detector to have more TP while little FP than detecting a large

proportion of all the positives irregardless of the FP. Usually multiple different detectors are employed within the AMiner, therefore it is hoped that the undetected attacks by one detector are detected by another. Moreover, it is important to note that some detectors may be unable to detect certain anomalies, as some anomalies may not exhibit any deviations from normal behaviour with respect to certain characteristics.

As static variables are not affected by any change of parameters for the detectors that analyze their values' (constant) properties it is not meaningful to include them for these detectors for the hyperparameter tuning. Thus, they are filtered beforehand which is also convenient regarding the computational effort of this process. The EFD does not take the character of values of variables into account, only their occurrence in time and thus static variables are included for this detector.

5.4.1 NewMatchPathValueDetector

According to the mapping described in Sec. 4.2 the NMPVD expects variables that are stable regarding their unique occurrences. The modifiable parameter is the discrete threshold curve with the thresholds θ_m. As the thresholds should describe some kind of exponential decay we set $\theta_m = e^{-cm}$ and try different values of the decay c. In general, the larger n_s, the smaller are the segments and the less robust the method is to outliers. One would therefore want to have n_s as small as possible while still large enough to capture the behavior of an exponential decay. Furthermore, playing around with this value in the validation phase revealed that how fine-grained the discretization is has little influence on precision and recall (as long as it represents and exponential decay). We therefore fix $n_s = 5$.

The thresholds used for the tuning process are visualized in Fig. 5.1. The c's are chosen in a way so that the thresholds cover the meaningful range of all possible thresholds. Additionally, $c = 100$ should show that a too strong decay usually poses a too limiting constraint.

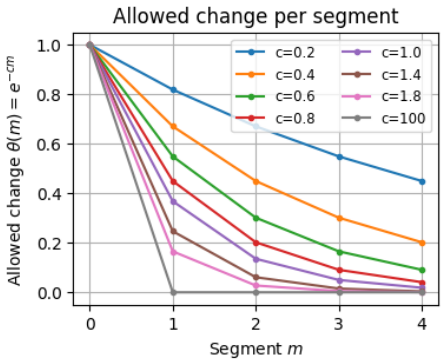
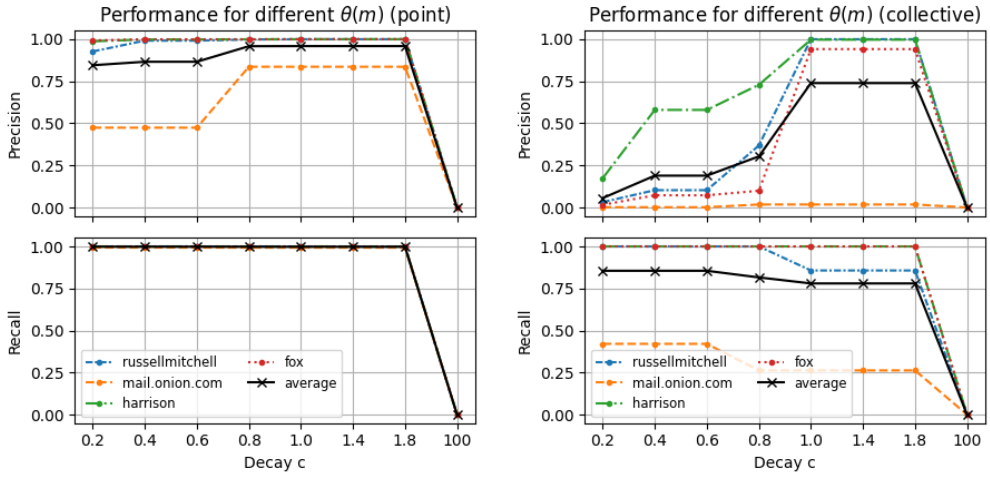
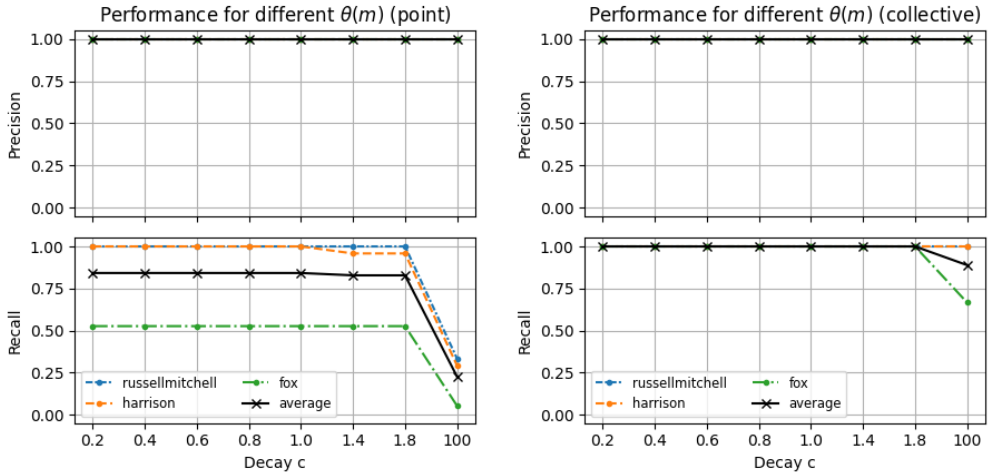


Figure 5.1: Visualization of the stability thresholds θ_m for different values of c.

From visual inspection of the performance metrics in Fig. 5.2 we conclude that the best



(a) Performance for Apache (point) (b) Performance for Apache (collective).



(c) Performance for audit (point). (d) Performance for audit (collective)

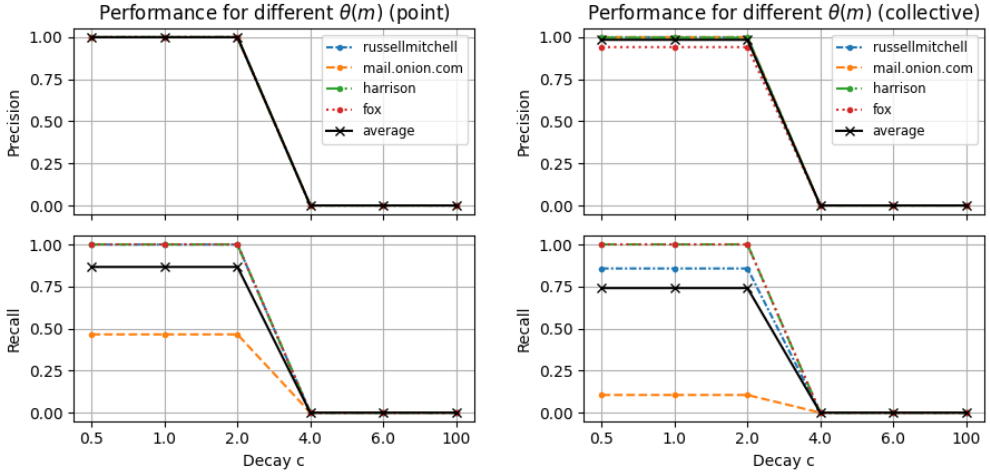
Figure 5.2: Precision and recall for the NewMatchPathValueDetector, configured with different threshold curves.

setting of the thresholds is $c = 1.8$, thus $\theta = [1.0, 0.165, 0.027, 0.005, 0.001]$, as it yields high precision for all datasets while the average recall is still at least 0.75.

5.4.2 CharsetDetector

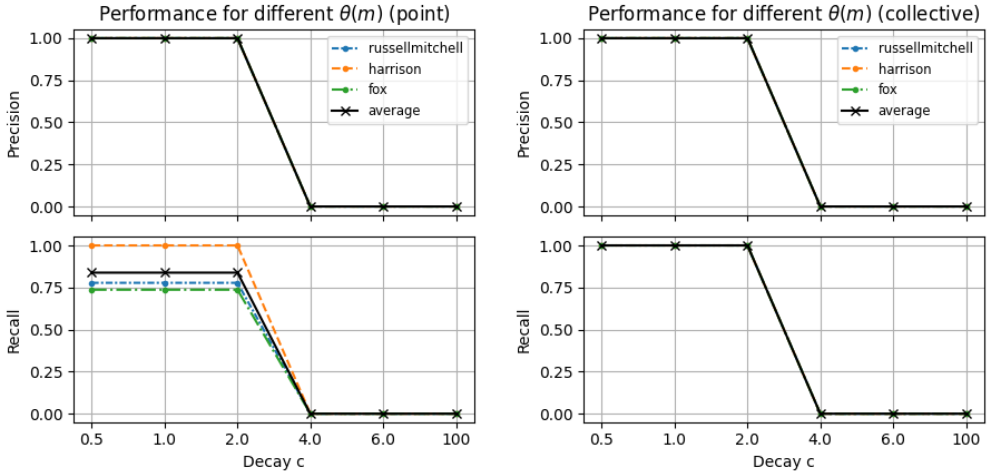
The procedure for the CSD is exactly the same as for the NMPVD but different values for the stability thresholds are used. We fix the number of thresholds to 10 and test for $c = 0.5, 1, 2, 4, 6, 100$. A higher resolution and more restrictive thresholds seemed necessary.

In Fig. 5.3 one can see that there is only change visible between $c = 2$ and $c = 4$. It seems that each threshold $c < 4$ is suitable. We choose $c = 1$ and thus $\theta_m = [1.0, 0.368, 0.135, 0.05, 0.018, 0.007, 0.002, 0.001, 0.0, 0.0]$ for the further analysis. In general, the characteristic "stability of character set" seems to be quite robust for all datasets as long as it is not too strict ($c > 2$). One could argue that the stability criterion for this detector is unnecessarily complex. However, since there is no downside, except a negligible increase in runtime for the selection process, we keep this criterion for the CharsetDetector as it makes sense to have some kind of limitation to the variables.



(a) Performance for Apache (point).

(b) Performance for Apache (collective).



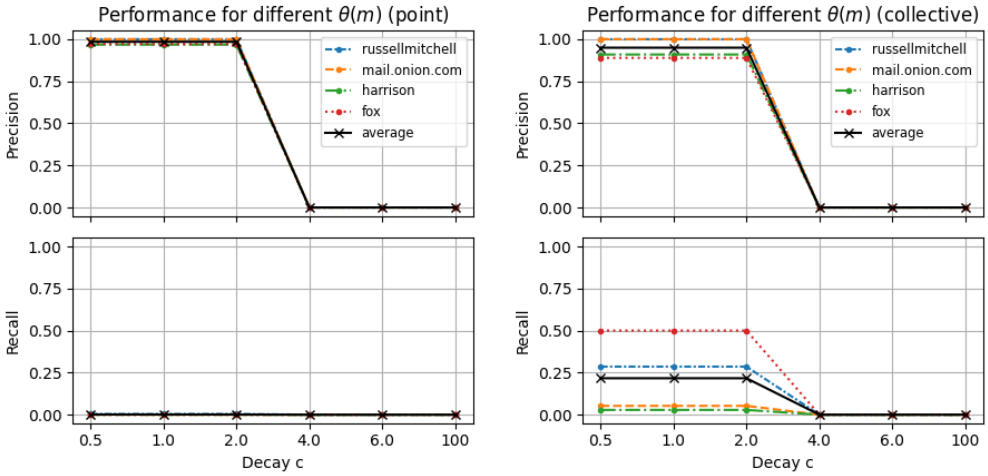
(c) Performance for audit (point).

(d) Performance for audit (collective).

Figure 5.3: Precision and recall for the CharsetDetector, configured with different threshold curves and tested on each dataset.

5.4.3 ValueRangeDetector

The VRD also expects stable variables and thus it is the same procedure as for the NMPVD. For this detector we expect little change as in the data there are almost no numeric variables given in the datasets. For instance, for the russellmitchell Apache dataset there are only 2 ("status code" and "content size"), for audit there are 0. Consequently, the number of different values the performance metrics can have for Apache data is very limited and it is not necessary to assess the performance of the VRD for audit data. However, in order to not use "magic values" for the stability thresholds it is necessary to justify them in some way.



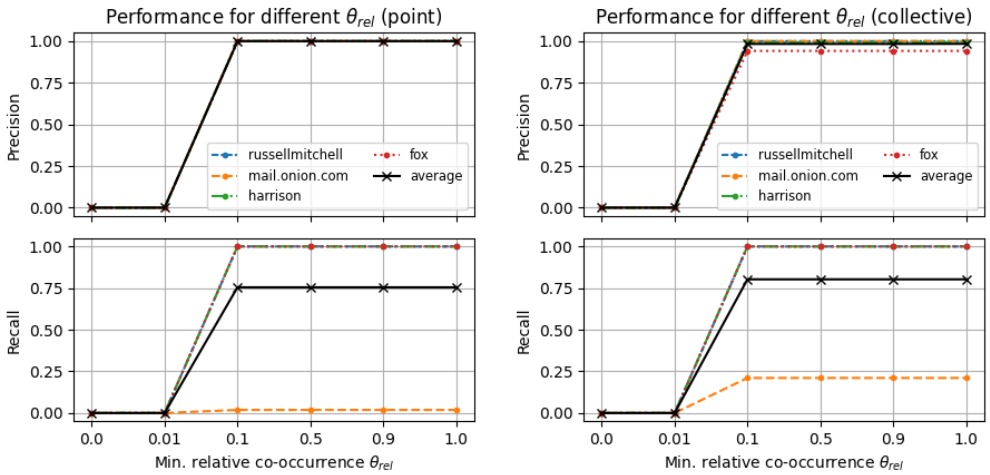
(a) Performance for Apache (point). (b) Performance for Apache (collective).

Figure 5.4: Precision and recall for the ValueRangeDetector, configured with different threshold curves and tested on each dataset.

One can see, the performance remains unchanged until $c > 4$. As for the CSD this could indicate that the stability criterion is unnecessarily complex for this detector and that the only necessary constraint on the variables is that they are numeric. In the case of our data, this is indeed the situation, as by inspecting the generated configuration files, there are always both numeric variables specified for each run where neither precision nor recall is 0 (only variables are specified for the VRD). Nevertheless, to keep the stability criterion for this detector guarantees that numerical variables for possible other datasets are limited in their value range and there is no downside except a negligible increase in runtime. For convenience we choose $c = 1$ as for the CSD for further investigations.

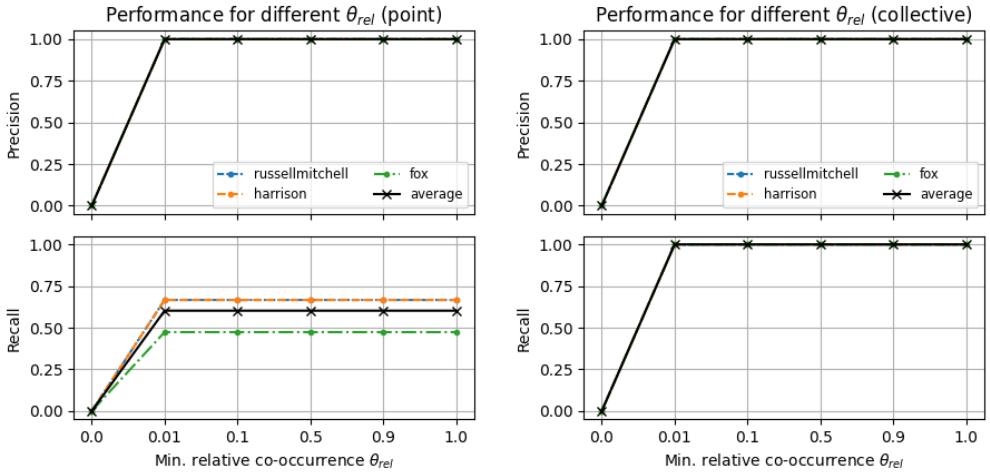
5.4.4 NewMatchPathValueComboDetector

The modifiable parameter for the NMPVCD is the relative minimum co-occurrence θ_{rel}. We evaluate the following values $\theta_{rel} = 0, 0.01, 0.1, 0.5, 0.9, 1.0$.



(a) Point anomalies for Apache data.

(b) Collective anomalies for Apache data.



(c) Point anomalies for audit data.

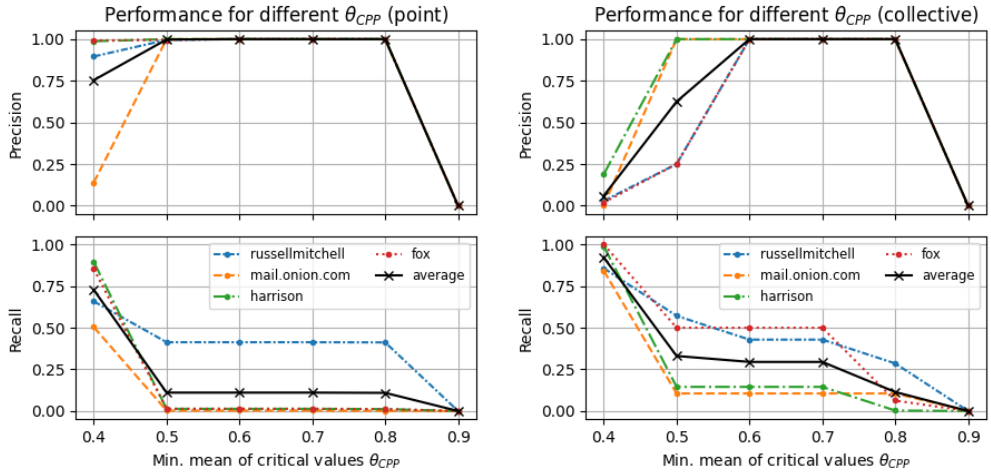
(d) Collective anomalies for audit data.

Figure 5.5: Precision and recall for the NewMatchPathValueComboDetector, configured with different minimum co-occurrence thresholds and tested on each dataset.

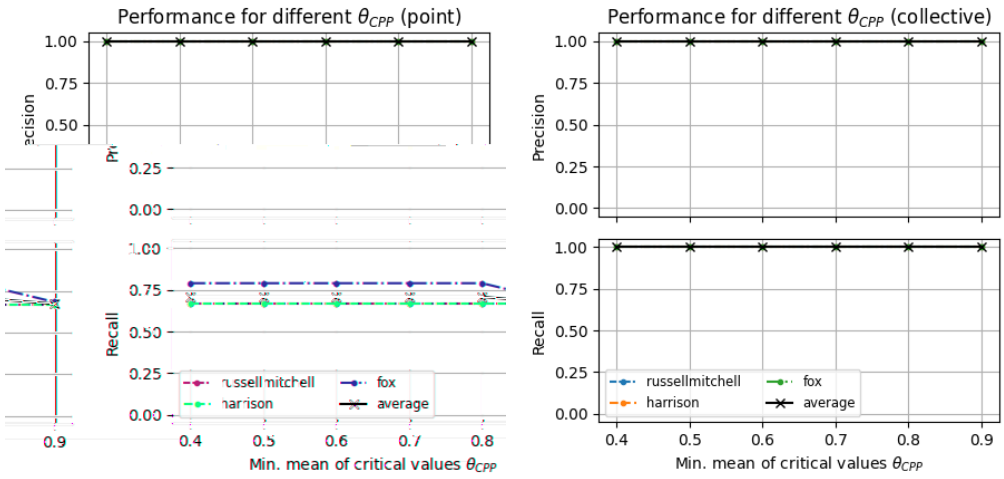
The plots of Figure 5.5 exhibit an "all-or-nothing" behavior for the performance which reveals quite an interesting property of the data: that if variables of a suitable combination co-occur a certain minimum number of times they occur almost always together (with the variable in the combination with the most occurrences). Hence, enforcing a higher minimum co-occurrence of variables is not necessary and we choose $\theta_{rel} = 0.1$ for the evaluations with the test data.

5.4.5 EntropyDetector

The meaningful range for the lower limit of the mean critical value θ_{CPP} is covered by $\theta_{CPP} = 0.4, 0.5, 0.6, 0.7, 0.8, 0.9$. We limit the parameter for the probability threshold to $\phi \in [0.0, 0.9]$ as we can reasonably conclude that every occurrence with a critical value above 0.9 is likely enough to occur and also, to exclude the possibility of $\phi = 1$. The offset is set to $\delta = -0.05$ whereby any small non-zero value (with a significant magnitude) is sufficient.



(a) Performance for Apache (point). (b) Performance for Apache (collective).



(c) Performance for audit (point). (d) Performance for audit (collective).

Figure 5.6: Precision and recall for the EntropyDetector, configured with different θ_{CPP}.

From the plots in Fig. 5.6 we conclude that $\theta_{CPP} = 0.7$ clearly poses the best setting for the ED for Apache access data. Interestingly, for audit data the performance is very

robust to changes in this parameter while still performing outstandingly well. If we would only evaluate with audit data, we could not determine $\theta_{CPP} = 0.7$ as the best setting.

5.4.6 EventFrequencyDetector

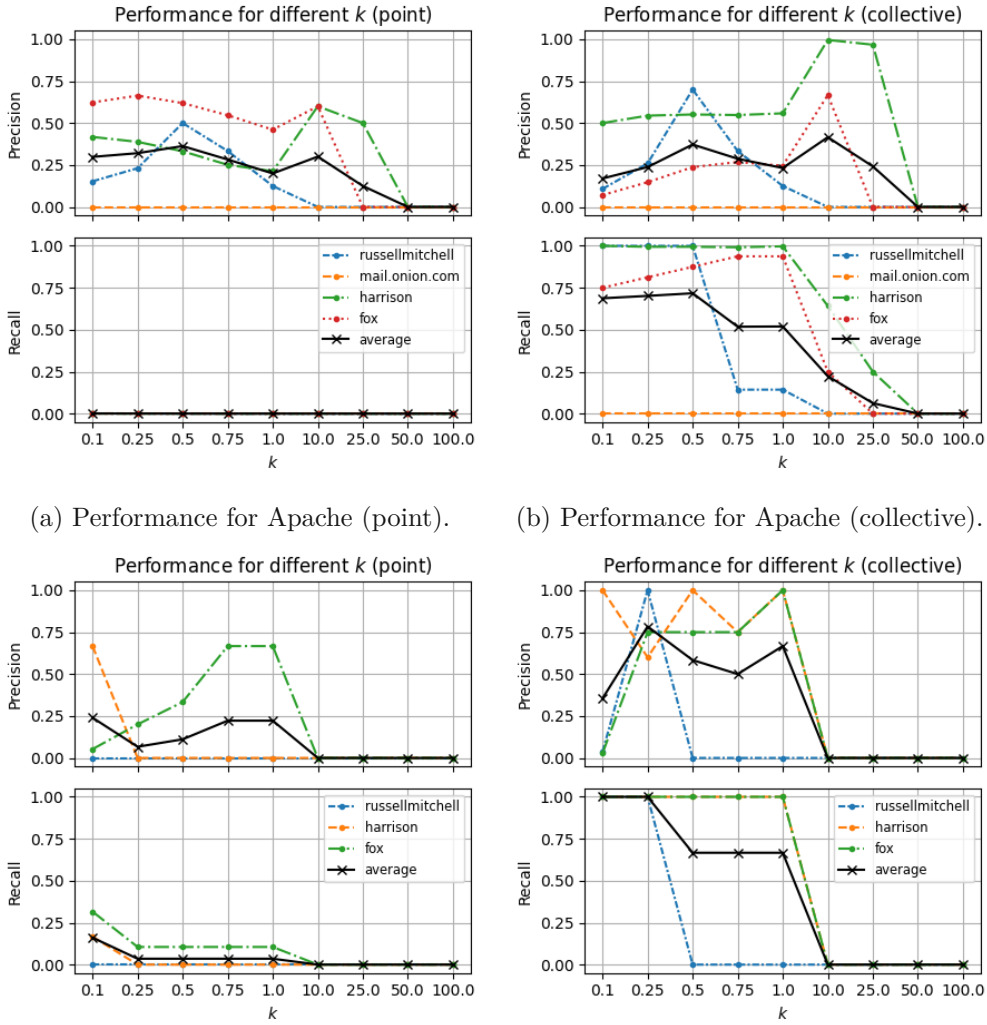
The time period of the collective anomalies was extended by 5 minutes. To justify the selection of this value empirically, we let the EFD run on the audit validation datasets with $n = 0.5$ and evaluate the results with a time extension of $\delta t = 0$ and $\delta t = 5$ minutes. In Table 5.1 one can see that the performance improves by a decrease of FP and increase of TP (for harrison and fox). For all these datasets the attacks occur in a period of less than 3 minutes. After the attack each dataset has more than 10 hours of normal log lines. Thus, it would be very unlikely if the detected anomalies are this close to the true anomalies by coincidence. Therefore, we assume that 5 minutes is a valid number for extending the attack periods of the collective anomalies. Too large extensions would skew the evaluation in an overly optimistic way.

Dataset	FP	TN	TP	FN	δt
russellmitchell	0	455	0	2	0
russellmitchell	0	455	0	2	5
harrison	1	372	0	3	0
harrison	0	373	3	0	5
fox	2	804	1	2	0
fox	1	805	3	0	5

Table 5.1: Absolute performance metrics depending on the time extension (in minutes) of the attack periods for the audit validation datasets with $n = 0.5$.

The modifiable parameter was tested with $n = 0.1, 0.25, 0.5, 0.75, 1, 10, 25, 50, 100$. The plots of Fig. 5.7 show a relatively chaotic behavior compared to the plots of the previous methods. Nevertheless, the average over all datasets' performances indicates a common behavior. For audit the best setting is $n = 0.25$ with an average performance of over 75% precision and 100% recall, yet the chaotic behavior might indicate that this is only by coincidence the best value. The results for Apache are less satisfying but present an opportunity for improvement through the optimization in a later section. For Apache we choose $n = 0.5$ for the further evaluations. Given that the results for the other detectors are rather similar for both audit and Apache Access data, it is possible to choose a suitable value for both data types. For the case of the EFD we unfortunately have two different values and thus, this would be a setting chosen by a user.

The rather chaotic results indicate that the event frequency method leaves room for possible improvements as it would be favourable to have similar behavior for audit and Apache data. Every difference in behavior implies the necessity of adjustments for users. However, there is the possibility that no common setting is suitable for all datasets regarding the window size and that the problem lies with the EFD.



(a) Performance for Apache (point).

(b) Performance for Apache (collective).

(c) Performance for audit (point).

(d) Performance for audit (collective).

Figure 5.7: Precision and recall for the EventFrequencyDetector, configured with different θ_{CPP}.

5.5 General Performance

5.5.1 Baseline

The baseline consists of 3 manually created configurations from experts in the domain of cybersecurity, each of which has deep knowledge about the AMiner and is involved in its development or maintenance. The time it takes to manually create a configuration depends on the experience, how complex and diverse the data is and which detectors one wants to employ, just to name a few of the uncertainties. A very vague estimation would

suggest that the process takes between one hour and a whole work day.

A snippet of one of the expert's configurations is given in Listing 5.1 and shows two detector instances. Both instances contain multiple variables in a single instance. For the NMPVCD this is the expected scenario. However, for the case of the EFD the AMiner will analyze all variables as if they are one. This also works for all the other selected detectors. In many cases this is equivalent if, for instance, both variables contain values that are totally different. Of course, this also depends on the detector and in which aspect it assesses two values as different. Sometimes this is done out of convenience to reduce the number of instances, but mostly the expert combines variables that he or she believes interconnected. Such decisions are often based on experience. The CE does not account for these possibilities as no significant increase in performance is expected by passing multiple variables in a single instance and always passes only a single variable per detector instance (except for the NMPVCD).

Listing 5.1: Cutout of an experts' configuration file.

```
- type: "NewMatchPathValueComboDetector"
  paths:
  - "/model/client_ip/client_ip"
  - "/model/combined/combined/user_agent"
  - "/model/fm/request/request"
  - "/model/fm/request/method"

- type: "EventFrequencyDetector"
  id: "accesslog_frequency"
  paths:
  - "/model/fm/request/request"
  - "/model/fm/request/method"
  window_size: 10
```

Configurations for the AMiner may also contain variables that are not given in the data. The AMiner ignores instances containing such variables. However, when comparing the baseline to the CE only the relevant parts of the expert configurations are considered. Relevant parts of the configuration in this context are the variables present in the used dataset and detector instances of detectors in the made selection. This is especially important for the assessment of similarity in Section 5.6.

The configurations of the experts are evaluated for their performance by plugging them into the evaluation pipeline instead of generating a configuration from scratch (step 2).

5.5.2 Performance of initial Parameter Selection

In order to demonstrate the efficacy of each detector for both the CE and the baseline we will first assess the performance of each detector individually. The graphs in Fig. 5.8 and 5.9 show the detection performance of each detector configured with the CE compared to the performance of the experts' configurations. The evaluation metrics shown in the plots are computed for each Apache and audit data as an average of the configurations' performances on the datasets.

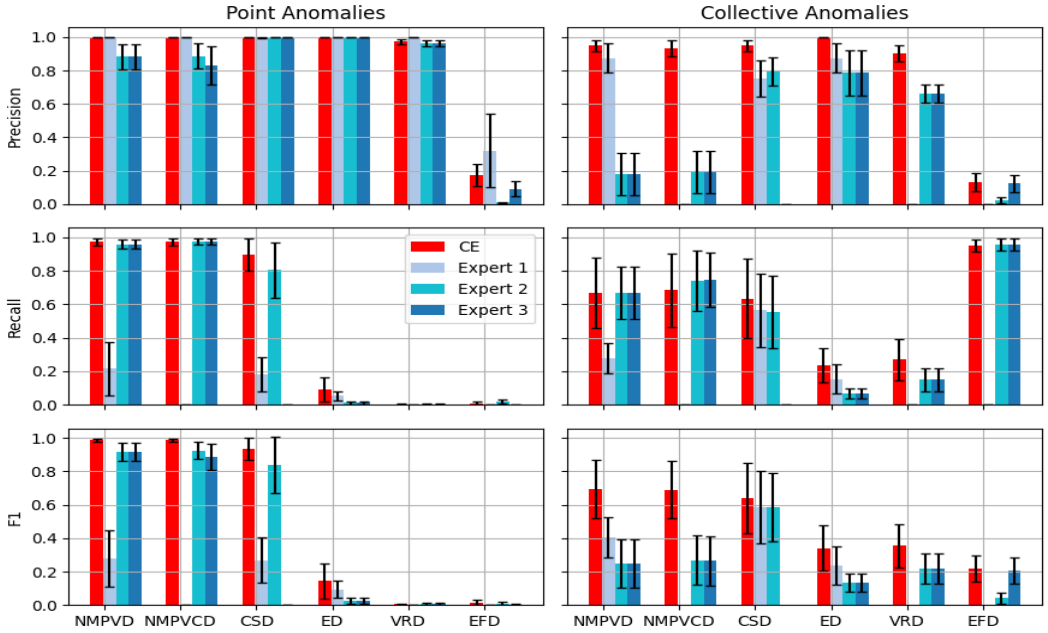


Figure 5.8: Precision, recall and F1-score averaged over all Apache Access datasets for each detector configured with the CE and the experts' configurations.

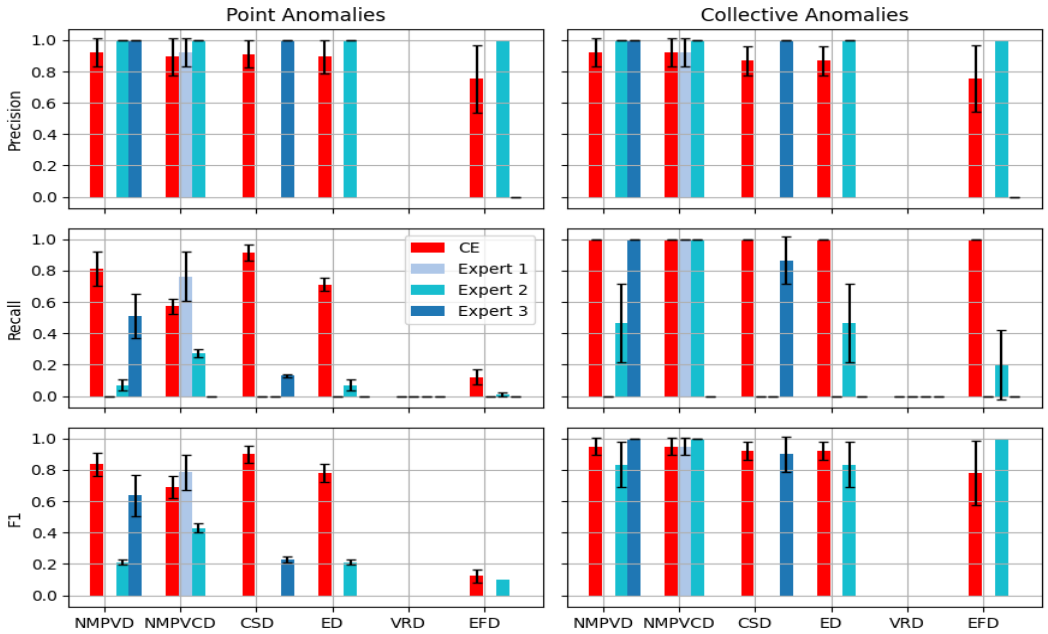


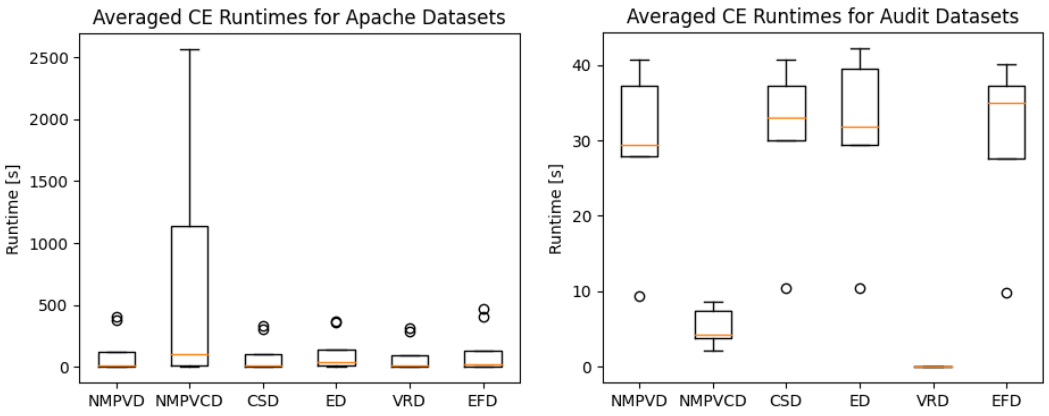
Figure 5.9: Precision, recall and F1-score averaged over all audit datasets for each detector configured with the CE and the experts' configurations.

Besides precision and recall we also show the F1-score (Eq. 2.2) which is the combination of both. This score is often used in literature and thus also used here for better comparability to other works in literature [LOSW23].

Especially for Apache data, the CE can keep up with the experts, maintaining a high precision for all detectors but the EFD. Since the expert configurations are similarly unsatisfactory for the EFD it can be assumed that the problem lies with the detector. In terms of collective anomalies for Apache, the CE even outperforms the experts, indicating that a wide range of different attack types are precisely detected with the help of the CE.

For audit data the CE performs slightly less satisfying than for Apache. However, the decrease in performance is almost only caused by the shaw dataset. For instance, the NMPVD, NMPVCD, CSD and ED all have precision 1.0 when using the median instead of the mean. We anticipate that the optimization is the remedy. Furthermore, one can see that the expert configurations often have 0 in their performance metrics. This is due to the fact that the configurations do not contain instances for all detectors for audit data, as in many cases it is not straightforward which variables or even which detectors are meaningful for a data type.

Additionally, Fig. 5.10 shows the runtimes of the CE for generating the configurations averaged over each dataset. In order to increase the reliability of the results the runtimes were averaged over 3 runs of the CE.



(a) Runtimes for Apache.

(b) Runtimes for audit.

Figure 5.10: Averaged runtimes for the CE for each detector.

One can see that the runtimes are mostly relatively low, whereas the configuration method for the NMPVCD exhibits larger execution times for Apache data because of the strong scaling of the method and the large number of samples in some of the Apache datasets (see Table 3.1). The maximum runtime was on average 2559 seconds (roughly 42 minutes) for the dataset mail.cup.com.

We let the CE now configure all detectors at once and allow all detectors of the selection

for the expert configurations. Figures 5.11 and 5.12 show the aggregated performances of all detectors of the selection.

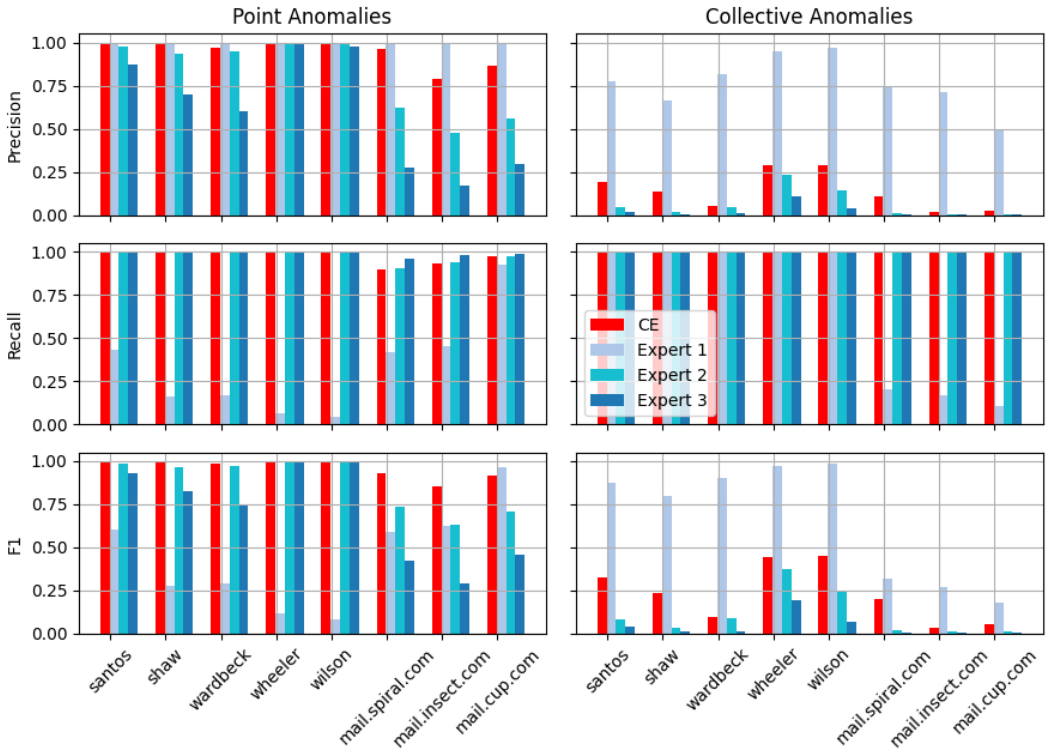


Figure 5.11: Precision, recall and F1-score for each Apache dataset, configured with the CE and the experts' configurations.

For Apache data the aggregated performances seem very promising for point anomalies while at the same time the precision for collective anomalies is less satisfactory. This is due to the fact that there are simply less true collective anomalies while the amount of false positives is the same as for point anomalies. Especially, the configuration of expert 1 maintains a solid precision for both audit and Apache compared to the others, yet with a lower recall for audit.

The precision for the audit datasets seems less satisfactory at first sight, yet one has to consider that the number of true anomalies is also very small (see Table 3.1). A single FP is therefore much more weighted for the audit datasets than for the Apache ones. For instance, for audit dataset shaw there are 14 FP and 18 TP. The comparison of the precision for the individual detectors in Fig. 5.9 with the aggregated precision in Fig. 5.12 indicates that the different detectors are often detecting different FP in the same dataset which leads to a poorer aggregated precision. Nevertheless, the experts' configurations outperform the CE which suggests that the CE is simply less reliable for audit data.

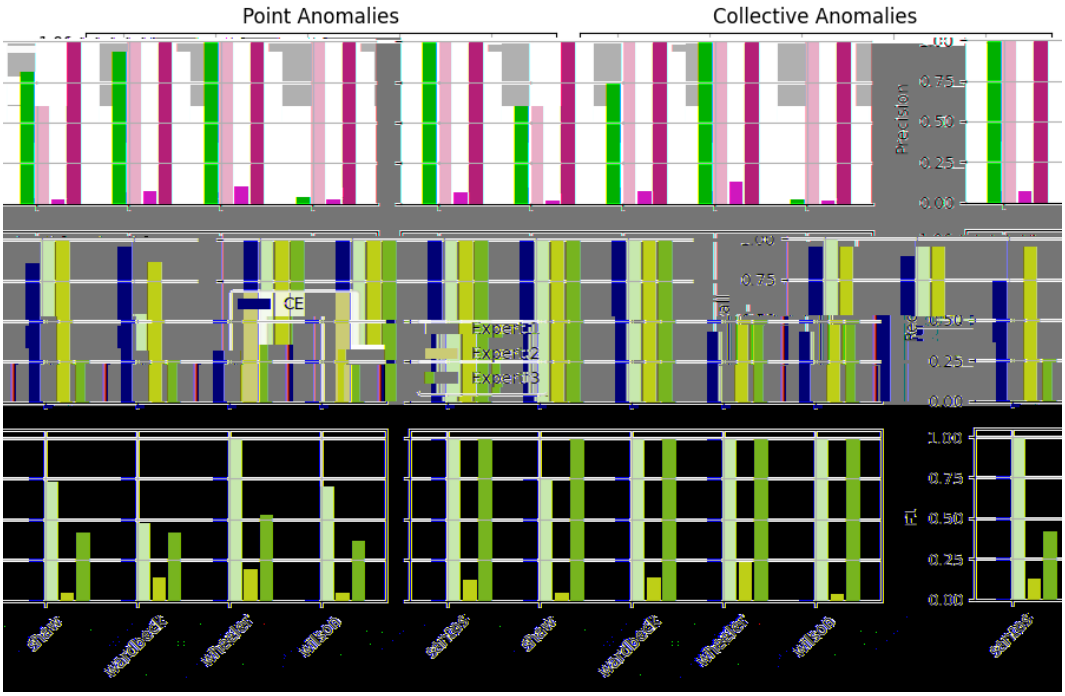


Figure 5.12: Precision, recall and F1-score for each audit dataset, configured with the CE and the experts' configurations.

5.5.3 Optimization

For the evaluation of the optimization approach we take the configuration of the CE and the experts as input. The optimized configurations are then fed into the evaluation pipeline. We fix the number of allowed FP to $\theta_1 = 10$ and the number of FP per minute to $\theta_2 = 0.05$ (both weighted) as they proved to be effective during the validation phase. In general, some small numbers have to be chosen for these thresholds to tolerate a small amount of falsely detected anomalies as we only present limited portions of the training data. The number of splits is set to $K = 3$.

Figures 5.13, 5.14 and 5.15, 5.16 visualize the detection performances of the different configurations for each dataset, with and without the optimization. We can see, that the optimization indeed improves the overall performance for both CE and the experts at the cost of almost no reduction in recall, except for the 3 Apache datasets from the AIT Log Data Set V1.0, especially for collective anomalies. However, the precision greatly increased for both types of datasets.

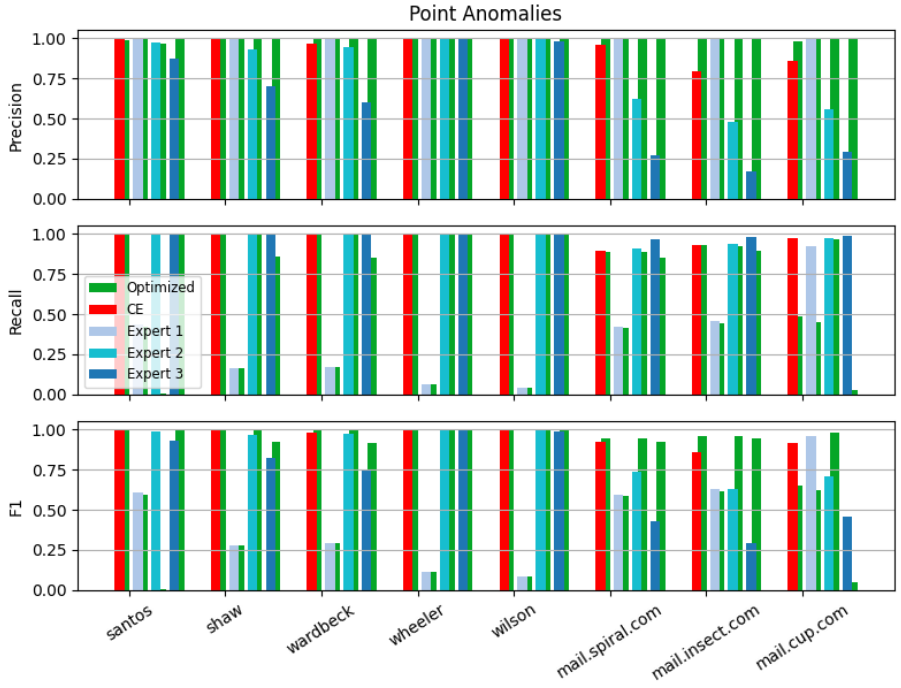


Figure 5.13: Precision and recall for point anomalies for each Apache dataset, configured with the CE and the experts' configurations with and without the optimization.

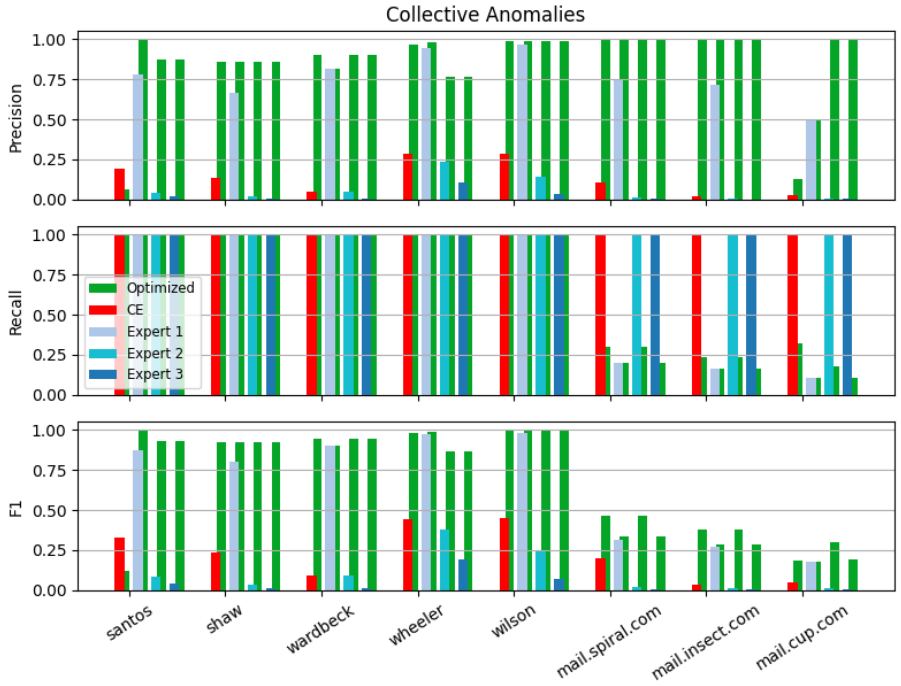


Figure 5.14: Precision and recall for collective anomalies for each Apache dataset, configured with the CE and the experts' configurations with and without the optimization.

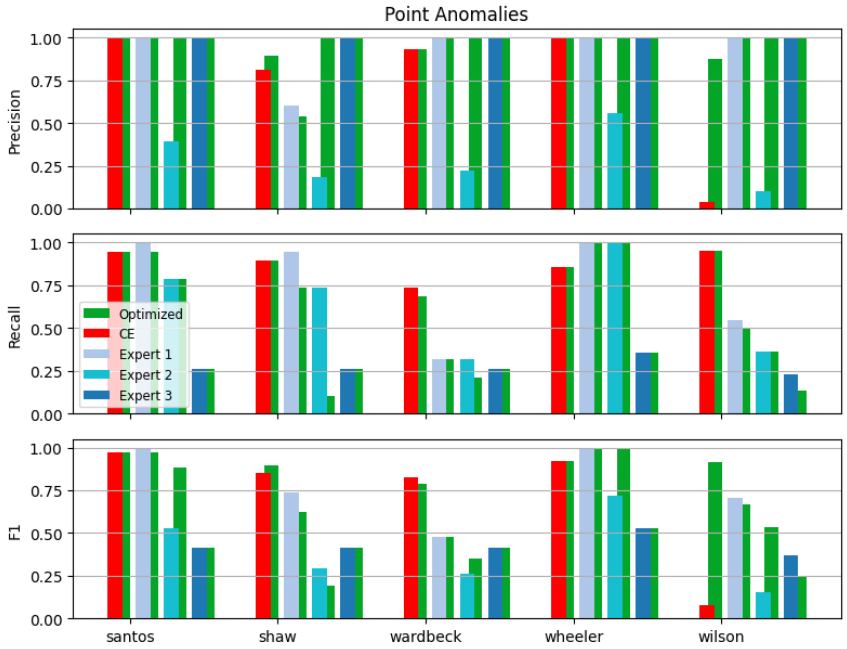


Figure 5.15: Precision and recall for point anomalies for each audit dataset, configured with the CE and the experts' configurations with and without the optimization.

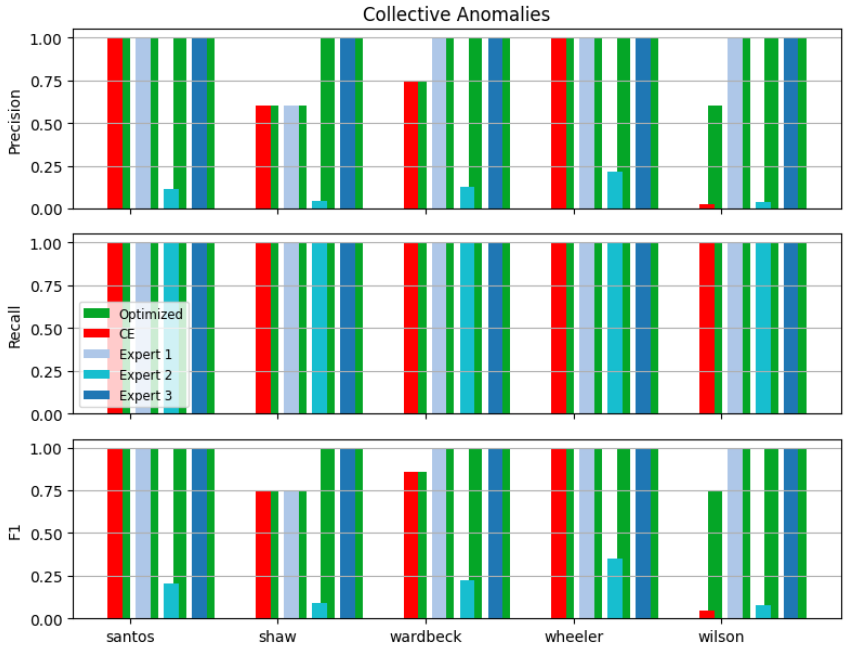


Figure 5.16: Precision and recall for collective anomalies for each audit dataset, configured with the CE and the experts' configurations with and without the optimization.

5.6 Similarity of Configurations

The previous evaluation approach is solely based on the evaluation of traditional machine learning metrics, but the generated configurations themselves might hold additional information worth investigating. Therefore, we want to assess how similar the artificial configurations are, compared to the manually created ones from domain experts. Note, that for the evaluation of the similarity we use the optimized configurations of the CE and the non-optimized expert configurations.

The configurations can be represented as associative arrays. They consist of different keys and values of different types. Consequently, it is not straightforward how to measure their similarity. A meaningful and general approach is therefore to assess the similarity of all present detector-variable pairs since this defines what features of the data the corresponding detector should investigate. In case of the manual configurations, it shows which features the domain expert considers as important.

A detector-variable pair is given as (d_a, A) with detector d and the set of variables A for the configuration a which is defined as a set of detector-variable pairs

$$a = \{(d_a, A)_i \mid i = 1, ..., |a|\}. \tag{5.2}$$

For all detectors but the NMPVCD we have $|A| = 1$ since they take a single variable as input. The NMPVCD expects combinations of variables which is why we define A as a set of variables and $|A| \geq 1$. The comparison of two configurations a and b is then between the sets of variables A and B for which $d_a = d_b$ holds. For this comparison we use the Jaccard similarity coefficient [dFC21] which is generally defined as the size of the intersection of two sets A and B divided by the size of their union

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{5.3}$$

where $|\cdot|$ denotes the cardinality of a set. The comparison of pairs that are not from the NMPVCD can only yield $J(A, B) = 0$ or $J(A, B) = 1$. If it would not be for the NMPVCD we would not need the Jaccard index. We define

$$\widetilde{J}(a, b) = \begin{cases} J(A, B) & \text{if } d_a = d_b, \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

to impose the condition. The similarity $s(a, b) \in [0, 1]$ of configuration a and configuration b is then the sum over all Jaccard indices where $d_a = d_b$ divided by the size of the configuration b. Thus,

$$s(a, b) = \frac{1}{|b|} \sum_{i}^{|a|} \sum_{j}^{|b|} \widetilde{J}(a_i, b_j). \tag{5.5}$$

The sums over the modified Jaccard indices $\widetilde{J}$ are equivalent to the sum of every element of a matrix where only the elements are non-zero where $d_a = d_b$.

For the graphs in Fig. 5.17, 5.18 we use a as the artificial configuration of the CE and b as the expert's configuration. Since the formula divides by the size of the expert configuration b the similarity can be understood as: the percentage of detector-variable pairs in the expert configuration b that are also given in the artificial configuration a.

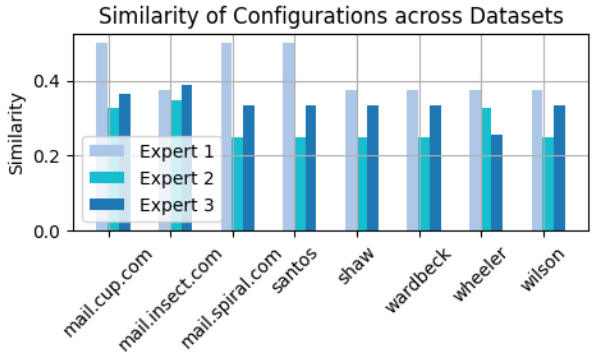


Figure 5.17: Similarity between the CE and the expert configurations for different Apache datasets.

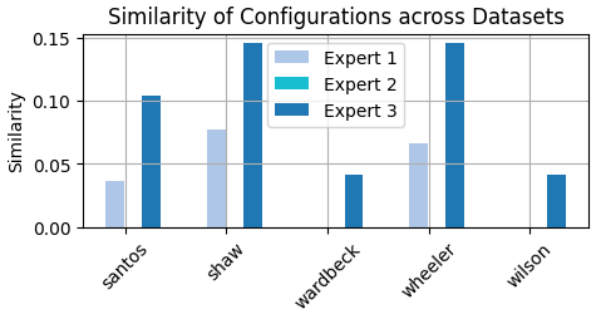


Figure 5.18: Similarity between the CE and the expert configurations for different audit datasets.

By comparison of the performance of Fig. 5.15 and the similarities in 5.18 one can see that even tough a similarity of only 0.07 at most is given between the configurations of the CE and expert 1 for e.g. dataset santos and wheeler, expert 1 and the CE achieve a precision of 1.0. This suggests that there are many different variables that exhibit the same anomalies. It is also interesting that the configurations of expert 2 exhibit very little similarity, just as the expert 1 configurations, but performing very poor for audit data.

The heatmaps from Fig. 5.19, 5.20 show how similar the configurations from the CE are with each other compared across different datasets. Since the similarity formula defined in Eq. 5.5 is not commutative we get different values for whichever configuration we define as a and which as b. As we also get a satisfying performance for precision and recall across almost all datasets it follows that the configurations are effectively portable

from one dataset to another, due to their high similarity, especially for Apache datasets. This further implies that for Apache the important variables for the detection are mostly the same across different datasets (of the same type). At least for the made selection of detectors and datasets, this suggests that it is possible to define a single suitable configuration for all Apache datasets, at least in terms of the variables. Interestingly, the configuration of dataset mail.spiral.com differs the most from all the other Apache datasets, yet it does not differ significantly in performance, not even from the other datasets from AIT Log Data Set V1.0. One reason why the similarity of configurations for the shaw and santos audit configurations is significantly lower than for the wardbeck, wheeler and wilson ones is that the santos and shaw audit configurations do not contain instances with the NMPVCD.

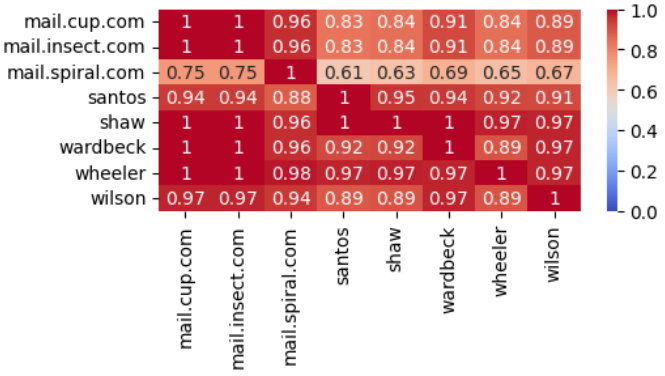


Figure 5.19: Heatmap of similarities $s(a, b)$ across the configurations generated on different Apache datasets (a on y-axis, b on x-axis).

Figure 5.20: Heatmap of similarities $s(a, b)$ across the configurations generated on different audit datasets (a on y-axis, b on x-axis).

Listing 5.2 shows a snippet of the configuration created by expert 3. The snippet contains detector instances with variables that usually appear in Apache Access datasets. In the following, we exemplarily describe the differences between the CE configuration for dataset shaw[1] (a) and Listing 5.2 (b). Given the high similarity among all Apache

[1]Configuration for dataset "shaw" on CE GitHub page, `https://github.com/ait-aecid/aminer-configuration-engine/blob/main/configurations/generated/apache/shaw/shaw.yaml`; accessed 21-July-2024.

configurations (Fig. 5.19), comparing just one configuration is sufficient. From Listing 5.2 we are only interested in "type" which refers to the detector type and "paths" which refers to variables. There we have the following detectors:

1. **NewMatchPathValueDetector** (line 2-8): "/model/client_ip/client_ip" is also listed in the NMPVD instances of the shaw configuration but not "/model/user_id" and "/model/fm/request/request". Therefore, $s_1(a, b) = 1/3$.
2. **NewMatchPathValueComboDetector** (line 10-22)): Is not present in the shaw configuration, thus $s_2(a, b) = 0$.
3. **CharsetDetector** (line 24-30): "/model/combined/combined/referer" is also present for the CSD in the shaw configuration. $s_3(a, b) = 1$.
4. **EntropyDetector** (line 32-36): "/model/fm/request/request" is not given for any EntropyDetector instance. $s_4(a, b) = 0$.
5. **ValueRangeDetector** (line 38-42): "/model/content_size" is given for the VRD of the shaw configuration. $s_5(a, b) = 1$.
6. **EventFrequencyDetector** (line 44-49): The detector is not listed in the shaw configuration, thus $s_6(a, b) = 0$

With $|b| = 7$ this leads to $s(a, b) = \sum_i^6 s_i(a, b)/7 = 1/3 + 0 + 1 + 0 + 1 + 0 \approx 0.33$ which is exactly the value visible in Fig. 5.17 for the similarity between the configurations of the CE and expert 3.

Listing 5.2: Snippet of the configuration file of expert 3.

```
1  Analysis:
2    - type: NewMatchPathValueDetector
3      id: apache_NMPVD
4      persistence_id: apache_NMPVD
5      paths:
6        - "/model/client_ip/client_ip"
7        - "/model/user_id"
8        - "/model/fm/request/request"
9
10   - type: NewMatchPathValueComboDetector
11     id: apache_NMPVCD
12     persistence_id: apache_NMPVCD
13     paths:
14       - "/model/client_ip/client_ip"
15       - "/model/combined/combined/user_agent"
16
17   - type: NewMatchPathValueComboDetector
18     id: apache_NMPVCD2
19     persistence_id: apache_NMPVCD2
20     paths:
21       - "/model/combined/combined/referer"
```

```
22        - "/model/fm/request/request"
23
24    - type: CharsetDetector
25      id: apache_CD
26      persistence_id: apache_CD
27      id_path_list:
28        - "/model/client_ip/client_ip"
29      paths:
30        - "/model/combined/combined/referer"
31
32    - type: EntropyDetector
33      id: apache_ED
34      persistence_id: apache_ED
35      paths:
36        - "/model/fm/request/request"
37
38    - type: ValueRangeDetector
39      id: apache_VRD
40      persistence_id: apache_VRD
41      paths:
42        - "/model/content_size"
43
44    - type: EventFrequencyDetector
45      id: apache_EFD
46      persistence_id: apache_EFD
47      paths:
48        - "/model/status_code"
49      window_size: 60
```

The CE's configurations contain significantly more instances of detectors than the experts' ones, as the CE analyzes every available variable of the data. The experts focus on a smaller set of variables, which are typically more obvious indicators for anomalies. For instance, for audit the experts consider the variables related to "syscall" (system call), "exe" (executable) or "uid" (user id) as especially important. The user id, for example, is important to see which user triggered which actions. For Apache data the status code is considered the most important, but also the most obvious variable for the manifestation of anomalies. Also "request" related variables are present in all three configurations of the experts. Such variables occur many times in a short period of time for scan attacks and are therefore especially useful for the EFD. An attacker aware of the typical variables in which an intrusion attempt might manifest itself might be able to evade an anomalous behavior in such a variable. On the other hand, detection tools configured by the CE might find anomalies that manifest themselves in variables that neither experts nor attackers would consider. For instance, for the CSD the configurations generated on the shaw dataset contain 17 instances for Apache and 37 for audit, while the configurations of expert 1, 2 and 3 contain 3, 2 and 2 CSD instances (for both audit and Apache variables).

Chapter Summary

In this chapter we provided a clear definition for anomalies and how we evaluate them to guarantee a transparent evaluation.

Furthermore, this chapter contains the results of the hyperparameter tuning that determined which are the best hyperparameters for the parameter selection methods or the meta configuration, respectively. With this setting we showed the overall performance of the CE compared to the baseline of manually created configurations and the performance with and without the optimization.

Additionally, the similarity of manual and synthetic configurations is computed with the Jaccard index to show how different or how similar the experts and the Configuration-Engine configure the AMiner and how similar the generated configurations are across different datasets.

With the results presented in this chapter the research question and the associated sub-questions from Section 1.2 can be answered satisfactorily.

CHAPTER 6

Conclusion

This work introduces the Configuration-Engine (CE), a novel approach for the configuration of anomaly detection algorithms in a semi-supervised manner. The CE analyzes system log data under the assumption of the data being anomaly-free. The dataset is parsed into a set of variables. The core of the CE is the classification of these variables into sets based on their character and behavior over time. The objective is the automation of the tedious configuration process that usually requires domain expertise. These sets of variables exhibit some measure of learnability for the associated detector. The characteristics that define a variable set totally depend on the requirements of the detector one wants to configure. The most important characteristics that were defined in this work are stability, co-occurrence, character pair probability and event frequency. Each configuration method is based on these characteristics and serves as an extension to their associated detection algorithm to minimize the necessary input by transforming the extracted information from the data into the input parameters of the associated detectors.

The approach is demonstrated and evaluated on the AMiner and its detectors, yet the general approach of defining a configuration method for a detector is applicable to any kind of anomaly detection algorithm. On the other hand, the defined configuration methods are more tailored to the specific detectors of the AMiner or to those utilizing similar detection techniques. Encouragingly, the evaluated configuration methods achieve satisfactory results with which we can answer the research question:

To what extent can automated configuration methods improve the effectiveness of anomaly detection tools in identifying intrusions compared to manually created configurations, with respect to detection metrics such as precision and recall in audit and Apache log data?

With the CE the detectors of the AMiner are able to detect collective anomalies with an average precision of over 0.95 for the Apache datasets and over 0.9 for the audit

datasets for all detectors excluding the EFD. In fact, almost all of the collective and point anomalies were discovered in each dataset with and without the optimization approach. Nevertheless, the aggregated performance of the detectors for audit datasets was mediocre while 2 of the 3 experts' configurations performed very well with precision and recall being mostly 1.0. Encouragingly, the optimization was able to effectively improve the overall precision of almost all configurations by a significant amount (while mostly maintaining a high recall), by pruning the FP sources that were often introduced by the EFD.

Moreover, it is shown that the experts' configurations are considerably dissimilar to the artificial configurations of the CE. At the same time, the CE's configurations are surprisingly similar to each other across different datasets, indicating the possibility of effective portability of configurations across different datasets, especially for Apache data.

Future Work

The CE and its underlying methods provide a solid basis for future research with numerous possibilities for improvements and extensions that were not covered due to the limited scope of this work.

Thereby, one area of interest is the expansion of the range of configuration methods for detectors. The existing framework is robust, yet only a decent selection of configuration methods is defined which have potential for refinement, especially the method for the EFD. Furthermore, testing the methods of the CE with more diverse datasets and incorporating additional parsers or even a parser generator would increase its universal applicability.

Also, it would be beneficial to improve the efficiency of the configuration methods or even the whole process itself. In terms of the implementation, especially streaming in the data in a line-by-line or batch-wise manner, maybe coupled with parallelization, could improve the CE's efficiency and lower the computational limitation for the amount of input data. In the same context, it might be useful to extract and condense the data of datasets with low information density beforehand to reduce the computational effort for the CE.

List of Figures

List of Tables

List of Algorithms

Bibliography

[ATD19] Amjad M. Al Tobi and Ishbel Duncan. Improving intrusion detection model prediction by threshold adaptation. *Information*, 10(5), 2019.

[BS93] Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1(1):1–23, 1993.

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

[Coc77] William Gemmell Cochran. *Sampling techniques*. john wiley & sons, 1977.

[cro24] Crowdstrike 2024 global threat report, 2024. Retrieved from `https://www.crowdstrike.com/resources/reports/crowdstrike-2024-global-threat-report/`, accessed 24-June-2024.

[dFC21] Luciano da F. Costa. Further generalizations of the jaccard index. *CoRR*, abs/2110.09619, 2021.

[DLZS17] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1285–1298, New York, NY, USA, 2017. Association for Computing Machinery.

[ESL23] Arik Ermshaus, Patrick Schäfer, and Ulf Leser. Window size selection in unsupervised time series analytics: A review and benchmark. In Thomas Guyet, Georgiana Ifrim, Simon Malinowski, Anthony Bagnall, Patrick Shafer, and Vincent Lemaire, editors, *Advanced Analytics and Learning on Temporal Data*, pages 83–101, Cham, 2023. Springer International Publishing.

[GU16] Markus Goldstein and Seiichi Uchida. A comparative evaluation of unsupervised anomaly detection algorithms for multivariate data. *PLOS ONE*, 11(4):1–31, 04 2016.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[HZH+18] 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*, 15(6):931–944, 2018.

[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, 2016.

[HZHL20] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. Loghub: A large collection of system log datasets towards automated log analytics. *CoRR*, abs/2008.06448, 2020.

[KBD+08] Marius Kloft, Ulf Brefeld, Patrick Düessel, Christian Gehl, and Pavel Laskov. Automatic feature selection for anomaly detection. In *Proceedings of the 1st ACM Workshop on Workshop on AISec*, AISec '08, page 71–76, New York, NY, USA, 2008. Association for Computing Machinery.

[KV03] Christopher Kruegel and Giovanni Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, page 251–261, New York, NY, USA, 2003. Association for Computing Machinery.

[LCT+23] Ifigeneia Lella, Cosmin Ciobanu, Eleni Tsekmezoglou, Marianthi Theocharidou, Erika Magonara, Apostolos Malatras, Rossen Svetozarov Naydenov, et al. Enisa threat landscape 2023: July 2022 to june 2023. 2023.

[LFY+10] Jian-Guang Lou, Qiang Fu, Shenqi Yang, Ye Xu, and Jiang Li. Mining invariants from console logs for system problem detection. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, June 2010.

[LHW+21] Max Landauer, Georg Höld, Markus Wurzenberger, Florian Skopik, and Andreas Rauber. Iterative selection of categorical variables for log data anomaly detection. In *Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26*, pages 757–777. Springer, 2021.

[LOSW23] Max Landauer, Sebastian Onder, Florian Skopik, and Markus Wurzenberger. Deep learning for anomaly detection in log data: A survey. *Machine Learning with Applications*, 12:100470, 2023.

[LSF+22] Max Landauer, Florian Skopik, Maximilian Frank, Wolfgang Hotwagner, Markus Wurzenberger, and Andreas Rauber. Ait log data set v2.0, February 2022. M. Landauer, F. Skopik, M. Frank, W. Hotwagner, M. Wurzenberger, and A. Rauber. "Maintainable Log Datasets for Evaluation of Intrusion Detection Systems". arXiv:2203.08580.

[LSW+20] Max Landauer, Florian Skopik, Markus Wurzenberger, Wolfgang Hotwagner, and Andreas Rauber. Ait log data set v1.1, November 2020. M. Landauer, F. Skopik, M. Wurzenberger, W. Hotwagner and A. Rauber, "Have it Your Way: Generating Customized Log Datasets With a Model-Driven Simulation Testbed," in IEEE Transactions on Reliability, vol. 70, no. 1, pp. 402-415, March 2021, doi: 10.1109/TR.2020.3031317.

[LSW23] Max Landauer, Florian Skopik, and Markus Wurzenberger. A critical review of common log data sets used for evaluation of sequence-based anomaly detection techniques. *arXiv preprint arXiv:2309.02854*, 2023.

[LWS+23] Max Landauer, Markus Wurzenberger, Florian Skopik, Wolfgang Hotwagner, and Georg Höld. Aminer: A modular log data analysis pipeline for anomaly-based intrusion detection. *Digital Threats*, 4(1), mar 2023.

[McK10] Wes McKinney. Data structures for statistical computing in python. In *SciPy*, volume 445, pages 51–56, 2010.

[MLZ+19] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In *IJCAI*, volume 19, pages 4739–4745, 2019.

[PdOV+12] Cláudia Pascoal, M. Rosário de Oliveira, Rui Valadas, Peter Filzmoser, Paulo Salvador, and António Pacheco. Robust feature selection and robust pca for internet traffic anomaly detection. In *2012 Proceedings IEEE INFOCOM*, pages 1755–1763, 2012.

[pdt24] The pandas development team. pandas-dev/pandas: Pandas, April 2024.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[TD04] David MJ Tax and Robert PW Duin. Support vector data description. *Machine learning*, 54:45–66, 2004.

[TH19] Ayman Taha and Ali Hadi. Anomaly detection methods for categorical data: A review. *ACM Computing Surveys*, 52:1–35, 05 2019.

[Vaa03] R. 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.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[WHLS24] Markus Wurzenberger, Georg Höld, Max Landauer, and Florian Skopik. Analysis of statistical properties of variables in log data for advanced anomaly detection in cyber security. *Computers Security*, 137:103631, 2024.

[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, April 2019.

[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 *Proceedings of the International Conference on Information Systems Security and Privacy (ICISSP)*, pages 386–397, 01 2018.

[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*, SOSP '09, page 117–132, New York, NY, USA, 2009. Association for Computing Machinery.

[ZXL+19] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, Junjie Chen,

Xiaoting He, Randolph Yao, Jian-Guang Lou, Murali Chintalapati, Furao Shen, and Dongmei Zhang. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 807–817, New York, NY, USA, 2019. Association for Computing Machinery.