Logs to the Rescue

Creating meaningful representations from log files for Anomaly Detection

G.H.R. Timmerman

Master Thesis

Logs to the Rescue

Creating meaningful representations from log files for Anomaly Detection

by

G.H.R. Timmerman

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday September 27, 2023 at 3:00 PM.

Student number:	5664829
Institution:	Delft University of Technology
Faculty:	Electrical Engineering, Mathematics and Computer Science
Project duration:	December 2022 – September 2023
Thesis committee:	S. Verwer, TU Delft, Associate Professor, Cybersecurity and Algorithms (supervisor)
	A. Anand, TU Delft, Associate Professor, Web Information Systems
	T. Mulder, Eye Security, Incident Response Lead (company supervisor)

Preface

Before you lies my Master Thesis called ”Logs to the Rescue: Creating meaningful representations from log files for Anomaly Detection”. It focuses on improving the information extraction from logs using language models and clustering to reduce the complexity of log file representation for the purpose of anomaly detection when investigating cyber incidents.

This thesis was written to fulfil the graduation requirements of the Master Computer Science with a Cybersecurity specialization at the Delft University of Technology. The study was done in combination with an internship at Eye Security during the last year of my study between December 2022 and September 2023 for a total of 9 months.

I would like to express my sincere gratitude to my daily supervisor Assoc. Prof. Ir. Sicco Verwer (TUDelft) and company supervisor Tijmen Mulder (Eye Security) for their guidance and support in helping me with difficulties along the way. They provided me with valuable feedback, insights, and suggestions that improved the quality of my research and thesis.

I would also like to thank Chris Hammerschmidth and Tom Catshoek (APTA Technologies) for their helpful advice and expertise on the topics and techniques used in this research. They generously shared their knowledge and experience with me and helped me overcome several technical challenges.

I am grateful to my friends that I have met during my study that helped me stay motivated during the meetings and our weekly get-togethers. They made my study journey more enjoyable and memorable.

Additionally, I want to thank Eye Security for the opportunity and allowing me to do an internship, providing me the help, data and computing resources needed to perform my research.

Last, but definitely not least, I would like to thank my parents and family for their unconditional love and support throughout my life. They always encouraged me to pursue my dreams and goals and gave me the strength and confidence to face any obstacles.

I hope you enjoy reading this thesis and find it interesting and informative.

Gerben Timmerman
Delft, September 2023

Abstract

This thesis offers a comprehensive exploration of log-based anomaly detection within the domain of cybersecurity incident response. The research describes a different approach and explores relevant log features for language model training, experimentation with different language models and training methodologies, and the investigation of the potential contribution of extra contextual features. The newly proposed approach is compared against an already implemented baseline in a finite-state classifier called FlexFringe, assessing their performance in detecting malicious anomalies across diverse datasets and hosts.

Key findings from this research underscore the importance of including human language for the generation of coherent clusters and a better performance of pretrained language models over models that were fine-tuned or built from scratch. Furthermore, the influence of clustering parameters on cluster quality proves to be crucial for cluster quality. Additionally, we gained insights into how extra contextual features are useful for log analysis.

In light of these findings, the study provides several recommendations for future research, including the expansion of the methodology to accommodate various log sources, the enhancement of preprocessing techniques, the integration of newer and more advanced language models, and the pursuit of efficient hyperparameter optimization. This work contributes to the continual advancement of log-based anomaly detection and its critical role in enhancing cybersecurity practices.

Contents

1 Introduction

1.1 Topic Introduction

As our society becomes increasingly digitalized, the security of our digital assets has become a growing concern. More and more sensitive data, such as financial records, medical records, and personal information, are stored online, making them vulnerable to unauthorized access. (Rajasekharaiah et al., 2020) Furthermore, as the internet of things (IoT) connects more and more aspects of our lives, the attack surface grows, and the risks increase even further. (Rajendran et al., 2019) However, as the technology that powers our digital world continues to evolve, so do the techniques that hackers and cybercriminals use to exploit it. This has resulted in an increase in cyberattacks, from phishing scams and malware infections to massive data breaches that expose millions of user records. (Hammouchi et al., 2019) As a result, the need for effective cybersecurity measures is more critical than ever, to safeguard ourselves, our businesses, and our society as a whole from the increasing threat of cybercrime. Securing our digital assets has become a complicated challenge, requiring a mix of technical expertise, robust security protocols, and constant vigilance.

One challenge is to detect whether the computers or network infrastructure of an individual or organization has been breached by an adversary. Whenever these environments are compromised, the adversaries who gained access perform various actions on those systems to achieve their malicious objective. Hence, it is crucial to monitor the activities on these systems. Attackers may leave many traces in the form of network packets, system calls, and warning or error statements to name a few, which are stored in a collection of logs. Logs are records that store events in a chronological order of activities, errors, and other noteworthy occurrences within a system or application. Some systems are critical to the functionality of an organization and can not afford to be down or have any failures. Using other methods like doing exhaustive searches or manual inspection for malicious behavior can be very taxing on a system, but also very intrusive. Monitoring by only using logs can address this issue, since logs can be easily collected without risking failures or demanding too much from the systems that need monitoring.

The application that this thesis focuses on is that we are trying to find actions of an adversary in a system or network environment by detecting anomalous behavior through the use of these logs. For this task, we intend to use machine learning in the form of language models that use these logs to learn to distinguish normal from anomalous behavior. This way we can detect suspicious behavior before and after an attack occurred. By proactively scanning for anomalous behavior, the likelihood of an attack spreading further through the system becomes much lower. Such models can alert security analysts that they have to be vigilant and tell them what to look for. Similar to incident prevention, these models can also be applied to incident response, which is what our research focuses on. The model could learn what normal behavior is happening in on the host's computer, only to point out where and what activities are abnormal. When an incident responder knows where to look, the attack can be resolved much faster and more efficiently.

1.2 Problem Relevance

As explained, logs can provide valuable information in detecting system compromise, but it is a difficult task to use them effectively to determine whether activities are anomalous or not. One solution is to let security analysts make the decisions based on information gathered through tools that analyze the logs and find attackers based on common malicious behavior (e.g., brute force attempt). A limitation of this method is that it does not detect unknown tactics, techniques, and procedures (TTP) and creates a lot of overhead and false positives for them to make precise decisions. Moreover, logs can be huge, complex, and may contain a large amount of irrelevant or redundant information, making it hard for a human to efficiently and accurately identify important events and patterns. Automating the detection process could make filtering through the logs more efficient in terms of speed and accuracy. However, this approach comes with its own new set of challenges. For example, the vast majority of logs are unstructured or semi-structured, meaning that they are not formatted consistently. They may contain different types of data, such as timestamps, error messages, and system metrics, making it difficult to develop a consistent approach for parsing and analyzing them. Additionally, they are often produced by multiple components, applications or services within a system, each of which may use a different logging library or format. Another problem is that these automated solutions tend to be sensitive and thus not 100% accurate in their decisions to mark certain events as anomalous, leading to many false alarms.

1.2.1 Motivation

While some of these automated solutions already achieve reasonable results, their performance in terms of detection rate and accuracy is not perfect yet. This research aims at further improving the performance of anomaly detection models.

A few of these related works that tried to tackle this problem are, for instance, LogBERT and LAnoBERT (Guo et al., 2021; Lee et al., 2021). These approaches utilize BERT-based masked language models to detect and classify whether certain log sequences are anomalous or not. LogBERT employs BERT to capture semantic representations of log messages, formulating anomaly detection as a masked language modeling problem, while LAnoBERT directly feeds raw log messages into BERT without log parsing, focusing on predictive probabilities. Another approach, ClusterLog (Egersdoerfer et al., 2022), addresses the challenges of noisy and ambiguous log sequences by clustering logs based on their semantic similarity and sentiment scores, simplifying the anomaly detection task using a sequence classification model on clustered log IDs. These will be further described in Section 2.6.

Another approach is proposed by APTA Technologies[1] and Eye Security[2]. APTA technologies specializes in computer-generated data analysis, helping cybersecurity incident responders come to findings faster and more reliably. Eye Security is a cybersecurity company that protects small-medium enterprises against digital threats. These two companies that have partnered up to create their own version of an anomaly detection model called FlexFringe and aim to create a less labor-intensive way of handling the aforementioned cyber incidents. These are also the companies where this thesis was written.

Their FlexFringe implementation uses Finite-state Automata to model normal versus anomalous behavior. (Verwer and Hammerschmidt, 2017) It uses a weighted word representation method, namely TF-IDF, to convert the contents of logs to numerical features in a vector for each log event. Similar log events are then clustered together, and the cluster IDs are then fed into a finite-state machine model that tracks the transitions between log events. These concepts will be further explained in chapter 2. While the foundation of this model already exists, its TF-IDF vectorizer method of information/feature extraction from the logs is a relatively simple method and could be replaced with a more complex method that better captures the relations between the features in the logs. What this thesis aims to find out is whether this feature extraction step can be performed through the use of so-called transformer models, which are the latest state-of-the-art in language modelling. We want to see how they compare with the current TF-IDF implementation in capturing meaning of the log contents.

Therefore, the overall goal of this thesis research is to develop a new preprocessing approach

[1]https://www.apta.tech/
[2]https://www.eye.security/

which aims to provide the anomaly detection model with features that better convey the meaning or semantics of a log event. Using this improved preprocessing pipeline, FlexFringe will hopefully be able to make better distinctions between behavior that should be labelled as normal and anomalous because of these improved features.

1.2.2 Stakeholders

The stakeholders that have an interest in this project are:

1. Industry professionals;
2. Researchers in academia;
3. End-users of the implementation.

The first group of stakeholders represent the security industry. These consist of Security Analysts and Incident Responders before/after an attack or breach is detected, by having a proper effective anomaly detection model and solution in place, they could save valuable time by not wasting their efforts on false alarms raised by the detection system. If the features and models can also inform them of where the alerts are coming from and provide context, that could be even better. Additionally, there are also model designers that would like to know how to build a good performing information extraction model. The next group are researchers in academia. This group can study and implement the insights gained from this research and improve on them. By providing the implementation with new logs, they can further evaluate the robustness of the new approach / system and improve on its design and usability. Finally, there are the end-users within the environments of where the final solution will be deployed. These users have to be assured that their activities are being monitored while respecting their privacy. The anomaly detection process should not use any sensitive features that can be traced back to an individual.

1.3 Research Questions, Hypothesis and Contributions

The main goal of this thesis is to enhance the information extraction techniques from event log files to boost the performance of existing anomaly detection models. We propose to implement a transformer encoder language model (LM) that can capture the underlying context among various features from event logs, whose last hidden layer can serve as a feature vector for the creation of clusters, where the sequence of clusters can be used to train a finite-state automaton (FSA) model that detects anomalies. To accomplish this goal, this thesis addresses the following main research question with four sub-questions:

How can embeddings and language models capture useful representations of log files for analysis in Cybersecurity?

SQ1 *What features from logs are relevant to train LMs?*

SQ2 *Which LM and what training method are most suited for feature extraction?*

SQ3 *How does a transformer LM compare to a weighted word representation model as a feature extraction method in anomaly detection?*

SQ4 *Does adding additional contextual features improve anomaly detection?*

We want to investigate how we can best capture useful representations of log events. First, we want to find out what information is stored in the logs and which features are relevant to extract meaning from. Secondly, we want to test several models and training approaches to see which method achieves the best results. Thirdly, we want to compare the best new method against the current implementation to see if it leads to improved results. Finally, we want to explore whether the addition of extra contextual features can help the anomaly detection process.
To learn the language used in logs, a large amount of data is required to train and test the performance of the language models. This data has been provided by Eye Security and will be explored in chapter 3. All the code used to build the language models can be accessed on GitHub [3].

[3] https://github.com/gerbentimmerman

1.3.1 Hypothesis

The hypothesis is that the performance of anomaly detection improves using transformer-based language models as a feature extraction method over the baseline TF-IDF weighted word representation model. We argue that this method enables the extraction of a better representation using the context of features in logs, instead of relying on independent tokens in a log message, as is the case with TF-IDF.

1.3.2 Contributions

The main contributions of this thesis towards the field of anomaly detection are the following:

C1 An insight in the feature selection process and preprocessing steps of Windows event logs.

C2 A new feature extraction approach using transformer language models that are able to capture more complex relations between the log features and additional contextual features, leading to an improved anomaly detection method.

C3 A comparative study between four transformer models and a baseline TF-IDF vectorizer.

1.4 Outline

The thesis is structured as follows: to answer the research questions, it is important to know more about the theoretical background, which is presented in chapter 2. In chapter 3, a description of the data, preprocessing and materials is given. Chapter 4 provides a detailed overview of the methodology used to conduct this research, namely, the model build pipeline, training of the language models and evaluation method. An overview of the experimental setup is given in chapter 5. The results from the experiments and evaluation method will be presented in chapter 6. Key findings from the results and limitations will be discussed in chapter 7. Finally, in chapter 8, we offer a conclusion where we answer the research questions and provide suggestions for future research.

2 Background

This chapter provides introductory background information about logs, previous work in the area of Anomaly Detection, Language Models, Clustering and Finite State Automata. First, we explain more about the origins of (EVTX) logs and what information they can contain. Second, we explore the task of anomaly detection itself and how it will be applied to our study of finding anomalies in log files. The third section dives into Language Models, discusses what they are, gives a brief overview of progress over the years and elaborates on recent developments. The fourth section describes what clustering is and elaborates on the technique that we will be using in this research to group similar log events together. In the fifth section, we explore the technique behind FlexFringe, namely, finite-state automata and how they can be used to follow transitions of log events to find rare patterns. Finally, we show relevant works that have tried to tackle the same problem and highlight innovations that this research makes over those works.

2.1 Logs

Logs refer to files or records that contain a chronological record of events, actions, or messages originating from an operating system, software applications, networks, servers, firewalls, hardware infrastructure, etc. To give a definition in computing terms from Sharif (2021): "An event is any significant action or occurrence that's recognized by a software system.". The event is typically recorded in a special file called the event log. All operating systems and most applications generate event logs to store events related to the source that produces them. The way a system's or application's logs are populated depend on how the logging process is configured by a developer or system administrator. These event logs are essential for purposes such as troubleshooting, performance monitoring, root cause incident analysis, compliance and auditing.

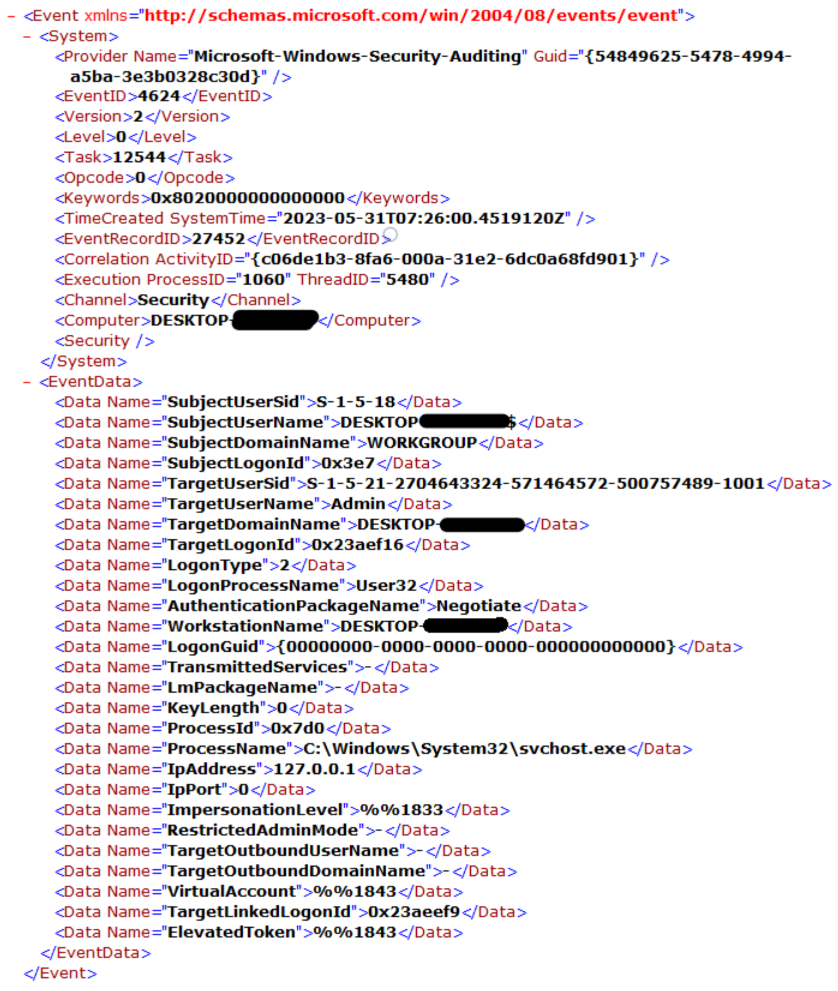
In this research, we aim to develop an approach that is able to catch deviating behavior that helps in the detection of adversarial events during a cyberattack. Therefore, we mainly focus on the root cause incident analysis, where the logs can be used to track malicious activities happening on a computer and to monitor the system, applications and infrastructure for security-related events. We will be working with Windows Event Logs to achieve this goal. A short introduction and overview of the Windows Event Logs structure and the information that they collect and contain are discussed next.

2.1.1 What are Windows Event Logs?

The Windows Event Log (EVTX) is a format used by Microsoft Windows to log information from a system running a Windows operating system, together with information produced by the applications running on the system. These logs are saved in so-called .evtx files. Windows Event Logs are often used to troubleshoot problems with the system or applications, to monitor their operations, and to possibly detect future problems that might occur. They contain a variety of different types of events and store them into a standard XML format which allows them to be shared with security experts, administrators and IT support analysts for investigative purposes. Additionally, it also allows them to be processed in a more structured manner, though the resulting format still very much varies between applications, components and services.

Each event log entry contains information about the event, where this information is then assigned to a so-called attribute in XML format. The entry contains data such as the event ID, event source (e.g., the system component or application that generated the event), level, date and time of the event, username, computer name, and other details about the event (Charter, 2008).

An example of the contents of an evtx file viewed inside the Windows Event Viewer can be seen in Figure 2.1. Here, information related to a user logon event has been saved as an entry in the security channel and is coming from the *Security.evtx* file.



```
- <Event xmlns="http://schemas.microsoft.com/win/2004/08/events/event">
  - <System>
      <Provider Name="Microsoft-Windows-Security-Auditing" Guid="{54849625-5478-4994-
        a5ba-3e3b0328c30d}" />
      <EventID>4624</EventID>
      <Version>2</Version>
      <Level>0</Level>
      <Task>12544</Task>
      <Opcode>0</Opcode>
      <Keywords>0x8020000000000000</Keywords>
      <TimeCreated SystemTime="2023-05-31T07:26:00.4519120Z" />
      <EventRecordID>27452</EventRecordID>
      <Correlation ActivityID="{c06de1b3-8fa6-000a-31e2-6dc0a68fd901}" />
      <Execution ProcessID="1060" ThreadID="5480" />
      <Channel>Security</Channel>
      <Computer>DESKTOP-[redacted]</Computer>
      <Security />
    </System>
  - <EventData>
      <Data Name="SubjectUserSid">S-1-5-18</Data>
      <Data Name="SubjectUserName">DESKTOP[redacted]$</Data>
      <Data Name="SubjectDomainName">WORKGROUP</Data>
      <Data Name="SubjectLogonId">0x3e7</Data>
      <Data Name="TargetUserSid">S-1-5-21-2704643324-571464572-500757489-1001</Data>
      <Data Name="TargetUserName">Admin</Data>
      <Data Name="TargetDomainName">DESKTOP-[redacted]</Data>
      <Data Name="TargetLogonId">0x23aef16</Data>
      <Data Name="LogonType">2</Data>
      <Data Name="LogonProcessName">User32</Data>
      <Data Name="AuthenticationPackageName">Negotiate</Data>
      <Data Name="WorkstationName">DESKTOP-[redacted]</Data>
      <Data Name="LogonGuid">{00000000-0000-0000-0000-000000000000}</Data>
      <Data Name="TransmittedServices">-</Data>
      <Data Name="LmPackageName">-</Data>
      <Data Name="KeyLength">0</Data>
      <Data Name="ProcessId">0x7d0</Data>
      <Data Name="ProcessName">C:\Windows\System32\svchost.exe</Data>
      <Data Name="IpAddress">127.0.0.1</Data>
      <Data Name="IpPort">0</Data>
      <Data Name="ImpersonationLevel">%%1833</Data>
      <Data Name="RestrictedAdminMode">-</Data>
      <Data Name="TargetOutboundUserName">-</Data>
      <Data Name="TargetOutboundDomainName">-</Data>
      <Data Name="VirtualAccount">%%1843</Data>
      <Data Name="TargetLinkedLogonId">0x23aeef9</Data>
      <Data Name="ElevatedToken">%%1843</Data>
    </EventData>
  </Event>
```

Figure 2.1: Example of the log contents related to a user logon on a Windows system in XML format.

Events are written to the event logs by providers and are coming from so-called channels. According to nxlog (2022) a provider can be a service, driver, or program that runs on the computer. A channel is a stream of events collected from several providers and they are saved in an evtx file that corresponds to the channel name. Channels are organized into two groups: 'Windows Logs' and 'Applications and Services Logs'. The first group 'Windows Logs' are events logged by the Windows operating system and can only contain the following five channels below. The second group 'Applications and Services Logs' consist of different channels created for individual applications and components.(SolarWinds, 2023; Yasar and Gillis, 2023)

- **Application:** These are events generated by applications running on the system, such as errors or warnings related to the operation of the application.

- **Security:** These are events related to security-related activities on the system, such as logon attempts, failed logon attempts, and other security-related activities.

- **System:** These are events related to the operation of the Windows operating system itself, such as system startup, shutdown, and configuration changes.
- **Setup:** Contains events that occur during the installation of the Windows operating system. On domain controllers, this log will also record events related to Active Directory.
- **Forwarded Events:** Consists of event logs forwarded from other computers in the same network.

Channels are organized into two main types: Direct and Serviced. Direct type channels, which are disabled by default, collect a large volume of logs and cannot be shared with other programs and systems. On the other hand, Serviced type channels collect a low log volume and they can be shared with other programs and systems (nxlog, 2022). To make an even better distinction, these types are split up into four subtypes: Admin (Administrative), Operational, Analytic, and Debug. Table 2.1 shows an overview of the channel structure and their corresponding channel types. Each channel subtype has an intended audience, which determines the type of events that you write to the channel. Admin type channels support events that are intended for administrators, end users, and support personnel. Events written to the Admin channels should have a well-defined solution, on which the people reading them can act. Operational type channels should produce events that are used for troubleshooting. Analytic type channels produce events that are published in high volume and they describe program operations. Debug type channels intended for developers to be used for debugging purposes (Karl-Bridge-Microsoft, 2021).

Channel groups	**Channels**	**Channel type**
Windows Logs	Application	Administrative (serviced)
	Security	Administrative (serviced)
	Setup	Operational (serviced)
	System	Administrative (serviced)
	Forwarded Events	Operational (serviced)
Applications and Services Logs	Microsoft-Windows-Powershell/Admin	Administrative (serviced)
	Microsoft-Windows-Powershell/Operational	Operational (serviced)
	Microsoft-Windows-SMBServer/Audit	Analytic (direct)
	Microsoft-Windows-SettingSync/Debug	Debug (direct)
	(And many more provider-defined channels)	

Table 2.1: Overview of the channel structure and their types. Adapted from nxlog (2022).

More details on which contents of the Windows Event Logs are used as features in our approach will be elaborated in Section 3.2.1.

2.2 Anomaly Detection

In this section, we explore the task of anomaly detection and how it can be applied to the cybersecurity domain. Additionally, we provide some adversarial examples and elaborate on what can be learned from these.

2.2.1 Definition of Anomalies and the Anomaly Detection Task

According to Merriam-Webster (2023), an anomaly is "A deviation from the common rule". In the context of data analysis, anomalies refer to data points, events, or patterns that deviate significantly from the expected or normal behavior of a dataset. These can be values that are much larger or smaller than the average, unexpected patterns or trends, or events that are rare or unusual. Anomalies also known as outliers or novelties and can occur due to various reasons such as measurement errors, data corruption, unusual conditions, or undesired behavior. There are several types of anomalies (Kampakis, 2022):

- **Point anomalies**: A single observation or data point that deviates from the expected pattern, range, or norm in relation to other data points. In other words, the data point is unexpected.

- **Collective anomalies**: Happens when looking at single data points in isolation appear normal, but when observing multiple groups of these data points they show unexpected patterns, behaviors, or results.

- **Contextual anomalies**: Occurs when a data point may appear normal when considered on its own, but becomes anomalous when considered in the context of other data points.

The task of anomaly detection is to identify these data points. They may represent important information or insights that could indicate potential problems or threats. The task usually involves preparing and preprocessing the data, selecting an appropriate anomaly detection algorithm or technique and defining a threshold or criteria for identifying anomalies, and at last analyzing and interpreting the results. Various techniques can be used for anomaly detection, including statistical methods, machine learning algorithms, and deep learning models.

2.2.2 Application in Cybersecurity

The identification and analysis of anomalies can be used for various applications, such as system monitoring, and quality control to ensure that a system or application is running or functioning in the way it should be. Some examples of applications that are related to the Cybersecurity domain are fraud detection, malware detection, intrusion detection and incident response. Anomaly detection can also proactively help to identify potential cyber threats and prevent security breaches. By detecting anomalies in network traffic, user behavior, system activity, or financial transactions, security teams can take appropriate action to mitigate the risks of cyberattacks and protect their organizations from potential damage. For this thesis, we are mainly focussing on Incident Response (IR). IR happens when a person or organization has been compromised by an attack and takes action to assess, respond to, and mitigate cyber threats, either by themselves or via an external party like EYE Security. A well-thought-out IR plan combined with the proper tools are important for organizations to protect and reduce the damage dealt to their data, systems, finances and reputation.

For this application, having an effective anomaly detection tool can prove to be essential in the case of IR to help find the root-cause of a compromised system. These automated tools can provide suggestions of activities that look strange and advise the analyst on where to look, which can help in a faster resolvement of the threats. Using the logs that contain clues of potential malicious actions, our goal is to find these anomalous activities performed by adversaries on a particular system.

2.2.3 Adversarial Examples

To illustrate the type of behavior that we are trying to detect, we provide two examples below. Of course, in theory we would like to catch all adversarial behavior even though we do not know what to look for. If an automated tool can detect this without too many false positives, that would be ideal.

A concrete example is the modification of permissions on an object by an unauthorized user. This action generates Event ID 4670 in the Security.evtx file. This event indicates that someone has changed the permissions on a file, folder, registry key, or other securable object. An analyst may want to know what object was modified, what permissions were changed, and who made the change. This could indicate an attempt to gain unauthorized access or tamper with sensitive data (Bradley, 2020). In our second example, we can check for the occurrence of event ID 1102 which clears the Security logs regardless of any audit policy or status. Besides attackers aiming to cover their tracks, it is unlikely that a 'normal' user would intentionally delete these logs. Looking into the user that cleared the logs can help identify whether that account was compromised or not (Bradley, 2020).

Of course these events on their own are already interesting enough to suggest analyst to look into them, but we also aim to do is detecting rare malicious transitions. An example of rare transitions between events that a 'normal' user would not do is the execution of a program followed by the attempts at modification of permissions on that same program or object. A 'normal' user is unlikely to have reasons to show behavior like this.

2.3 Language Models

In this section, we provide more information about the language models that are used in this research to extract information from the event logs. First, we tell more about the origin and idea behind these models and why we want to use them. Then a short history of advancements of language models is provided, where we build up to the current state-of-the-art techniques and models, namely, transformers. Here, we explain how they are designed and what recent models we will be using in this thesis.

2.3.1 What are Language Models?

Language models are computational models that are trained to learn and capture the patterns and structure of (human) language. They can be used for a wide range of natural language processing (NLP) tasks such as text classification, language translation, sentiment analysis, question-answering, summarization and more. They tend to be based on statistical or machine learning techniques and are trained with data from large corpora such as a collection of books, webpages or social media posts. Language models were initially designed to learn relations in human language, but we can also use them to learn the underlying structure in other forms of languages, like the log language that we will be using in this research. Instead of training with data from human sources, we will be training with structured computer-generated log data to train the models. The end goal is to generate a proper numerical representation of the information contained in the log.

2.3.2 History of advancements in LMs

Language models have a long and interesting history, dating back to the mid-20th century with the development of early computer programs that could process and generate digital human language. It has been one of continuous development and improvement, driven by advances in computer technology, making it more accessible and through peoples' production of large quantities of text data.

Starting back in the 1950s, researchers in the field of artificial intelligence (AI) were interested in developing computer programs that could process and generate human language. Examples of these early programs are ELIZA, SHRDLU, or LUNAR. Additionally, methods were developed early statistical models for language processing, including the Markov Chain and Hidden Markov Model (Baum and Petrie, 1966). These models were based on the idea that the likelihood of a word occurring in a language could be estimated based on the frequency of its occurrence in a large text corpus.

In the 1970s to 1990s, the wide availability of personal computers and large amounts of text data allowed for the development of more sophisticated statistical language models. During this time, n-gram models became popular, which represented a significant improvement over earlier models. N-gram models represent language as sequences of N consecutive words, and estimate the probability of a word occurring in a language based on its co-occurrence with previous words in the sequence. Furthermore, Sparck Jones (1972) introduced the concept of inverse document frequency (TF-IDF), which can be used to measure the relevance of terms in a document.

During the 2000s to 2010s, introduced the rise of Neural Networks and Deep Learning techniques (Bengio et al., 2000). With the arrival of the internet and the availability of even greater amounts of text data, researchers continued to develop more advanced language models. The rise of deep learning and neural networks in the 2010s also led to a new generation of language models, including the Recurrent Neural Network (RNN) and the Transformer architecture with Bidirectional Encoder Representations from Transformers (BERT) and Generative Pre-trained Transformer (GPT) (Rumelhart et al., 1986; Devlin et al., 2018; Radford et al., 2018). A shortcoming of RNN models is that they are unable to capture long-term dependencies due to the vanishing gradient problem, transformer models are able to capture these dependencies in language and performed well on tasks such as text generation, translation, and sentiment analysis.

Since the 2020s, there have been advancements in Pre-trained Large Language Models (LLMs), such as GPT-3, GPT-4 and (Open-)Llama (Brown et al., 2020; OpenAI, 2023; Geng and Liu, 2023; Touvron et al., 2023). These are very large pre-trained models with hundreds of billions or trillions of parameters. They have been trained on enormous amounts of text data from various sources and languages. The development of these models has led to many new applications and advances in areas where language modeling can play an important role. For example, domains such as search engines, healthcare, robotics, and software development. Some applications of LLMs include text generation, question-answering, code-generation, and more.

2.3.3 Weighted word representation models

Weighted word representations such as term frequency (TF) and term frequency-inverse document frequency (TF-IDF) are used to represent text data as weighted numeric vectors. TF measures the frequency of a term in a document. It assigns higher weight to words that appear more frequently in the document. In Formula 2.1 the formula for TF is shown. $TF_{t,d}$ is the term frequency of term t in document d, $f_{t,d}$ is the frequency of term t in document d, and the denominator is the sum of the frequencies of all terms in document d.

$$TF_{t,d} = \frac{f_{t,d}}{\sum_i f_{i,d}} \tag{2.1}$$

TF-IDF was introduced by Sparck Jones (1972). It assigns higher weight to words that appear frequently in a document and infrequently in the corpus, which is the collection of documents. The weight is lower if it occurs less frequently in the document, or if it occurs in many documents. It is at its lowest point if it occurs in almost all documents. It is calculated as the product of the term frequency and inverse document frequency. In Formula 2.2, $TF_{t,d}$ is the term frequency of term t in document d, and IDF_t is the inverse document frequency of term t in the corpus. The IDF_t is calculated as shown in Figure 2.3. Here, N is the total number of documents in the corpus or dataset, and n_t is the number of documents that contain term t. The log function is used to dampen the effect of rare terms.

$$TFIDF_{t,d} = TF_{t,d} \times IDF_t \tag{2.2}$$

$$IDF_t = \log \frac{N}{n_t} \tag{2.3}$$

2.3.4 Transformer models

The transformer is a neural network model that uses a mechanism called self-attention to process sequential data, such as natural language or speech. Self-attention enables the model to attend to different segments of the input sequence and learn their interdependencies. For instance, given a sentence as input, the model can identify the important words, their relations, and their irrelevance. Unlike recurrent models, such as RNN, LSTM and GRU, that process the input sequence sequentially, the transformer can process the entire sequence in parallel, resulting in faster and more efficient computation. The transformer architecture was first proposed by Vaswani et al. (2017) and has been widely applied to various natural language processing tasks, such as machine translation, text summarization, question answering, sentiment classification, and more.

A simplified overview of the transformer architecture is shown in Figure 2.2. We will explain a general overview of the main components present in the transformer architecture. A transformer can be divided into two parts: an encoder and a decoder part. These parts can be further divided into a total of seven steps, listed below: (Menon, 2023; Ankit, 2022).

1. **Inputs and Input Embeddings:** Here, the input text sequence is tokenized and transformed into numeric vectors called embeddings that can be processed by the transformer. An embedding captures some semantic and syntactic information about the token.

2. **Positional Encodings:** Positional encodings provide information about the position of each token in the sequence. This way, the models can better understand the context through the order of words. Without them, each word would be treated as independent of each other and generating grammatically correct sentences would be much harder.

3. **Encoder:** The encoder takes the input sequence of embeddings that have been added with positional encoding from the previous components and passes it through a stack of N encoder layers, each consisting of two sub-layers: a multi-head self-attention layer and a feed-forward network layer. The encoder takes the input embeddings and applies self-attention to capture the dependencies between the input tokens. The encoder output result is a high dimensional representation of the meaning and structure of the input sequence.

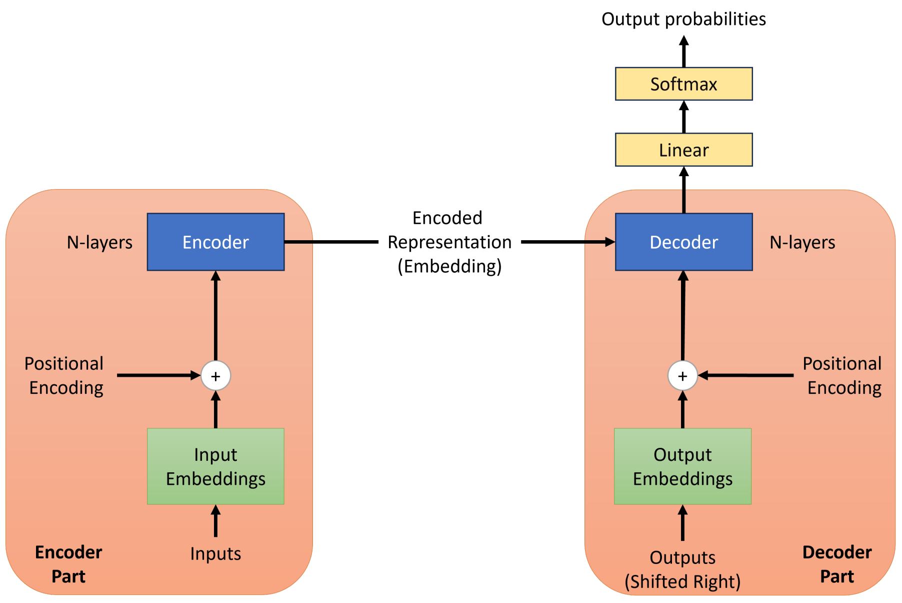


Figure 2.2: Overview of the transformer architecture's main components.

4. **Outputs (shifted right):** The outputs are sequences of tokens that are generated by the decoder. The outputs are shifted right by one position, so that each output token is predicted based on the previous output tokens and the encoder output. The outputs are also masked to prevent the decoder from attending to future tokens.

5. **Output Embeddings:** The output embeddings are learned vectors that represent each output token in a high-dimensional space. The output embeddings are added to positional encodings that indicate the position of each output token in the sequence.

6. **Decoder:** The decoder is composed of N identical layers, each consisting of three sub-layers: a masked multi-head self-attention layer, an encoder-decoder attention layer, and a feed-forward network layer. The decoder takes both the encoder output and the output embeddings as input. It then applies masked self-attention on the output embeddings to capture the dependencies between the output tokens. The encoder-decoder attention layer allows the decoder to use the encoder output and learn the alignment between the input and output sequences.

7. **Linear Layer and Softmax:** The linear layer is a fully connected layer that projects the decoder output to a V-dimensional space, where V is the size of the vocabulary. The softmax layer is a normalization layer that converts the linear output to a probability distribution over the vocabulary. The linear layer and softmax can be seen as a shared classifier that predicts the next token given the previous tokens.

An example of a transformer application is machine translation. For example, if the task is to translate from Dutch to English, the encoder would take a Dutch sentence and transform it into a representation that captures its meaning and structure. The decoder would then take that representation and generate an English sentence that preserves the meaning and structure of the original sentence. In this thesis, we are using so-called encoder models that only use the encoder part of the Transformer architecture. These models are better at retrieving a high dimensional representation of an input text by using the above-mentioned self-attention mechanisms to capture the text's semantic and syntactic features, such as the topic, the sentiment, the style, or the structure. These features can be to group texts that share common characteristics together into clusters, which is what we are planning to do with the log events.

Logs are a form of data that are not typically used in human language, and thus pose a challenge for natural language processing (NLP) systems that are trained on natural language spoken by humans. Logs can be seen as a whole different language that requires specialized models to analyze and understand. Training language models on log data could be beneficial to capturing their meaning. Therefore, one of the research questions we address in this thesis is how different types of pre-trained language models compare against each other and against models that are trained from scratch or fine-tuned on log data.

2.3.5 RoBERTa Transformer

RoBERTa is a language model that is based on BERT, but improves over it by using more data and better training methods. RoBERTa stands for Robustly Optimized BERT Approach. Like BERT, RoBERTa is a transformer-based language model that uses self-attention to process input sequences and generate contextualized representations of words in a sentence.

One key difference between RoBERTa and BERT is that RoBERTa was trained on a much larger dataset and using a more effective training procedure. RoBERTa used 160 GB of text data from various sources, such as books, news, web pages, and Wikipedia. RoBERTa also used larger batch sizes, larger learning rates, more training steps, and more advanced optimization techniques than BERT. These changes allowed RoBERTa to learn more from the data and achieve better performance on downstream tasks.

Another difference between RoBERTa and BERT is that RoBERTa removed the next sentence prediction objective that BERT used for pretraining. RoBERTa found that this objective was not helpful for learning general language understanding and could be replaced by longer sequences of text. RoBERTa also used a byte-level version of BPE as a tokenizer, which can handle more languages and characters than the character-level version that BERT used.

Our plan in this research will be to experiment with a sentence similarity optimized language model and the RoBERTa transformer architecture, of which we will use three variations: creating a model from scratch, using the pre-trained as is, and thirdly fine-tuning the pre-trained model. In Section 4.1.2, we elaborate more on the choice for these models and approaches.

2.3.6 TF-IDF vs. Transformer models

To show the difference between the two approaches, we compare the two methods using the following example sentence:

> "*Error 101: Connection failed - The user could not be authenticated on host.*"

In this sentence, TF-IDF will assign weights to words based on their importance in a document relative to a larger corpus. For this sentence, TF-IDF will recognize terms like ”Error” ”Connection”, ”failed”, ”user” ”authenticated” and ”host” as significant due to their uniqueness in the document compared to common words. It represents the sentence based on these weighted terms, providing a simplified yet interpretable representation.

A Transformer model, in our case RoBERTa, processes the text by tokenizing it and generating contextual embeddings for each token, considering their context within the sentence. This continues until we reach the end of the sentence, after which we are left with a sentence embedding. In this sentence, a transformer will understand the connections between ”Error” and ”101”, ”Connection” and ”failed”, and ”user”, ”authenticated” and ”host” as well as their long range connections, capturing the complex relationships in the text.

TF-IDF provides an efficient and interpretable representation, but a shortcoming is that it does not capture the relationship of words to each other. Roberta performs better in comprehending the context and nuances of the sentence, making it potentially more suitable for our task of understanding a log event. However, it comes at the cost of increased computational demands.

2.4 Clustering

This section dives into the underlying concept of clustering techniques. Clustering is an unsupervised machine learning technique that involves grouping similar data points together based on some defined similarity metric. Clustering is relevant to this research, because after retrieving the embedding vectors representing a log event through the language models, we want to group similar log events together to reduce the granularity and complexity of event types to track in our sequence model.

2.4.1 Mini-batch K-means

The current approach with TF-IDF uses a clustering method called Mini-batch K-means (Sculley, 2010), The Mini-batch K-means clustering method is a variation of the K-means algorithm (Steinhaus et al., 1956; MacQueen, 1967). Like the original K-means algorithm, Mini-batch K-means is a clustering algorithm that partitions a dataset into a specified number of clusters based on the similarity of their data points. However, this algorithm was chosen over the original K-means algorithm because Mini-batch K-means is designed to be more efficient in terms of speed and consumption of resources when handling large datasets. K-means processes the entire dataset in each iteration, while Mini-batch K-means processes only a small subset (mini-batch) of the dataset.

It takes the following steps to create, update and return the final clusters:

1. Initialize the cluster centroids: First, the algorithm begins by randomly selecting a set of k initial centroids from the data points.
2. Select a mini-batch of data from the dataset.
3. Assign each data point to a centroid: For each data point in the mini-batch, the algorithm calculates the distance to each of the k centroids and assigns it to the closest centroid.
4. Update the centroids: The algorithm calculates the mean of the data points assigned to each centroid and moves the centroid to the mean.
5. Repeat steps 2 to 4 until convergence: The algorithm continues to iteratively assign data points to centroids and update the centroids until there is no further improvement in the clustering.

2.4.2 DBSCAN

The clustering method used for our new approach is called DBSCAN which stands for Density-Based Spatial Clustering of Applications with Noise by Ester et al. (1996). It is a density-based clustering algorithm that works on the assumption that clusters are dense regions in space separated by regions of lower density. It groups these 'densely grouped' data points into the same cluster and marks isolated points as outliers or noise, as illustrated in Figure 2.3.

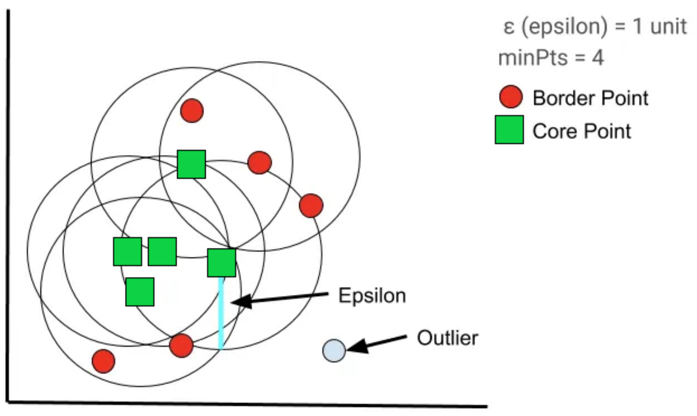


Figure 2.3: Illustration of the DBSCAN clustering classification process. Figure taken and adapted from Mysiak (2020)

The algorithm has two parameters: epsilon and min_samples. Epsilon specifies the radius of a neighborhood around a point (measured by the Euclidean distance) and min_samples specifies the minimum number of samples required to form a dense region. A point is classified as a core point if it has at least min_samples points within its epsilon neighborhood (including itself). A point is seen as a border point if it is not a core point and if it is in the epsilon neighborhood of a core point. A point is classified as a noise point if it is neither a core point nor a border point (Mysiak, 2020).

It takes the following steps to create and return the final clusters:

1. The algorithm randomly picks a data point and checks if it is a core point.

2. When a core point is picked, each point within its neighborhood is checked to see whether they are core points as well. If the criteria are met, these points are classified as core points and the cluster will expand with those new core points. If a point does not meet the number of min_samples requirements, it is classified as a border point of the cluster.
3. Repeats step 1 and 2 until all points are visited.
4. At the end, noise points are those that are not assigned to any cluster.

An example of the DBSCAN clustering process is shown in Figure 2.3, where the core are classified if the number of samples (including itself) within the reach of epsilon is larger or equal to 4. If the number is lower than 4 but still connected to a core point, the data point will be classified as a border point. Finally, if a point is not connected to any neighborhood of another point, it is considered an outlier.

2.5 Finite-state Automata

In this section, we will elaborate on the nature of the FlexFringe classification model that is used to find anomalies. Finite-state automata (FSA), otherwise called finite-state machines (FSM), are logical models describing these traces. They are models of behavior composed of a finite number of states, transitions between those states, and actions. A state consists of information from the past, while a transition specifies a change from one state to another and is triggered by a condition that must be met. An action is a definition of a task to be executed at a specific time. (Zhang et al., 2009)

FSA can be used for anomaly detection by modeling the normal behavior of a system and detecting deviations from that behavior. The basic idea is to create an FSA that represents the expected sequence of events or states in the system under normal conditions, and then use this model to identify anomalous behavior. To build an FSA for anomaly detection, one needs to first define what constitutes normal behavior. This can be done by analyzing the system logs, data collected from sensors, or other sources of information, and identifying patterns and sequences of events that are common under normal conditions. Once the normal behavior is understood, an FSA can be constructed to represent this behavior.

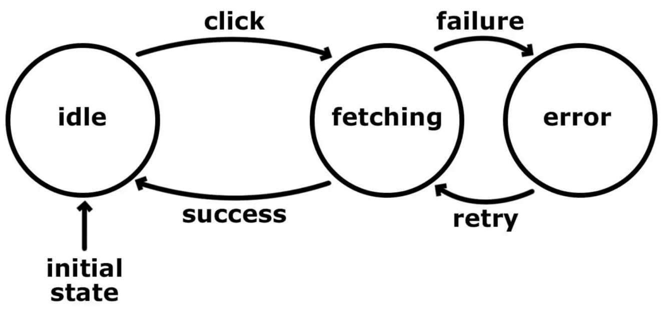


Figure 2.4: Example of a Finite-state Machine (Na, 2021)

The FSA is typically constructed as a directed graph, as can be seen in Figure 2.4 each node represents a state and each edge represents a transition between states. The nodes are labeled with the events or observations that correspond to that state, and the edges are labeled with the conditions under which the system transitions from one state to another. For example, an edge might be labeled with a particular input or output signal, or with a threshold value for a sensor reading.

When new events are recorded, the FSA is traversed. If the system deviates from the expected sequence of events, an alarm can be raised to signal an anomaly. This can be done by checking whether the observed events or observations match the labels on the edges of the FSA, and if not, whether they fall within some acceptable tolerance.

The motivation of using FSA models is that they enable us to find both rare states and rare transitions between states. This can help to determine whether some states or consecutive actions are anomalous, instead of only judging from single actions.

2.6 Related Work on Anomaly Detection using Language Models

In this section, we review several important and recent works in the field of log anomaly detection using language models. We also identify the main challenges and limitations of the existing methods, and explain how our work addresses them, while at the same time highlighting what separates our work.

2.6.1 LogBert

One of the recent and influential works in the field of log anomaly detection is the paper by Guo et al. (2021), who introduce LogBERT, a novel framework that leverages the power of BERT (Devlin et al., 2018), a pre-trained language model for natural language processing, to learn semantic representations of log messages and detect anomalies. Unlike previous methods that rely on manual feature engineering or statistical analysis, they treat log data as natural language text and apply BERT to capture the contextual and syntactic information of log messages. They formulate the anomaly detection task as a masked language modelling problem, where they first preprocess the log data by extracting sequences of log keys, which are abstract representations of the events or attributes in the log. Then, they randomly mask log keys and use BERT to predict the most likely candidates to fill in the blanks. If the observed value of a masked log key is among the top-k predicted candidates, it is considered normal; otherwise, it is deemed anomalous. Furthermore, they define a threshold for the number of anomalous log keys in a log sequence, and label the whole sequence as anomalous if it exceeds that threshold. They evaluate their framework on three public datasets: HDFS, BGL, and Thunderbird, and demonstrate that it achieves superior performance compared to the state-of-the-art baselines.

2.6.2 LAnoBERT

Another approach that applies the BERT masked language model to log anomaly detection is presented by Lee et al. (2021). Their proposed model, LAnoBERT, is similar to LogBERT in that it uses BERT to predict the masked log keys and compare them with the observed values. However, unlike LogBERT, LAnoBERT does not require a log parser to preprocess the log data. Instead, it directly feeds the raw log messages to BERT and learns the patterns from various log sources without needing a predefined log language structure. Moreover, LAnoBERT trains the model with only normal logs and assumes that abnormal logs will produce large errors and low predictive probabilities for the observed values. It uses the masked language modeling loss function as a scoring metric to measure the anomaly degree of each log key during the test process. If the score exceeds a certain threshold, the log key is considered anomalous. LAnoBERT also evaluates its performance on HDFS and BGL datasets and shows that it outperforms several existing methods and some supervised learning-based models.

2.6.3 ClusterLog

A different approach that tackles the problem of noisy and ambiguous log sequences is proposed by Egersdoerfer et al. (2022). They argue that the methods based on BERT Masked language model depend heavily on the quality of log sequences, which should be accurate and representative of the system's logic. However, in distributed systems and concurrent execution threads, there may be 'clock-skew' issues that lead to misaligned or incomplete log sequences. Moreover, a high granularity of unique log keys makes it difficult for a classification model to learn the log patterns. To address these issues, Egersdoerfer et al. (2022) introduce ClusterLog, which is a log preprocessing method that clusters individual logs based on their semantic similarity in order to reduce the complexity and noise of log sequences. The intuition is that by grouping similar logs together in a cluster, the log sequences become more regular and less variable, since there are fewer unique key identifiers and fewer combinations to learn. Their model consists of four steps.

The first step is the preprocessing step, where they remove excess characters and information from the raw log messages and keep only natural language text (log keys). The second step is the semantic embedding step, where they feed the log keys to a pre-trained language model that generates an embedding vector for each log key, representing its semantic information. Additionally, they append a sentiment score as an extra dimension to the embedding vector, based on the sentiment of the log key. The idea is that similar logs will have similar embeddings and sentiment scores, and will be close in the embedding space, while different logs will have different embeddings and sentiment scores, and will be far apart in the embedding space. The third step is the clustering step, where they apply a clustering algorithm to group the logs into similar clusters based on their embeddings and sentiment scores. They assign each log a cluster ID that represents its cluster membership. The fourth step is the anomaly detection step, where they feed a sequence of 10 cluster IDs to a two-layer LSTM classification model that predicts the probabilities of each cluster ID being the next one in the sequence. If the observed cluster ID is among the top k candidates with the highest probability, it is considered normal; otherwise, it is deemed anomalous. They evaluate their method on HDFS and BGL datasets and show that ClusterLog performs best with the addition of the sentiment feature and a variable number of candidates in the sequence classification model. It also outperforms other state-of-the-art models on both datasets and it performs more stable when training on a small dataset, proving that reduced granularity in log keys makes it easier to learn the log sequence patterns with less data.

2.6.4 Innovation over related work

Similar to LogBERT and LAnoBERT, our approach will also use masked language modelling to learn the log language. However, unlike the previous works that rely on a single language model architecture, we will explore the use of different and improved language model architectures such as RoBERTa and compare their performance on this task. Additionally, we will investigate whether the effect of training with log data related to our task is beneficial or not. We will also examine which training method (training from scratch, fine-tuning or using a pre-trained model) leads to the best results.

Secondly, our approach will use a different anomaly detection method. Instead of comparing the observed tokens with the top predicted candidates by the masked model as in LogBERT, we will use an approach similar to ClusterLog, namely, to reduce the granularity of the log keys by using the mean of all sentence token embeddings to generate sentence embeddings and cluster them based on their similarity. Then, we will use a finite-state automata model called FlexFringe to learn the common and rare states and transitions between the cluster ID sequences and detect anomalies based on their occurrences.

Thirdly, we will explore the possibility of adding other extra features to the log embeddings or next to the cluster IDs, similar to the sentiment feature used by ClusterLog. For example, we will take into consideration whether the event log was generated during work hours or not, or whether a new IP address, username or hostname occurred in the logs. These features may provide additional contextual information besides language to the model and can help an analyst make better and more informed decisions when investigating certain alerts.

Finally, in our approach, we will use windows event logs coming from endpoints of various hosts where malicious activities took place, which are from a different source from the logs present in the public datasets used by the other related works. These windows event logs have different characteristics and challenges that require different solutions.

3 Data & Materials

In order to build a language model, a large amount of training data is needed to make accurate predictions. First, an elaboration is given on the collection process of the log data from various hosts. Subsequently, the data preparation steps needed to train the language models are explained. At last, we dive deeper into the annotation process of our test set.

3.1 Collection

3.1.1 Data Origins

The data used in this research has been made available by Eye Security. This cybersecurity company closely monitors threats to systems and cloud environments. Additionally, they also perform Incident Response (IR) for clients that have compromised systems. The data that we are using to train the language models consists of multiple Windows EVTX logs that come from several of these compromised systems also called hosts. For this research, we were allowed to work with data coming from 73 unique hosts whose evtx files contain a total number of 42.453.253 logs. A more detailed view on the amount of logs per host is shown in Appendix A.1.

3.1.2 Conversion to Usable Format for Preprocessing

To get the content of the log messages into a usable format for processing, we are using a tool from Eric Zimmerman called: EvtxECmd[1]. This tool is a command line interface that can process evtx files and export them to CSV, JSON, or XML formats. It does this by using custom maps that define how to parse and display different event types. This tool extracts the features and information from all EVTX files present on a host and stores it into a single CSV file where each feature is represented in a column.

3.2 Data Preprocessing

During the preprocessing phase, we want to process the log events data and get a numerical representation of the text that is representative of the log events. First, we select the features from the logs that provide relevant information for detecting anomalies. After which we combine them into a string called the feature string. When providing the feature string to the model, the model's tokenizer will convert the text inputs to numerical tokens that can be processed by the model.

Additionally, we remove key identifiers from the feature string that are unique to a specific log event and clean them of any noise. The goal of taking these measures is to deliver the data in a more usable and generalizable format to make learning log language relations easier for the machine learning algorithms.

3.2.1 Construction of the Feature String

The first step is to select the features from the log events. These features are then concatenated into a string. To be more elaborate, a feature string is a sequence of characters that represents

[1]`https://github.com/EricZimmerman/evtx`

some aspect of the text, such as words, punctuation, capitalization, etc. The tokenizer can split the feature string into smaller units called tokens, which are then mapped to unique IDs. The language model then processes these IDs. When processing the generated CSV files with the features from EvtxECmd, we selected the features shown in Table 3.1 based on expert knowledge from security analysts.

Selected Features	Removed Features
EventId, Level, Provider, Channel, Computer, MapDescription, UserId, UserName, RemoteHost, ExecutableInfo SourceFile, Payload	RecordNumber, EventRecordId, TimeCreated, ProcessId, ThreadId, ChunkNumber, PayloadData1, PayloadData2, PayloadData3, PayloadData4, PayloadData5, PayloadData6, HiddenRecord, Keywords, ExtraDataOffset,

Table 3.1: Selected features from the CSV files generated by the EvtxECmd tool.

$$\begin{aligned} \text{SourceFile} + \text{Provider} + \text{Channel} + \text{EventId} + \text{Level} + \text{MapDescription} + \\ \text{Computer} + \text{UserId} + \text{UserName} + \text{RemoteHost} + \text{ExecutableInfo} + \text{Payload} \end{aligned} \tag{3.1}$$

Formula 3.1 shows the concatenation of the features into a specific order that improves readability and understanding of the contents of the string. We will now elaborate on the contents of each selected feature.

SourceFile Lists the name of the evtx file that contained the log event. This is a key feature, because it is used for identifying the source of each event record when processing multiple evtx files. When researching the logs' contents, we found that different meanings of the same EventId exist when browsing the various source evtx files. Therefore, we need to provide the language model with extra context of where these events were retrieved from.

Provider Represents the name of the application or service that generated the event record.

Channel The name of the event log channel where the event occurred. .

EventId The numeric identifier of the event.

Level The severity level of the event. This feature adds extra context to the log events, where the idea is that more severe log events are more important to investigate. The possible options are critical, error, warning, information and verbose types of log events.

Computer The name of the computer where the event occurred.

UserId The security identifier (SID) of the user associated with the event.

UserName The security identifier (SID) of the user associated with the event, formatted as DOMAIN\username. Not always present, but when it is, we need to include it.

RemoteHost The IP address, hostname, or other identifier of a remote host involved in the event. If a log contains events that contain a Remote Host which is unexpected, unauthorized or not trustworthy, it could indicate a malicious activity or a compromise.

ExecutableInfo The name and path of an executable file involved in the event. We split up the path into separate tokens, because it is interesting to know whether the execution of a specific file happens in a particular location or directory on the system, and this helps the language model to generalize across various systems. The name of the executable can help indicate whether the activities associated to the program are benign or show malicious actions.

MapDescription The description of the event provided by the custom map.

Payload The raw XML data of the event record or any additional information extracted by the custom map. The payload could contain information that might be valuable to understand the contents of the log. Since they are raw logs, they contain a lot of noise, which we need to filter out because it could disturb the language understanding process.

With the provider feature, we can observe if there are log events from a provider that is not expected or known to be installed on the system. Alternatively, we can check if an event log contains values that are expected but with abnormal frequencies or values, it could indicate a misconfiguration or a malicious activity taking place. This also holds for the Channel and EventId features.

Although the Computer, UserId and UserName will get anonymized, we still include them, represented by a tag, in the feature string whenever they are present in the log. This is done to provide the model with the context that those features are present, might be present later on in future data it encounters, or might not be present at all. Further elaboration on the anonymization process will be given in Section 3.2.3.

3.2.2 Data Cleaning of the Feature String

In terms of data cleaning, we rather not change much, since simple tokenization should be sufficient when dealing with text classification. Camacho-Collados and Pilehvar (2017) show that a simple tokenization typically works equally or better than more complex preprocessing techniques in neural networks. The paper mentions that even though complex techniques like lemmatization or stemming have been useful in conventional linear models as an effective way to deal with sparsity. (Toman et al., 2006; Mullen and Collier, 2004) Now, due to the generalization power of word embeddings, neural networks and transformer architectures are very capable of overcoming data sparsity. Additionally, the authors find that word embeddings trained on multiword-grouped corpora perform surprisingly well when applied to simple tokenized datasets. This multiword-grouping technique groups consecutive tokens (United States) together into a single token (United_States) to represent one meaning, instead of treating them as two separate words that each have a different meaning. Therefore, it is very useful to apply this technique to features that consists of names that indicate a program, service, file location etc. separated by spaces. It would be more meaningful to treat them as single words that convey a specific meaning, instead of handling them as separate instances.

In addition to the above techniques, we also have to deal with noise. Noise in log events refers to irrelevant or extraneous information that can make it more difficult for the language model to correctly identify the relevant information in the log. This can include things like timestamps, IP addresses, process IDs, and other details that are not directly related to the content of the log message. When a language model is trained on log events that contain noise, it can lead to the model learning incorrect or irrelevant patterns in the data. This can result in worse performance when the model is used to analyze logs in a real-world setting. Basically, we want to remove all unique identifiers of a log event, since these do not help to understand the meaning from the content. Therefore, we take several measures in the data cleaning process that attempt to remove noisy information.

The following cleaning measures are performed on the constructed feature string, and examples are shown in Table 3.2.

1. Multi-word grouping of single features that are separated by spaces
2. Replace dates with DATE tag
3. Replace timestamps with TIME tag
4. Replace some HEX values with HEX tag
5. Replace FLOAT values with FLOAT tag
6. Strip accents from characters and provide ASCII equivalent
7. Strip line breaks, quotations, brackets and whitespace characters
8. Unescape HTML text
9. Removal of noisy tags and information
10. Removal of words > 35 characters

Cleaning Measure	**Original Text**	**Cleaned Text**
Multiword grouping	Microsoft-Windows-User Profiles Service	Microsoft-Windows-User_Profiles_Service
Replacing values with tag	0x7613 9.5	<HEX> <FLOAT>
Stripping accents and whitespace	éxample: \t text \n	example: text
Unescape HTML text	<p>text</p>	text
Remove noise	{Info : "value"} \n	Info : value
Remove tokens with len>35	<LONGTEXT>	""

Table 3.2: Examples of steps taken in the data cleaning process

3.2.3 Anonymization of sensitive information

Since we are also dealing with sensitive data retrieved from specific computers and users, we also attempt to anonymize the data as much as possible. Anonymization, in the context of data processing, is the process of removing or obfuscating Personally Identifiable Information (PII) or sensitive information from the data before it is used for training. One of the reasons to anonymize the data before is that language models are trained on large amounts of text data, which often contains personal information such as names, addresses, and other identifying details that could potentially be retrieved. By anonymizing the data, the privacy of individuals can be protected. Secondly, we want to avoid bias in the models. It is not in our interest to let a particular name or IP address influence the decisions that the model has to make. Anonymization can help avoid bias in the language model by removing PII and sensitive information that could lead to unfair discrimination against certain groups of people or more applicable to our use case: computer names, usernames and IP addresses. If one wants to capture behavior related to specific usernames, IPs and computers, it will always be possible to add them later by fine-tuning the base-model to a specific host environment or customer.

Furthermore, anonymization plays an important role in compliance with regulations. Many countries have data protection regulations that require the anonymization of personal data before it is processed or shared. By anonymizing the data, organizations can ensure compliance with these regulations.

The following anonymization steps were performed to hide personal information within the logs:

- Replace names of computers with <COMPUTER> tag.
- Replace user IDs with <USERID> tag.
- Replace usernames with <USERNAME> tag.
- Replace IPv4 and IPv6 addresses with <IP> tag.

Examples of the anonymization measures are shown in Table 3.3.

Anonymization Measure	Original Text	Anonymized Text
Replace names of computers	SRV-Example, WKS-Example	<COMPUTER>, <COMPUTER>
Replace usernames	username1, username2	<USERNAME>, <USERNAME>
Replace IPv4 and IPv6 addresses	192.168.0.1, FE80:CD00:0:CDE:1257:0:211E:729C	<IP>, <IP>

Table 3.3: Examples of steps taken in the anonymization process

3.3 Deduplication of Validation Set

A validation set is a part of the data that is held out from training and used to measure the model's performance on new data and avoid overfitting. It is useful for tracking the learning process and it helps to verify if the model can handle unseen examples or if it is just learning the training data by heart. When creating the validation set for the training process, we shuffled the whole dataset and randomly sampled 0.1% to create a validation and test set, both having a size of 21.227 logs. During the research, we discovered that there could be duplicate log lines in the validation set. Evaluation with duplicate values can lead to a skewed result when judging the performance of the language models. When the model performs well on certain log lines and they make up a large portion of the validation set, the final results would look better than they actually are, compared to when the variation in log lines had an equal distribution in the validation set. Therefore, we had to get rid of duplicated and too similar looking logs in the validation set. We chose to implement a token-based Jaccard similarity score, which is a similarity metric ranging from 0 (not at all similar) to 1 (equally similar) (Jaccard, 1901). The metric indicates how similar two sets of words/tokens are, based on the ratio of their intersection and union. With a similarity threshold of 0.9, we only kept log strings that had a lower similarity than our set threshold, resulting in a final validation set of 1814 unique log events.

4 Methodology

In this chapter, we present the methodology used to extract information from the Windows event logs. We show the data pipeline and how the language models are created. At last, the training process of the models is described.

4.1 Model Build Pipeline

The model build pipeline consists of 6 phases and an overview is shown in Figure 4.1. The first and second phases are the collection of logs and log preprocessing, where the log contents get extracted and transformed into the right format needed for the language model. The third phase consists of the retrieval of the embeddings from the language model, which is a numerical sentence-based representation of the information stored in the logs. The next (fourth) phase possibly adds new key information as extra dimensions to the log embeddings. In the fifth phase, we apply clustering to the logs to reduce log granularity by grouping similar logs together in their own separate cluster. In the final sixth phase, we perform sequential analysis to detect any anomalous clusters or transitions between clusters or states.

4.1.1 Data Preprocessing

The first step in the data pipeline is the extraction of the log contents from the evtx files and put them into CSV format to be able to process them further as described in Section 3.1.2. After the evtx files have been extracted, we can start constructing the feature string that we will give to the language models to analyze. As shown in Section 3.2.1, the feature string consists of several important features like Provider, Channel, etc., retrieved from the log contents based on expert knowledge that we want the model to learn and understand to be able to form a better numerical representation (embeddings). Once the all feature strings have been created, they are stored in a clean dataset that we can load and process without having to go through the same construction and cleaning steps again.

However, in order to get the embeddings back from the models, we first need to convert the feature string into the right format for the language models to perform the masked language modelling training task as a pretraining objective. In this task, the data represents a sentence or sequence of text and some words are randomly masked. The language model then predicts these masked words based on the rest of the words present in the sentence. We will elaborate more on the task in the next Section 4.1.2.

To be able to do this, the following four steps are needed before feeding the data to our language models:

1. Apply byte-pair encoding to split words into subwords.
2. Add special tokens to mark sentence boundaries and masked words
3. Convert the tokens to their numerical form

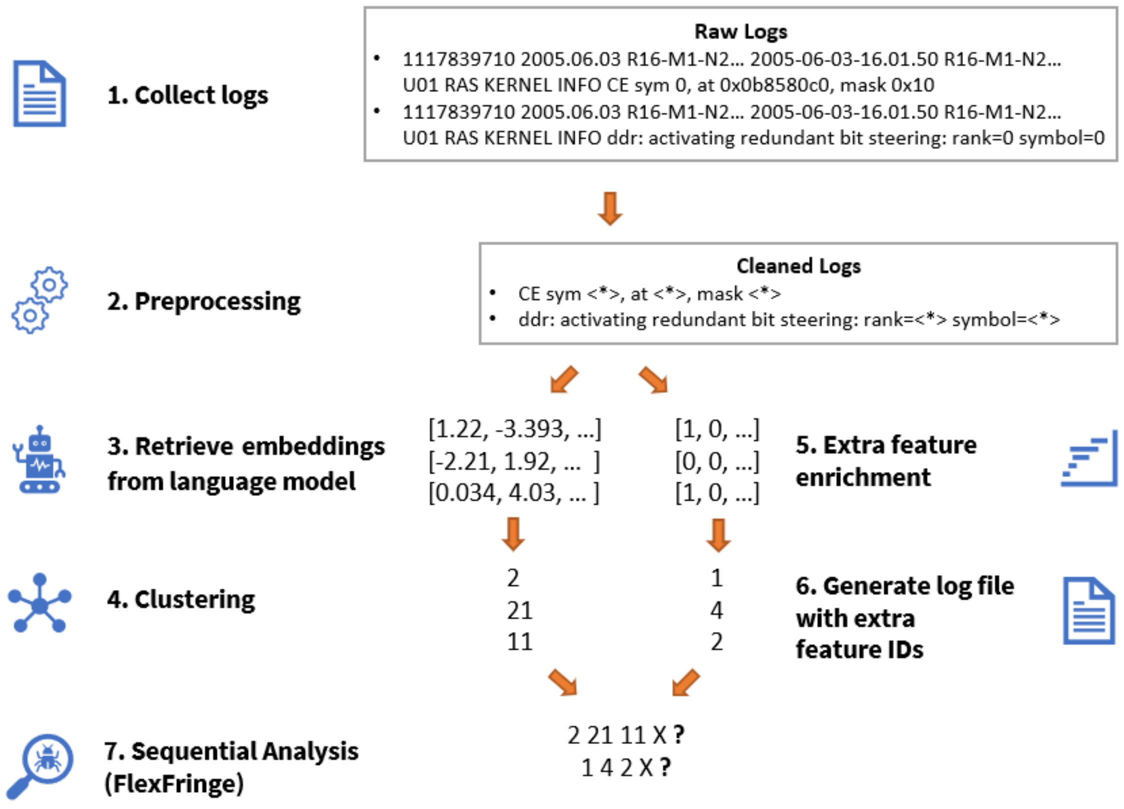

Raw Logs
1117839710 2005.06.03 R16-M1-N2... 2005-06-03-16.01.50 R16-M1-N2... U01 RAS KERNEL INFO CE sym 0, at 0x0b8580c0, mask 0x10
1117839710 2005.06.03 R16-M1-N2... 2005-06-03-16.01.50 R16-M1-N2... U01 RAS KERNEL INFO ddr: activating redundant bit steering: rank=0 symbol=0
1. Collect logs
Cleaned Logs
CE sym <*>, at <*>, mask <*>
ddr: activating redundant bit steering: rank=<*> symbol=<*>
2. Preprocessing
3. Retrieve embeddings from language model
[1.22, -3.393, ...]
[-2.21, 1.92, ...]
[0.034, 4.03, ...]
[1, 0, ...]
[0, 0, ...]
[1, 0, ...]
5. Extra feature enrichment
4. Clustering
2
21
11
1
4
2
6. Generate log file with extra feature IDs
7. Sequential Analysis (FlexFringe)
2 21 11 X ?
1 4 2 X ?

Figure 4.1: Illustration of the model build pipeline along with an example.

4. Truncate or add (dynamic) padding to create inputs of equal lengths

As a first step, we need to tokenize the sequences we feed to the model. A tokenizer splits text into smaller chunks called tokens which could consists of individual words, punctuation, numbers, but also phrases, sentences or complete paragraphs. Since we use the RoBERTa language model, we apply the same tokenization method used in the RoBERTa paper by Liu et al. (2019) called byte-pair encoding (BPE). BPE is a morphological or subword tokenization method created by Sennrich et al. (2015) that splits words into smaller units based on their frequency in a large training corpus. When creating the RoBERTa model, the authors used a clever implementation of BPE introduced by Radford et al. (2019). Instead of using Unicode characters as the base subword units, this implementation uses bytes. It consists of two parts: A token learner and a token segmenter. The token learner takes a raw training corpus and creates a vocabulary which the token segmenter can use to tokenize or break up the input sequences. The token learner starts with a base vocabulary of 256 characters (bytes) and iteratively merges the most frequent pair of characters until reaching a predefined vocabulary size. According to Liu et al. (2019): ”Using bytes makes it possible to learn a subword vocabulary of a relatively modest size (50K units) that can still encode any input text without introducing any unknown tokens.”. It treats spaces as part of the tokens and encodes them with a special character ’Ġ’ which allows the tokenizer to distinguish between words that are at the beginning of a sentence (without space) or not. An example can be found in Table 4.1. As shown in the table, the Ġ character is added to indicate the start of a new word.

Input	”This is a test to test tokenization for RoBERTa-base.”
Output	[’This’, ’Ġis’, ’Ġa’, ’Ġtest’, ’Ġto’, ’Ġtest’, ’Ġtoken’, ’ization’, ’Ġfor’, ’ĠRo’, ’BER’, ’Ta’, ’-’, ’base’, ’.’]

Table 4.1: Example of Byte Level BPE tokenizer

The second step involves the addition of special tokens to mark the start and end of an input sentence, to indicate which words are masked to the model, a padding token to make sure the input sequences have an equal length and an unknown token if it so happens that the tokenizer does not recognize a character. These tokens are listed in Table 4.2.

Special Token	Token value
Start of sequence token	<s>
End of sequence token	<\s>
Masked value token	<mask>
Padding token	<pad>
Unknown token	<unk>

Table 4.2: Special tokens added to the vocabulary of the tokenizer.

In the third step the tokens are converted to their numerical form represented by a token ID based on the vocabulary saved by the tokenizer which maps a token to an integer ID. The fourth step truncates or adds padding tokens to make the sequences equal to a specified length. This is necessary when training a language model, because we perform matrix operations that require fixed dimensions. Truncation happens when the total length of the tokens in a sequence exceeds a certain specified amount of tokens or the maximum length the model is able to handle. In the case of the RoBERTa-base model it is 512 tokens, but this can be increased if we were to take a larger language model.

Using the Hugging Face library, we can use model-specific tokenizers. These model-specific tokenizers are specifically created to easily convert the strings to a format suited to train their corresponding language models. (Wolf et al., 2020)

4.1.2 Creating the Language Models

Most pretrained language models are trained on natural language, and can be used to fine-tune on a more downstream task like text classification. Since we are dealing with log data which deviates a lot from normal language, the existing pre-trained models might not be suited for extracting information from this type of data. Therefore, we want to compare them against other newly trained language models, either from scratch or fine-tuned, to test whether they understand the relations between the information present in log files better. The transformer language models that we will use in the experiments are listed below:

- sentence-transformers/all-MiniLM-L6-v2 (pre-trained)
- Normal encoder model pretrained without fine-tuning (pre-trained roberta-base)
- Normal encoder model from scratch (roberta-base with randomly initiated weights)
- Fine-tuning a pre-trained model with new log messages (pre-trained roberta-base)

We compare four different models for generating sentence embeddings from log messages. The first model is a sentence similarity model called all-MiniLM-L6-v2 that has been optimized to transform a sentence (log) into a numerical vector (embedding) that captures its semantic information as accurately as possible. The other three models will be based on the RoBERTa-base model, which is a state-of-the-art transformer-based language model developed by Liu et al. (2019). This model has been trained on several large datasets and has a high level of linguistic understanding.

The all-MiniLM-L6-v2 model has been trained on a large 1B sentence pairs corpus, where they used a contrastive learning objective. In this objective a sentence is picked from the dataset and the model has to predict which sentence, out of a set of randomly sampled sentences, was actually paired with the sentence. We will use the pre-trained model as it is. Fine-tuning this model on log data could potentially improve its performance. Unfortunately, this is not an option for us since we do not have access to labeled sentence pairs. The second model is the RoBERTa-base model published by the model authors, without further retraining or fine-tuning steps. The reason for using this model is to see how the original model compares to the models what we will be training using log data. As our third model, we train a new RoBERTa-base model from scratch, where we randomly initialize the weights and parameters. In this case, the model has to learn the language patterns and context of log messages from scratch, without using any prior knowledge. The fourth model is a fine-tuned version of RoBERTa-base. Instead of training models from scratch, which cost a considerable amount of computing resources and time, it might be better to fine-tune these pre-trained models to a specific task with new data. A fine-tuned model is a model that has been trained on a large dataset and then adapted to a specific task or domain by using a smaller dataset related to that domain or task. In our case, the pre-trained model RoBERTa has been trained on a large corpus in a self-supervised fashion, but now we want it to be able to better adapt to our domain of understanding log contents. We adapt the pretrained model to our specific domain and task by using a smaller dataset of log messages. Fine-tuning allows us to leverage the prior knowledge of the pretrained model and adjust it to a specific task or domain by using a new, smaller dataset related to that domain or task.

In our case, pretrained RoBERTa-base possesses a high level of linguistic knowledge, as it has been exposed to several large datasets during its training. This prior knowledge could be beneficial for our task, if we can fine-tune the weights and parameters of this model to capture the specific language patterns and context of log files. The resulting output embeddings of these models could then reflect the language more accurately than those obtained from models trained from scratch. Models trained from scratch would have to relearn the semantic and syntactic information of words in their context, as well as the relationships between the information elements in the log messages.

We train the log-based models using the same method as the pre-trained models, which is through the masked language modeling task. In masked language modeling, a certain percentage of the input tokens (typically around 15%) are randomly selected and replaced with a special ”<mask>” token. The model is then trained to predict the original value of the masked tokens based on the context provided by the other tokens in the sequence. This enables the model to learn more complex and contextual relationships and dependencies between different parts of a sentence or text passage. In short, it allows models to learn to understand language in a more comprehensive and contextualized way. During training, the data collator will insert the “<mask>” tokens randomly for each batch, ensuring that the model sees different data in each epoch. After the training of the language models is completed, we can use them for inference. The models will produce an embedding for each log, which is a numerical vector that represents the semantic information contained in the log.

To evaluate the performance of different models for generating sentence embeddings from log messages, we conduct experiments with the four described models, namely, the sentence similarity model, the pretrained RoBERTa-base model, the RoBERTa-base model trained from scratch, and the fine-tuned RoBERTa-base model. Out of these four models, only two require training on our log data, while the other two are used as they are. By comparing the results of these models, we aim to gain insights into which approach is most effective for this task and to answer the second sub-question of our research: ”Which language model and what training method is best suited for feature extraction?”. If the fine-tuned or the pre-trained models perform equally well or better, there is no need to spend valuable time and computation resources to train the models from scratch. This will in turn reduce the environmental impact and costs, since training a fine-tuned model only uses a fraction of the resources and time compared to training models trained from scratch. To implement these complex transformer models, we use the Hugging Face library (Wolf et al., 2020), which provides a standardized and user-friendly way for working with various state-of-the-art language models.

4.1.3 Clustering of the log embeddings

When we have retrieved the embeddings for the log events, our next step is to convert them to states, which are represented by a single integer ID. We use clustering to group similar looking logs together, reducing the dimensionality and complexity of the data by finding groups of embeddings that share common features or patterns coming from the logs (Egersdoerfer et al., 2022). By calculating the distance between the log vectors, we aim to place similar logs in the same cluster represented by an integer ID based on their embeddings. We perform clustering on every single (security, application, system etc.) evtx file coming from a host. These states or event IDs are then provided to the classification model that attempts to recognize any anomalous patterns or activities.

To achieve this we chose to use a density-based clustering method called DBSCAN following the approach of Egersdoerfer et al. (2022). They provided several good motivations on why this algorithm is a good fit for our type of data. One of the reasons they specify is that we do not need to specify the number of centroids or clusters to the algorithm. This is a major disadvantage since it would mean that we will have to find an optimal number of clusters for every single user or host and evtx files that we want to analyze since the data can vary a lot between the different hosts and files. Having a fixed number of clusters introduces a risk that the algorithm groups logs together that are not similar. The classification algorithm will then get served an ID that does not provide a good representation of the event.

Another advantage comes from the second parameter in the model: min_samples When setting this parameter to 1, DBSCAN will not mark any of the outlier data points in the dataset as noise, but these will form their own unique clusters, allowing us to classify outlier events with a unique ID.

We found a very important third reason why choosing density-based clustering over centroid-based clustering is preferable. Centroid-based clustering methods, such as k-means, use the mean or average of the data points in each cluster to define the centroids. If there are many duplicate data points, they can distort the mean and influence the cluster assignment. This can result in suboptimal clusters that do not reflect the true structure of the data. For example, a duplicate point can pull the centroid closer to itself and away from the other points in the cluster. Density-based methods like DBSCAN are more suitable for this kind of data because they do not use centroids or means to define clusters. Instead, they use local density estimates to find clusters of high-density regions that are separated by low-density regions, which is why we think it is better suited for our data.

We also looked into using HDBSCAN (McInnes and Healy, 2017), which is an improved version of DBSCAN that finds clusters of varying densities (unlike DBSCAN) and is more robust to parameter selection. The method does not require any extensive parameter tuning and returns a well performing clustering out-of-the-box. The only parameter left to choose was min_samples, which we would have liked to set to 1, but it was restricted to a minimum value of 2. This made us unable to create unique clusters for single point outliers and they would all be classified as noise under the same cluster ID. The noise cluster ID will then get used for logs that are not at all similar and therefore loses its meaning. This severely impacts the ability for the sequential model in the next step to properly find anomalies. Therefore, we choose to use DBSCAN over HDBSCAN to perform the clustering step in the pipeline.

The log events can vary for each host because of the different behaviors of people and systems. For each host, we aim to optimize the epsilon parameter as much as possible for the data present in their logs. To achieve this, we would have liked to use the DBCV index by Moulavi et al. (2014). However, this method has not yet been implemented in a scalable Python solution.

Our aim was to perform multiple tests in which we increase the epsilon parameter in steps of 0.25 ranging from 0.1 to 1. Then for each change in the epsilon parameter, we would calculate the DBCV index, which is an objective metric optimized for density-based clustering. The DBCV index measures the density connectedness and separation of clusters. Based on this, we could have evaluated how well-separated our clusters are and get an indication about the quality of our clustering assignments. It assigns a score between -1 and 1. The closer to 1, the better optimized the clustering is for our data. Based on the values it returns, we would have selected the epsilon value that achieves the highest DBCV score. We adapted the source code from R to Python, but calculating the score took too much time. Therefore, we opted to stick to the default epsilon of 0.5.

4.1.4 Enrichment with extra features

In addition to the existing data coming from language, we sought to extract additional information from the logs that could serve as indicators of malicious behavior. We applied Feature Generation to create new features from the existing log text. Feature Generation (also known as feature construction, feature extraction or feature engineering) is the process of deriving new features from the original ones. For instance, we could detect IP addresses in a log and encode their presence as a binary feature (1 for present, 0 for absent). Feature engineering is often a vital step in machine learning tasks, as it enhances the ability of machine learning models to better distinguish between the data that they need to process. (Verdonck et al., 2021; Duboue, 2020)

Through various brainstorming sessions and expert knowledge from analysts at Eye Security and APTA Technologies, we identified two categories of additional features that were worth exploring: static and dynamic features. Static features always return the same result when provided with the same input and produce the same output for a given input, whereas dynamic features are dependent on the previous or current information they have encountered or processed.

Feature Type	
Static	**Dynamic**
Sentiment	New Username
Business hours	New IP
IP presence	New Hostname
Unknown IP	
Keyword presence	

Table 4.3: Listing of the various extra features implemented.

During the development, we explored three different approaches or implementations for feature enrichment. The first approach is to augment the feature string that we use to train the language model with additional features. For example, we could check whether the log event occurred during or outside business hours, and append this information as text to the feature string. The rationale behind this method is that the language model should be able to understand the log by incorporating this extra contextual information. However, this implementation has several limitations. First, this approach reduces the interpretability of the model, as it obscures the way the features of a log sentence are processed. Second, it is not scalable. If additional features need to be incorporated in the anomaly detection task, the entire language model needs to be retrained to accommodate these new features, which can take a significant amount of training time and costs.

Another approach is to enrich an embedding with extra dimensions based on certain features or the occurrence of certain words or phrases similar to Egersdoerfer et al. (2022). After generating the sentence embeddings, additional dimensions can be appended to the embeddings to indicate the presence or absence of features, words or phrases in the text. For instance, for the features described in Table 4.3, an extra dimension can be added to the embedding to reflect its presence or absence. This enrichment of features as dimensions in the embeddings is illustrated in Figure 4.2.

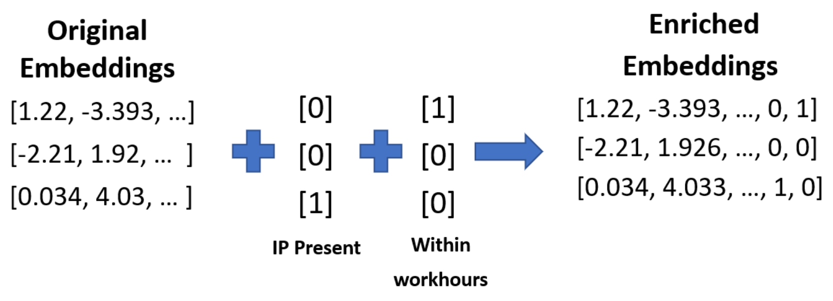


Figure 4.2: Illustration of old approach for feature enrichment of the embeddings.

This method addresses the drawback of the first approach by allowing us to incorporate new features into the embeddings without needing to retrain the language model. This makes the method more scalable and efficient, as we only need to append the new features to the existing embeddings and then perform the clustering step on the enriched embeddings. This would take less time than retraining the whole language model from scratch. Moreover, this method enables us to create more transparent and meaningful clusters, as we can leverage the additional features to capture more information about the log messages and their relationships. We can also examine the features of each cluster and understand why certain log messages are grouped together. However, we encountered a challenge when applying this method to our data. The clustering algorithm that we used relies on Euclidean Distance as a similarity measure, which works well for numeric data but not for binary data. Since our embeddings would contain both numeric and binary values, we could not find a suitable metric that could handle both types of data effectively and accurately. Therefore, we decided to abandon this method and only perform language based clustering on the log messages without any additional information.

Our final approach was to use the features as enrichment attributes for the FlexFringe model in a simplified manner. Rather than incorporating the features into the embedding as separate dimensions, we generate new logs based on the occurrence of the extra feature event at the same timestamp as the original logs. We store these new logs, timestamps and their IDs in a distinct file for each corresponding host. Table 4.4 illustrates the structure of such a host file, which contains the timestamp, feature description, and feature ID. FlexFringe can then load and process both these new enriched feature log IDs and the cluster IDs derived from the language models. By examining the patterns of these features jointly over a certain time interval, they could assist in providing more information for detecting anomalous malicious behavior to security analysts. This approach is scalable as well, since the list of extra features can be easily modified and extended with new features.

RecordNumber	Event	Timestamp	SourceFile	EventID	FeatureID	Feature Description
590957	590957	2023-01-22 01:04:12.561	Security.evtx	4634	1	Log outside of work hours
590958	590958	2023-01-22 01:04:12.565	Security.evtx	4672	1	Log outside of work hours
590958	590958	2023-01-22 01:04:12.565	Security.evtx	4672	5	Keyword admin detected
590959	590959	2023-01-22 01:04:12.565	Security.evtx	4624	1	Log outside of work hours
590959	590959	2023-01-22 01:04:12.565	Security.evtx	4624	2	IP present in log
590959	590959	2023-01-22 01:04:12.565	Security.evtx	4624	5	Keyword admin detected
590960	590960	2023-01-22 01:04:12.565	Security.evtx	4627	1	Log outside of work hours

Table 4.4: Example of extra generated features for security events of a host

4.1.5 FlexFringe

Finally, we use the cluster representations of the events and the additional feature enrichment logs as input to our FSA model called FlexFringe. FlexFringe is a flexible state machine learning algorithm developed by Verwer and Hammerschmidt (2017) and implemented by APTA Technologies. It allows for flexible learning of state machines (deterministic automata) from traces of events, which are the clustered log events in our case. FlexFringe can analyze the frequency and patterns of event transitions within different host domains and identify rare or unusual transitions that may indicate anomalies. FlexFringe encodes each sequence in the input data into a prefix tree. FlexFringe then gradually merges states that have similar future behavior. FlexFringe stops the merge process when no more states can be merged. The final result is a state machine model that depicts the dataset by states and the transitions among them (Nguyen, 2022). When applying it to our case, it computes a novelty score based on events it has not seen before or that events occur in a different context, as can be seen in Figure 4.3. We will feed both the enriched features and clusters to FlexFringe and let the model find anomalous behavior in the data based on these features. Although the goal of this research is not to evaluate the performance of this sequence classifier, we still need to use it to see how well the two pipeline approaches work in practice.

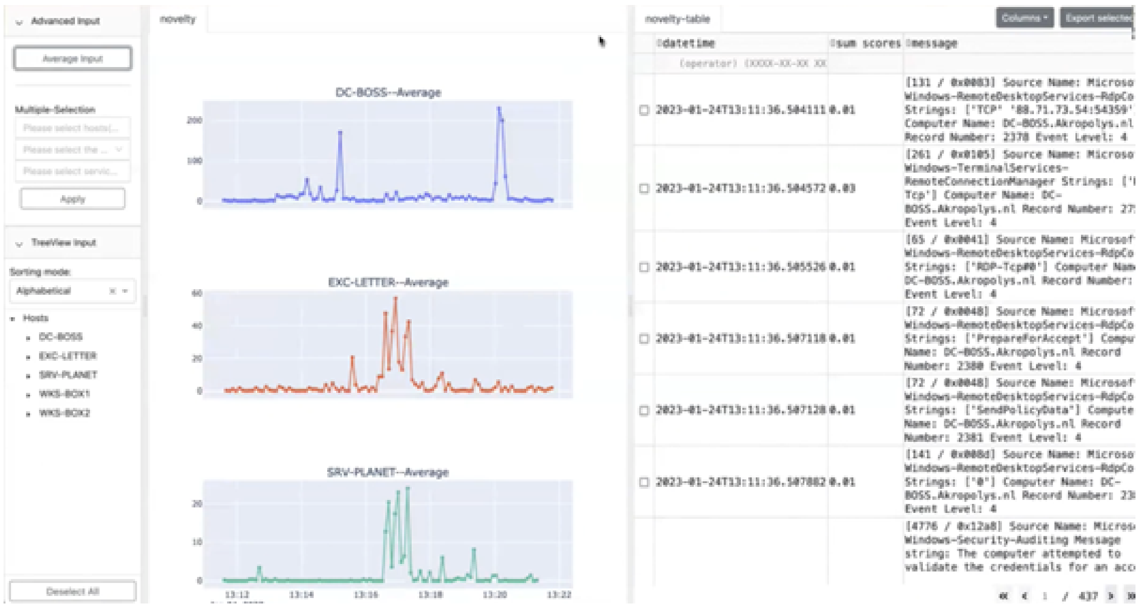
4.2 Training process of the models

4.2.1 Training phase

The models are trained using resources provided by Google Cloud Platform to Eye Security. Using the Hugging Face library we are able to train the models using GPUs which will decrease the time required to train the transformer language models. Before testing and implementing the whole model build pipeline, we had to thoroughly inspect and verify the code and approach to ensure we are not doing anything wrong that could lead to technical debt. Additionally, several optimizations were implemented to make the sure we utilize the GPU available optimally. Since the research was conducted at a company that does not have an unlimited budget to train the models, it would be very expensive if any errors were made during the training phase, which requires a considerable amount of computing power. Therefore, we had to perform several trial experiments to make sure we select the best compute instance configuration, which we will discuss in the next section. All experiments were tracked using mlflow and logged on DagsHub.

Training Costs

To estimate the costs and time required to train the roberta-base language models from scratch, we performed several trial runs on the Google Cloud Platform (GCP). GCP offers various types of GPUs that can be used for training deep learning models, such as NVIDIA Tesla T4, P4, P100 and



```
  datetime                    sum scores  message
  (operator) (XXXX-XX-XX XX
□ 2023-01-24T13:11:36.504111  0.01        [131 / 0x0083] Source Name: Microso
                                          Windows-RemoteDesktopServices-RdpCo
                                          Strings: ['TCP' '88.71.73.54:54359'
                                          Computer Name: DC-BOSS.Akropolys.nl
                                          Record Number: 2378 Event Level: 4
□ 2023-01-24T13:11:36.504572  0.03        [261 / 0x0105] Source Name: Microso
                                          Windows-TerminalServices-
                                          RemoteConnectionManager Strings: ['
                                          Tcp'] Computer Name: DC-
                                          BOSS.Akropolys.nl Record Number: 27
                                          Event Level: 4
□ 2023-01-24T13:11:36.505526  0.01        [65 / 0x0041] Source Name: Microsof
                                          Windows-RemoteDesktopServices-RdpCo
                                          Strings: ['RDP-Tcp#0'] Computer Nam
                                          DC-BOSS.Akropolys.nl Record Number:
                                          Event Level: 4
□ 2023-01-24T13:11:36.507118  0.01        [72 / 0x0048] Source Name: Microsof
                                          Windows-RemoteDesktopServices-RdpCo
                                          Strings: ['PrepareForAccept'] Compu
                                          Name: DC-BOSS.Akropolys.nl Record
                                          Number: 2380 Event Level: 4
□ 2023-01-24T13:11:36.507128  0.01        [72 / 0x0048] Source Name: Microsof
                                          Windows-RemoteDesktopServices-RdpCo
                                          Strings: ['SendPolicyData'] Compute
                                          Name: DC-BOSS.Akropolys.nl Record
                                          Number: 2381 Event Level: 4
□ 2023-01-24T13:11:36.507882  0.01        [141 / 0x008d] Source Name: Microso
                                          Windows-RemoteDesktopServices-RdpCo
                                          Strings: ['0'] Computer Name: DC-
                                          BOSS.Akropolys.nl Record Number: 23
                                          Event Level: 4
                                          [4776 / 0x12a8] Source Name: Micros
                                          Windows-Security-Auditing Message
                                          string: The computer attempted to
                                          validate the credentials for an acc
                                                          « ‹ 1 / 437 › »
```

Figure 4.3: Dashboard of the FlexFringe Anomaly Detection tool.

V100. Each GPU has different specifications and pricing, depending on the region and the usage. To compare the performance and cost of different GPUs, we perform trial runs of an hour and measured the training speed in terms of logs processed per second. Based on the results, we estimate the total time and cost for training the full model on each GPU type. This helped in choosing the most suitable GPU for our research and budget.

GPU Type		**Training Specs**		**Time (1hr +)**					**Costs**		
Name	**VRAM**	**Eval step size**	**batch size**	***Steps***	***it/s***	***n logs***	***log/s***	***perfm. vs. T4***	**per hour $**	**per day $ (24h)**	**per log**
T4 (Colab)	16 GB GDDR6	1000	8	11000	3,06	88000	24,4	0,846	-	-	-
T4 (GCP)	16 GB GDDR6	1000	8	13000	3,61	104000	28,9	1	0,477	11,448	0,0000045865
P100	16 GB HBM2	1000	8	12000	3,33	96000	26,7	0,92	1,454	34,896	0,0000151456
V100	16 GB HBM2	1000	8	35000	9,72	280000	66,7	2,308	2,11	50,64	0,0000087916
A100	40 GB HBM2	1000	32	12000	3,33	384000	106,7	3,692	2,65	63,60	0,0000069010

Table 4.5: Trial run results for using different GPU types.

The results of these trial runs are shown in Table 4.5. Based on this, we can conclude that using the T4 GPU has the best cost/performance when looking at the costs per processed log. However, using this GPU to train our models would take quite some time. The P100 performs worse and costs more per log to process, so we can discard this option. When looking at faster GPUs, only the V100 and A100 remain. Both of them are more costly than the T4, but they process the data 2.3x to 3.7x faster which, given the large training set that we use, is preferable. Between the A100 and V100, the A100 comes out on top in processing speed, but also in costs per log. The V100 can still be used, but only if it is the only one available.

Training from scratch and fine-tuning

Figure 4.4 shows the evaluation loss of both the fine-tuned and scratch versions of the RoBERTa model. The fine-tuned version achieved the lowest evaluation loss of 0.246 after 1253000 steps which amount to 40.1 million logs. The model from scratch achieved the lowest loss of 0.90 after 1288000 steps which amount to 41.2 million logs. It was to be expected that the training loss of the fine-tuned model would be lower, since that model already had pre-trained knowledge that it could use to predict the masked token.

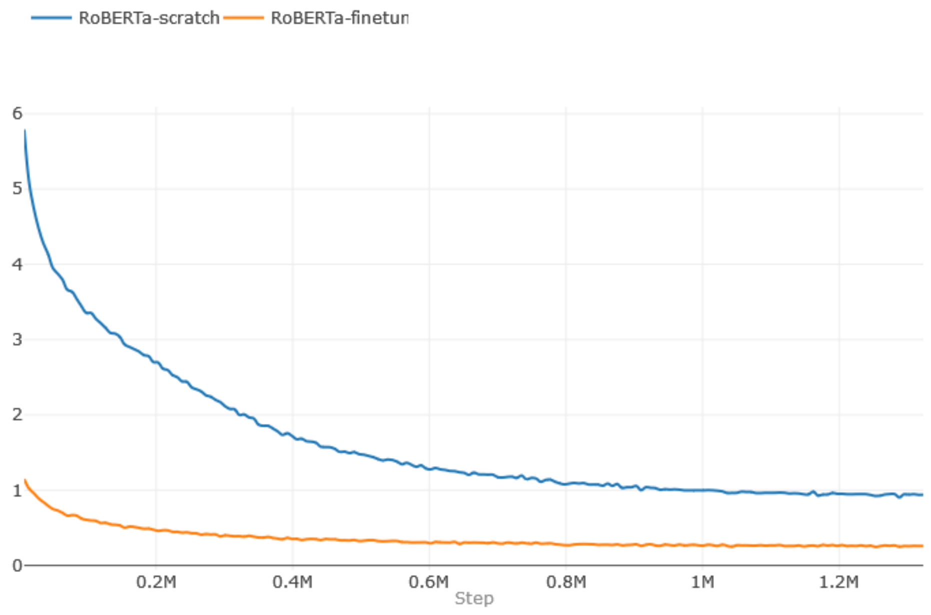


Figure 4.4: Evaluation loss during training of the RoBERTa models (scratch and fine-tuned)

4.3 Finding an evaluation method

After having developed the various language models, we need to come up with a way to evaluate them to see how they compare against each other and to find out whether the best performing model improves on the existing baseline. However, evaluating the models' performance and quality of understanding the log contents is not a trivial task, as there are no clear-cut criteria or metrics that can capture all the aspects of natural language and even more so log language. While working on the thesis, we came up with several plans to evaluate the models before choosing our final evaluation method, since our initial plans could not be used in the end. First, we planned to use F1-score as a comparison metric on labelled data. When that was not possible, we tried to use distant supervision, which also did not work out. At last, we ended up using manual inspection of the cluster contents to see how well they perform in practice.

4.3.1 F1-score on labelled data

At first, we planned to do an extrinsic evaluation where we evaluate and compare the performance of the traditional TF-IDF vectorizer model to the more complex transformer language models with added features to the embeddings. The final models would have been run on a test set containing data from various systems where (simulated) attacks have been performed. After which, we wanted to evaluate the predictions of the final models by calculating the macro F1-score, which treats all possible classes as equally important. Additionally, we would have looked at the precision and recall scores of all labels to find out how the models perform the on benign and anomalous labels.

$$Precision = \frac{TP}{TP + FP} \tag{4.1}$$

$$Recall = \frac{TP}{TP + FN} \tag{4.2}$$

$$F1score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4.3}$$

The F1-score is a commonly used metric for evaluating the performance of both binary and multi-class classification models. It combines the precision and recall of the model into a single score, which ranges from 0 to 1, with higher values indicating better performance since it classifies all input as correct. Precision measures the proportion of true positives (TP) among the total number of positive predictions (TP + FP). It represents the model's ability to correctly identify positive examples. Recall measures the proportion of true positives (TP) among the total number of actual positive examples (TP + FN). It represents the model's ability to identify all positive examples. The F1-score is the harmonic mean between precision and recall scores. It balances the trade-off between precision and recall and provides a single score that summarizes the model's performance. Despite being a good metric to directly compare the performance and effectiveness of the models, it became clear during the research that we could not use this metric. The reason being that we ended up not having proper labelled data to use as a test set for the models. Although there was data available coming from compromised hosts, the events in these logs were not annotated.

4.3.2 Distant supervision

Another option that we considered was using a method called distant supervision to solve the annotation problem presented in the last section. In distant supervision, an already existing dataset is used to collect examples for the relation (label) that we want to extract. We then use these examples to automatically generate labeled (training) data according to some filtering criteria. This is an effective way to create a large amount of training data. The data will most likely contain a lot of noise that does not represent the relation. Hence, this is why distant supervision is known to be a weak supervision method to produce (silver) data as opposed to humanly-annotated (gold) training and test data. While mainly used to create a training set, it can also be applied to create a test set. Since we had logs available coming from systems on which simulated attacks have been performed, our idea was to use the start and end times of the attacks to label the log events. Using this labelled data, we can then observe using the F1-score whether the attacks are picked up by the sequence classifier effectively and which information extraction approach works best. Unfortunately, both the data and start and end times were not available to use ultimately. Additionally, the data would not be representable of real world scenarios since it would have been simulated. Therefore, we chose to not use this evaluation method either.

4.3.3 Manual inspection

Instead of relying on automated metrics, which were not suitable for our task due to the lack of labelled data and the introduction of noise with distant supervision, we chose to perform manual inspection for both the process of choosing the best language model and the anomaly detection inspection with FlexFringe. By manually examining the cluster contents and looking for anomalous behavior in hosts, we can evaluate our approaches from a qualitative perspective and not be constrained by quantitative scores. Manual inspection also has some limitations, such as the subjectivity of the evaluator and the scalability issues of manually inspecting all the cluster contents of all hosts. However, even with these limitations, we think this is the best way to judge the results of our research.

Our idea is to perform several investigations into the resulting clusters of hosts. The first investigation focuses on how spread out the combination of the source EVTX file + event IDs are. Secondly, we also create visualizations of low-dimensional representations of the embeddings and conduct an investigation into several clusters to inspect their contents and see whether they are coherent and meaningful. Based on these observations, we can compare each language model and see which approach creates the most accurate, consistent and meaningful clusters.

After performing the cluster contents investigation, we select our best performing language model and compare it with the baseline TF-IDF implementation in the FlexFringe anomaly detection tool. Here, we analyze several hosts where malicious activities occurred. This allows us to assess how well our new, more complex approach performs. Additionally, we check the inclusion of the enriched features scores for capturing additional context clues about the log message. Based on these comparisons and inspections, we can determine whether the proposed solution outperforms the baseline TF-IDF method and investigate whether they have any limitations on specific logs.

5 Experiments

In this chapter, we list the experiments and tests that we conduct to answer our research question: How can embeddings and language models capture useful representations of log files for analysis in Cybersecurity? Our main hypothesis, as stated in the Introduction chapter, is that the performance of anomaly detection improves using transformer-based language models as a feature extraction method over the baseline TF-IDF weighted word representation model.

To test our hypothesis, we conduct two separate experiments that perform manual inspection of the log clusters and anomalies detected by using different language models. The first experiment aims to select the best and most consistent performing language model out of the four models. The second experiment compares the best model from our new approach against the baseline TF-IDF approach using the FlexFringe anomaly detection tool on several hosts where malicious activities took place.

5.1 Description of the datasets

For our evaluation, we have used the following datasets: 'Azure AD Labs' and 'Case April 2022'. We perform analysis on several hosts within these datasets to see how the clustering performs based on the generated embeddings and to see if we can detect any malicious activities using the current and new approach.

Azure AD Labs:

This dataset is a synthetic dataset with simulated activities separated into five machines: DC-APHRODITE, SRV-TITAN, WKS-PETER, WKS-FROUKJE, and SRV-CALYPSO illustrated in Figure 5.1. The dataset contains the log files from four of these machines, except for SRV-CALYPSO. The dataset represents four different lab tests where an attacker performed various attacks on the machines:

1: The attacker conducted a remote desktop protocol (RDP) brute force attack on the SRV-TITAN machine, but after a successful login, the attacker took no further action.

2: The attacker repeated the same RDP brute force attack on the SRV-TITAN machine, but this time performed several malicious actions on it.

3: The attacker exploited a well-known vulnerability in the exchange server on the SRV-CALYPSO machine. After several malicious actions, the attacker stopped their events, but the log generation continued. This file is not available in the dataset.

4: The attacker repeated the same exchange server vulnerability attack on the SRV-CALYPSO machine, but this time used their access to jump to the domain controller (DC-APHRODITE) and started performing malicious actions on that host.

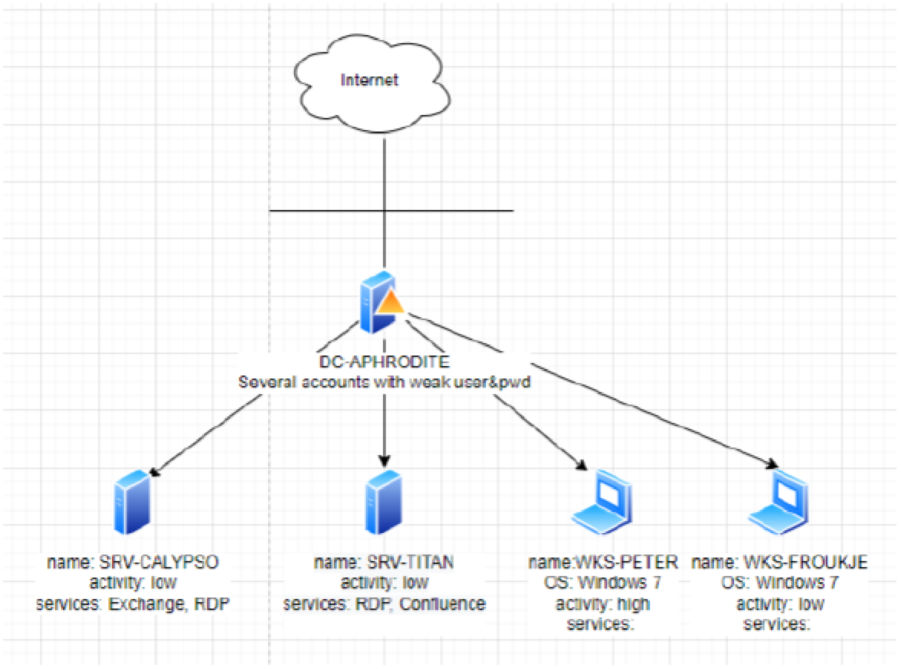


Figure 5.1: Overview of the Azure AD Labs environment

Case April 2022:
This dataset is about a real-world cyber incident that occurred in April 2022, where an attacker exploited a well-known vulnerability in Log4j. Log4j is an open-source logging framework for Java applications that allowed remote code execution (RCE) when an attacker sends a specially crafted input to a vulnerable system. The attacker used this vulnerability to compromise an environment of 73 machines, of which we have 42 machines with log information available. The dataset includes the log files from these machines, as well as the machine that was the root cause of the incident where hmsvc.exe malware was written to the disk

5.2 Experiment 1: Comparing the language models

The purpose of this experiment is to select the best and most consistent performing language model out of four models: all-MiniLM-L6-v2 and RoBERTa (pre-trained/fine-tuned/scratch). We use these models to generate embeddings for the log messages and cluster them using DBSCAN. We evaluate the quality of the clusters based on three criterions or indicators: the spread of combinations of source evtx files and event IDs, low-dimensional visualizations based on the generated clusters by DBSCAN, and the semantic coherence of the cluster contents.

1. The first criterion measures the spread of the combination of the source EVTX file + event IDs. A score closer to 1 indicates that generated event combinations could be similar in terms of their context and almost solely separated on their event ID. If not, this would indicate that the generated events differ from each other while having the same source file and event ID, implying that there are other similarities present besides being grouped based on event IDs. We calculate the ratio by dividing the number of unique combinations that are in multiple clusters (cluster spread) by the number of unique combinations.

2. In the second criterion, we want to visualize the high-dimensional data using t-SNE (Van der Maaten and Hinton, 2008). T-SNE is a technique for nonlinear dimensionality reduction that can create low-dimensional embeddings of high-dimensional data, which can then be visualized. We want to inspect these visualizations and look for patterns, clusters, outliers, or manifolds, that reflect the structure of the data. We provide our DBSCAN cluster labels to see where they end up in the visualization, giving us a good impression of how well the quality of the generated embeddings are and how easy it is to cluster similar embeddings.

3. For the third criterion of the experiment, we want to confirm that the clusters that we observe in the visualizations are in fact meaningful. We inspect the contents of several clusters per language model. We grab two malicious and two benign events. Next, we investigate the other logs that are present in the clusters to which the events have been assigned. We expect that a good clustering method should group together logs that have similar or related content and semantics. Therefore, we want to observe how well the clustering for each model captures the meaning and logic of the log messages. Here, we examine and provide examples of cluster contents and elaborate further on why certain clusters are meaningful or not.

Based on these results, we are able to choose the most consistent and accurate language model to include in the second part of the experiments, where we use it to detect anomalies through FlexFringe.

To summarize, we want to select the most optimal language model to extract information from the evtx dataset. To find it we use the all-MiniLM-L6-v2 model and the three versions of RoBERTa to retrieve embeddings, perform clustering to place similar looking logs together and manually evaluate which method performs best in this step by looking at the event ID spread, visualizations and inspecting our derived impressions by investigating the cluster contents.

5.3 Experiment 2: Manual inspection using FlexFringe

5.3.1 Experiment setup

The purpose of this experiment is to compare the best language model selected from Experiment 1 against the baseline TF-IDF approach using the FlexFringe anomaly detection tool on several hosts where malicious activities took place. We use these models to extract features from the log messages and feed them into the FlexFringe tool and APTA Dashboard to detect anomalies in the logs. We evaluate the performance of the anomaly detection methods based on two criteria: where they are able to identify malicious anomalous activities, and how they deal with false positive in hosts with benign activities.

1. The first criterion measures how well the anomaly detection methods can detect logs that contain malicious activities, such as malware infection, ransomware attack, credential theft, etc. described in Section 5.1. We expect that a good anomaly detection method should flag logs that indicate such activities as anomalies, while finding nothing out of the ordinary from normal logs. We manually inspect the anomalies detected by each method and compare them against the information provided by security analysts to see what should have been picked up.

2. In the second criterion, we want to get an indication of how each method deals with events from a benign host. Ideally, we want to minimize the number of false positives, as these will classify logs as potentially malicious while they are not. False positives can cause alert fatigue and reduce trust in the system.

To summarize, the aim of this experiment is to compare the performance of the two different language models for information extraction and to examine their impact on anomaly detection. We use the FlexFringe tool to use the clusters and information obtained from the extra features. We then investigate the anomalies detected by the tool and attempt to identify malicious activities by looking at the peaks in the graphs. We conduct this analysis on hosts from both datasets and explore several labelled types of malicious behavior to evaluate how well the tool detects them using the two information extraction methods.

This evaluation method enables us to assess the suitability of the language models and the additional features scores for capturing the context and content of a log message. Based on these comparisons and inspections, we can determine whether our proposed solution outperforms the baseline TF-IDF method.

5.4 Overview experimental setup

The above discussed intentions and evaluation methods can be summarized in the overview shown in Table 5.1. It comes down to finding the best performing language model in the first experiment.

This model then gets compared against the baseline TF-IDF method in the second experiment to see which method performs better in practice. In the end, we want to have enough evidence and observations to confirm our hypothesis. The results of these experiments will be discussed in the next Chapter 6.

#	Experiment	Models	Evaluation method
1	Comparing the language models	all-MiniLM-L6-v2 RoBERTa-base RoBERTa-base (scratch) RoBERTa-base (fine-tuned)	1. Measure spread 2. Visualization using t-SNE 3. Investigate cluster contents
2	Manual inspection using FlexFringe	Best model of Experiment #1 TF-IDF (Baseline)	1. Manual inspection of hosts containing malcious activities 2. Inspect benign host

Table 5.1: An overview of the two experiments where we compare the language models and perform manual inspection on hosts with anomalous behavior using FlexFringe.

6 Results

In this chapter, we will analyze the interesting results of the experiments presented in Chapter 5 using the methodology explained in Chapter 4. The results will be further discussed in Chapter 7, where we will elaborate on explanations for the most interesting results.

6.1 Comparison between transformer models

6.1.1 Measuring spread of event IDs

For the first criterion, we measure the cluster spread of the logs' evtx source files and associated event ID combinations. The closer the spread number is to 1, we could assume that the content of source file and eventID logs are similar in terms of meaning and grouped together by the clustering algorithm. The higher the spread ratio is, the clustering algorithm finds it harder to group similar logs events together.

Dataset	Model	Hosts #N	Unique Clusters (all hosts)	Cluster spread (all hosts)	Unique combinations SourceFile + ID (all hosts)	Cluster spread ratio
Azure-AD Labs (Test 1)	*all-minilm-l6-v2*	4	**1735**	**3386**	**2455**	**1.379**
	R-finetuned	4	7859	8688	2455	3.539
	R-pretrained	4	4410	5456	2455	2.222
	R-scratch	4	43187	43396	2455	17.677
Case April 2022	*all-minilm-l6-v2*	42	**30444**	**56959**	**37239**	**1.53**
	R-finetuned	42	141611	162106	37239	4.353
	R-pretrained	42	83283	104297	37239	2.801
	R-scratch	42	883186	898297	37239	24.122

Table 6.1: Overview of the spread ratio of the EVTX SourceFile + EventId combinations for every model based on all hosts in the Azure-AD Labs (Test 1) and Case April 2022 datasets.

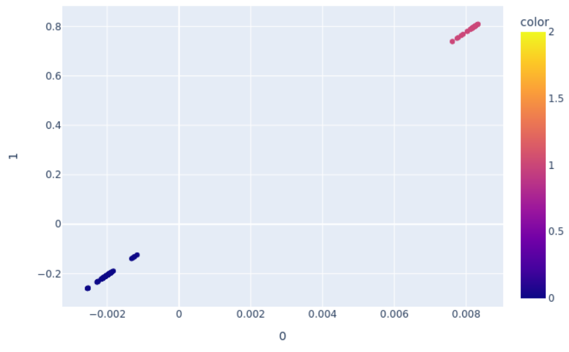
What we observe from our datasets is that the *all-minilm-l6* model achieves the best ratio spread scores *1.379* and *1.53* which are closest to 1 followed by *RoBERTa-pretrained* with scores of *2.222* and *2.801*. The *RoBERTa-scratch* model gets the highest score away from 1, namely, *17.677* and *24.122* for both datasets. We also observe that the *all-minilm-l6 model* is the only model where the number of unique clusters is lower than the number of unique SourceFile + EventId combinations. Based on these results, we can conclude that *all-minilm-l6* performs the best at putting each SourceFile and EventId combination into single clusters without them spreading to other clusters too often, while also grouping similar looking logs together into the same cluster (*1735* vs. *2455* and *30444* vs. *37239*). This indicates that the clusters could be well optimized for later classification by the FlexFringe model. The *RoBERTa-scratch* model has the most trouble putting unique combinations in a single cluster, generating significantly more unique clusters than the total number of unique combinations (*43187* vs. *2455* and *883186* vs. *37239*).

6.1.2 Visualizing the embeddings by their clusters

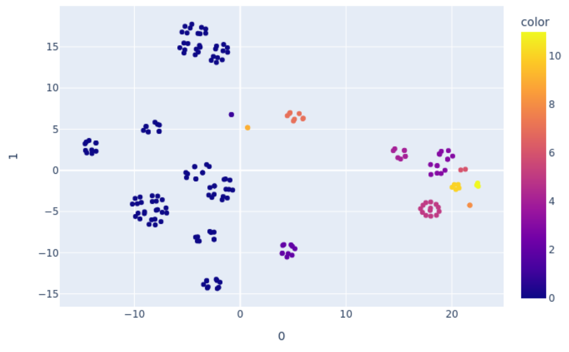
For this criterion, we investigated the generated t-SNE plots of every model for several hosts. We will discuss a couple of the t-SNE plots of the Windows PowerShell evtx file of an infected host in both dataset for all four models. We investigate how well the clusters are formed and grouped together, as observed from these t-SNE plots. We base our final conclusions on the overall patterns that we noticed when inspecting several of these t-SNE plots from other evtx files.

Azure AD Labs (Test 4)

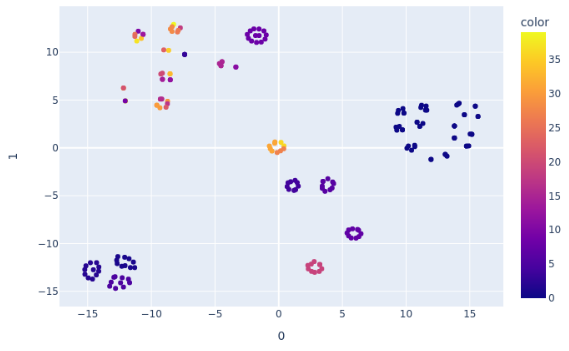
In Figure 6.1, we can observe the generated t-SNE plots for the four models. The first Subfigure 6.1a shows the *all-minilm-l6-v2* model, only got three clusters assigned to its logs and it tries to separate those from each other, this could be because of a suboptimal epsilon parameter in the DBSCAN clustering algorithm that allowed too many logs to be grouped together under the same cluster ID. The RoBERTa models in Subfigures 6.1b, 6.1c and 6.1d show a clearer cluster forming based on the variations in embeddings. We notice that the *RoBERTa-pretrained* model gets the most accurate forming of clusters compared to both *RoBERTa-scratch* and *RoBERTa-finetune* that both show several data points that are close in the space get put together in different clusters. Overall, we notice that the clustering approach does a good job at grouping similar embeddings together.



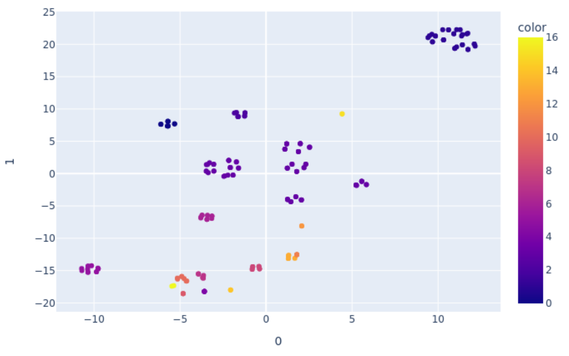
(a) all-MiniLM-L6-v2



(b) RoBERTa-base (pretrained)



(c) RoBERTa-base (scratch)



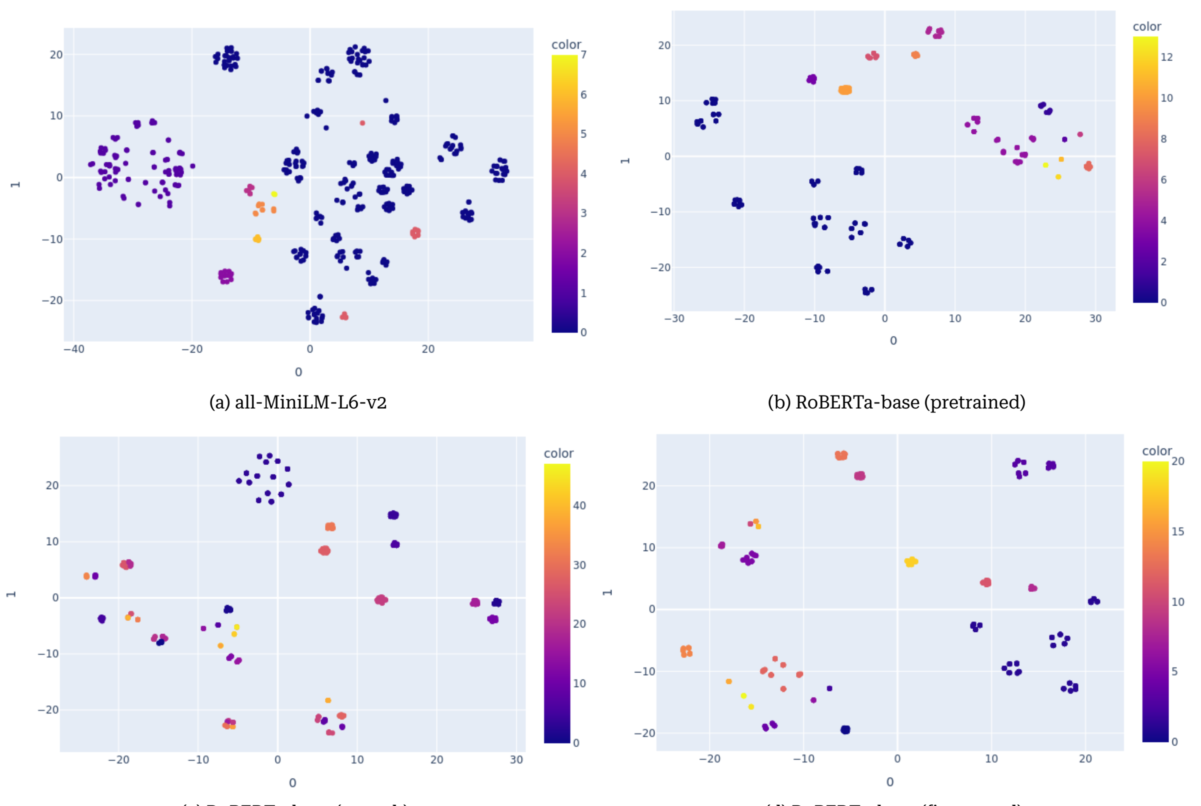
(d) RoBERTa-base (fine-tuned)

Figure 6.1: t-SNE plots of Azure AD Labs (test 4) Windows PowerShell evtx file on host DC-APHRODITE for the four language models. The color indicates the assigned cluster ID.

Case April 2022

Next, we will investigate the t-SNE plots of an infected host in the Case April 2022 dataset. The plots for the Windows PowerShell evtx logs are shown in Figure 6.2 for all four models again. As shown by the results in Figure 6.2a, we see that the *all-minilm-l6* this time also does a decent job at grouping logs events together. Again we observe that the clustering performed on *RoBERTa-pretrained* embeddings assigns the most accurate clusters with no cluster IDs being near other clusters. The *RoBERTa-scratch* model assigns a lot of unique cluster IDs near groups of other clusters, indicating that the embeddings are not of the desired quality. While *RoBERTa-finetune* does a lot better job than *RoBERTa-scratch*, it sometimes still assigns unique cluster IDs to points that close together.

Based on the resulting plots in both datasets, we assume that the *RoBERTa-pretrained* model is the best out of the other models at producing meaningful embeddings, since it performs best at grouping assigned clusters together. In the next test, we will investigate our assumption further by inspecting the cluster contents.



(a) all-MiniLM-L6-v2

(b) RoBERTa-base (pretrained)

(c) RoBERTa-base (scratch)

(d) RoBERTa-base (fine-tuned)

Figure 6.2: t-SNE plots of the Windows PowerShell evtx file from the Case April 2022 dataset on an infected host for the four language models. The color indicates the assigned cluster ID.

6.1.3 Inspecting the cluster contents

In this criterion, we perform an inspection of the contents of clusters created by our approaches. We selected two malicious events and two benign events, one of each type from both datasets, and we want to inspect the clusters amongst what other logs they end up for each of the four language models. Table 6.2 shows the selected events that we plan to investigate.

Cluster contents with malicious logs

We are interested in exploring amongst what other logs clusters containing malicious events, as presented in Table 6.2, are combined. Specifically, we want to determine whether the context of these malicious events tend to be grouped together with other similar potentially malicious events or if they form distinct clusters. The first type of malicious event involves authentication requests made by attackers in their efforts to gain unauthorized access to the system. The second event revolves around the 'whoami' command obfuscated in base64 encoding, often used by attackers to identify the system they are targeting.

For the first event, we observe that the *all-minilm-l6-v2* model successfully clusters all events associated with ID 4776 into a single cluster, aligning with our expectations as these events share

Type	Host	Source	ID	Activity
M	DC-APHRODITE (Azure AD Labs Test 1)	Security.evtx	4776	Malicious authentication requests by user John
M	Infected Host (Case April 2022)	PowerShell.evtx	600	'whoami' command executed in PowerShell, output sent to external IP
B	DC-APHRODITE (Azure AD Labs Test 1)	Application.evtx	1033	A program was installed on the host
B	Benign Host (Case April 2022)	Microsoft-Windows-GroupPolicy%4 Operational.evtx	4017	Making LDAP calls to connect and bind to Active Directory

Table 6.2: Selected events where M = Malicious and B = Benign.

identical meaning. However, a notable issue arises as this model also includes numerous unrelated events within the same cluster ID. In contrast, the RoBERTa-pretrained model does not exhibit this particular problem. As shown in Appendix Table A.2.1, we can observe that it generates multiple clusters for the same event ID. Yet, it assigns these new clusters only when the events are show the targeting of distinct hosts. While at the same time grouping similar events together that target the same host, which is what we like to see. This could be useful in tracking the occurrence of new events by the FlexFringe model. Conversely, the other two models, *RoBERTa-scratch* and *RoBERTa-finetune*, return different results. They assign each unique occurrence of event ID 4776 to a new individual cluster, failing to group this particular set of similar events together.

When we turn our attention to the clusters containing the second event, we find that the *all-minilm-l6-v2* model assigns only five clusters for the entire evtx file. Upon closer inspection of the cluster where the malicious event is assigned to, we observe that this cluster consists of several other logs with differing meanings. It would be more suitable to segregate these logs into separate clusters rather than putting them in the same cluster. Showing better results are the RoBERTa models that all return a consistent set of logs within the cluster containing the malicious event. These logs include instances of the same events where base64-encoded commands are executed within the PowerShell terminal, showing similar behavior to our selected event, which entails the 'whoami' command. Notably, they also show similar behavior from a few months prior, which might be of interest to a security analyst. Additionally, this cluster contains logs with different event IDs (400, 403, and 800) that feature similar obfuscated base64 code.

6.1.4 Cluster contents with benign logs

We also want to investigate some benign logs to see among which other logs they get grouped together and see if this makes sense based on their similar meaning. Appendix Table A.3, shows an example of one of the model's results when inspecting a benign event with the *RoBERTa-pretrained* model. The first event we investigate is the installation of a program on a host, and for the second event we inspect the log "Making LDAP calls to connect and bind to Active Directory" with is a log within the

When inspecting the first event with the *all-minilm-l6-v2* models, we notice some interesting behavior. It seems that when we observe the other logs inside the clusters in which the event ID gets put in that the model does a good job grouping the logs together based on the program context in which they are used. We see that Windows programs get put together inside a cluster and that VS, Python or Slack programs are put in different clusters. However, we also notice that other installer related events like "Installation completed successfully", "Program uninstalled successfully" and "Installer Exited" are grouped in the same cluster. For the *RoBERTa-pretrained* model, we observe similar behavior, it groups logs based on program context, but this time the other logs included in the clusters are only the "A program was deleted" events within the same program context. The *RoBERTa-scratch* model only returned unique isolated clusters with no other logs present in them back. These logs logically only contained the selected event. At last, the *RoBERTa-finetune* model performs better than the *RoBERTa-scratch* since it not solely returns logs as unique isolated clusters. It groups them together based on the program context, but does it worse than *all-minilm-l6-v2* and *RoBERTa-pretrained* by being too specific, sometimes separating logs that could be grouped together.

Regarding the second event's clusters, the *all-minilm-l6-v2* model only puts all occurrences of the event into a single cluster with no other events present. It does not differentiate between the EventData context present in the event. The *RoBERTa-pretrained* and *RoBERTa-finetune* models generated three clusters. The clusters don't consist of any other logs and we observe that the clusters are formed based on contextual values in the EventData yet again. For *RoBERTa-scratch*, yet again we notice that it forms isolated unique clusters, but noticed that it grouped two logs together that had only 1 character different, so it looks like that was sufficient for the clustering algorithm to group them together based on their similar embedding distance. However, if there are more character different as in the other events that have similar context, it will isolate these events into unique clusters because the distance again is too large.

6.1.5 Choosing the best language model

After evaluating the results from our experiments, we chose to continue with the RoBERTa-pretrained model. From our observations, we saw that we got the most consistent results based on all three criterions. The *all-minilm-l6-v2* model is very promising, but we assume due to a not very well optimized epsilon parameter, we got several cluster results back that did not look good because they were generalizing too often.

A pattern that we observe with RoBERTa-scratch is that it creates the highest number of unique clusters per event ID in the evtx files out of the four models. After investigating why this happens in the malicious and benign logs, we see that it tends to create separate clusters for almost similar messages. As shown by the spread in Table 6.1, t-SNE Subfigures 6.1c, 6.2c and from our manual inspection of the cluster contents, for this model the DBSCAN sometimes assigns an event to be the only event in an isolated cluster while it could be put in a different nearby cluster together with similar logs. While FlexFringe might still pick up on most of these new event sequences within a certain time period, this is less than ideal since it also increases the granularity of the sequences when no malicious activities are taking place on a host. This makes it hard to track what is normal behavior and what is anomalous since also most of the normal behavior will get assigned to be an isolated cluster. The RoBERTa-finetune model performed better than the previously mentioned models. However, sometimes it is a bit too specific as well when are slight differences in the context of events that we observed, generating more new unique clusters while grouping them would have been better. We observed that the RoBERTa-pretrained model retrieved better and more consistent results by generating the most accurate clusters while also separating them based on contextual information without being too specific and generating even more clusters. Therefore, in the second phase of the experiments, we will compare RoBERTa-pretrained + DBSCAN approach with the TF-IDF + Mini-batch K-means approach.

6.2 Transformer LM vs. TF-IDF baseline using FlexFringe

In this study, we conduct a comparative analysis of two distinct methodologies within the context of anomaly detection, using FlexFringe as the main classifier. We subsequently examine and interpret the outcomes by looking at the results presented in the APTA Dashboard. Our investigation begins with an exploration of the Case April 2022 dataset, focusing on the detection of anomalies within a real world infected host. Following this, we delve into the Azure AD Labs (Test 1) dataset, where we also investigate an infected host. Lastly, we want to find out how the approaches perform when dealing with a benign host.

6.2.1 Investigating Case April 2022 dataset

We're examining the labeled malicious activities within the PowerShell logs of an infected host from the Case April 2022 dataset. Security analysts identified the activities listed in Table 6.3 during the incident response case. To safeguard security and privacy, we've redacted specific contents. These activities occurred roughly around January 17th and March 29th of 2022.

#	**Malicious activities**
1	<filename1>.exe, <filename2>.dll and <filename3>.dat are downloaded via PowerShell from <IP> and written to C:\windows\debug
2	\item 'whoami' command is executed via PowerShell and its output is sent to <IP>
3	\item Start of repeating malicious PowerShell persistence: IEX ((new-object net.webclient).downloadstring('hxxp://<IP>/vmware/horizon/<filename>.php?p=<hostname>'))
4	\item First malicious PowerShell activity: IEX ((new-object net.webclient).downloadstring ('hxxp://<IP>/<filename>.jpg')))

Table 6.3: Labelled malicious events in the PowerShell logs on infected host from Case April 2022 dataset.

In Figure 6.3, we present the graphical output generated by the FlexFringe model for both the current approach and our new approach. These peaks in the graph indicate potential anomalous events based on previous patterns and behavior, and serve as valuable cues for security analysts conducting incident response investigations.

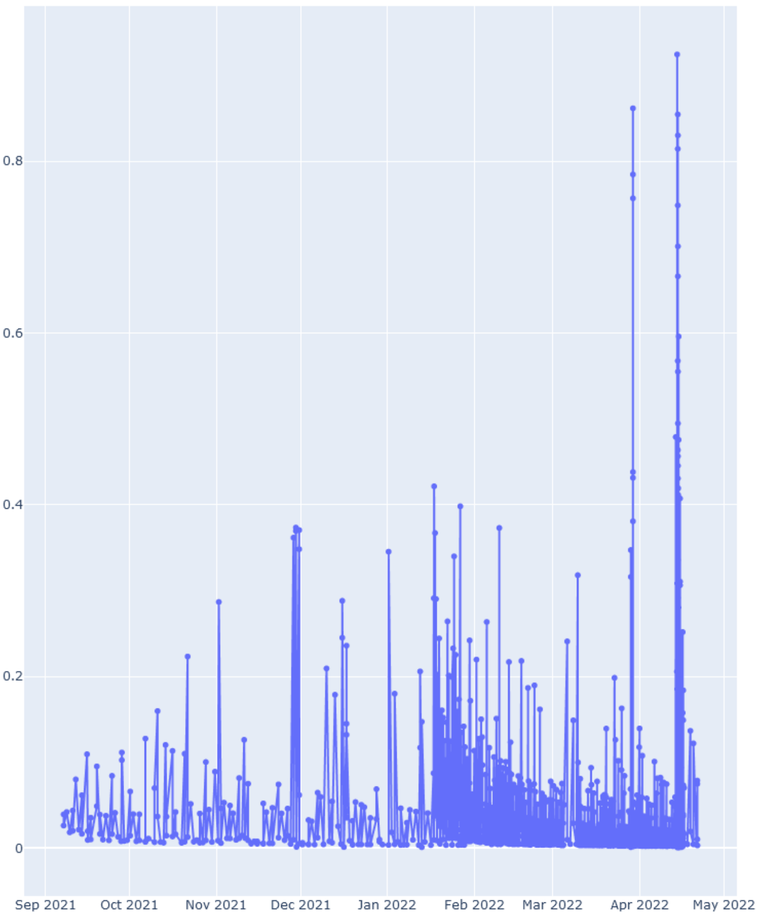
Initially, we applied the existing TF-IDF approach to identify these activities. As seen in Subfigure 6.3a, we quickly located events concerning the downloads and the installation of malicious files, as described in the first event of Table 6.3, occurring on March 29th. Additionally, we discovered the presence of the 'whoami' command encoded in base64; although it did not show as a distinct peak in the graph, its association with nearby events allowed us to identify it. Furthermore, we traced the inception of recurring malicious PowerShell persistence back to January 17th.

When analyzing the graph generated by our new approach in Subfigure 6.3b, we successfully detected most of the aforementioned malicious events. These events are visually represented as peaks on January 17th and March 29th, 2022. Notably, the 'whoami' command did not exhibit a distinct peak either, but could be derived by inspecting events near the peaks as well.

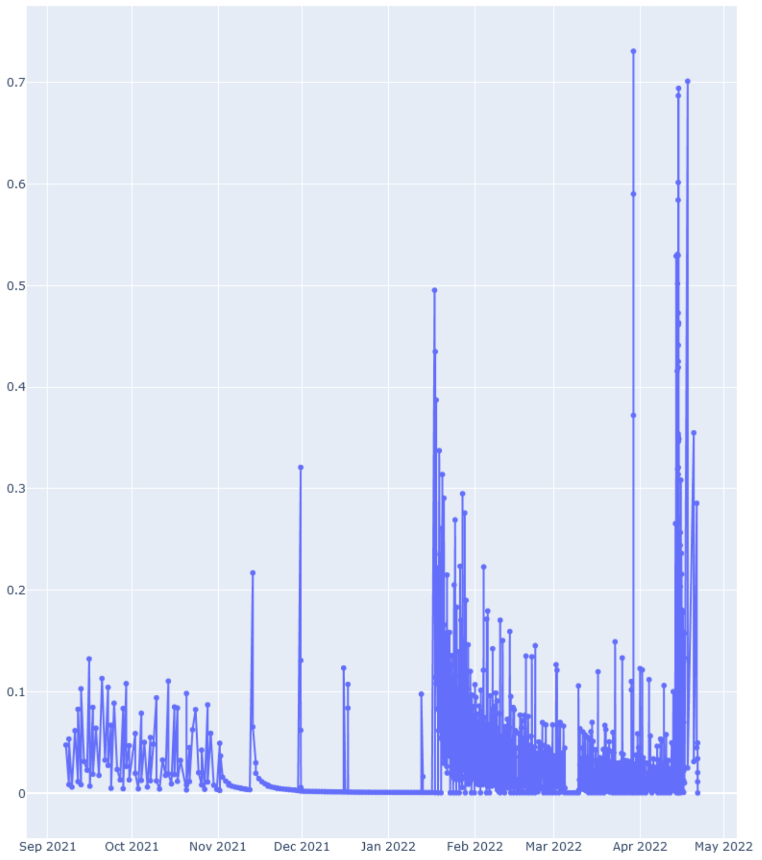
By examining the supplementary features within the evtx file, as displayed in Appendix Figure A.4.1, it becomes clear that these features contain a considerable amount of noise. Notably, the most valuable peaks in this context are primarily generated when new IP addresses and usernames appear in the logs. These IP addresses sometimes correspond to actual IPs, but at other times, they seem to come from software versioning that coincidentally follows a similar format to an IP address. In contrast, the static extra features, apart from the IPs present in the logs, do not significantly contribute to the analyst's understanding, as they are not notably present during the peaks coinciding with interesting events. However, some dynamic extra features, such as newly encountered usernames and IPs, could indeed offer valuable insights for investigation. Furthermore, it is noteworthy that a substantial number of these events occur at the same time as the timeframes of the malicious activities.

6.2.2 Investigating Azure AD Labs dataset

We investigated the Azure AD Lab (Test 1) dataset using both approaches to see what kind of behavior is happening on the system at important times as classified by security analysts. They deemed



(a) TF-IDF + MB-Kmeans



(b) RoBERTa-pretrained + DBSCAN

Figure 6.3: Comparison between the TF-IDF + Mini-Batch K-means vs. RoBERTa-pretrained + DBSCAN approaches for an infected host of the Case April 2022 dataset where anomalous behavior is detected and investigated.

the time around 13:11 January 24th, 2023, as an important time when malicious activities take place.

First, we examined the current TF-IDF approach and discovered several malicious activities by looking at the graphs in Appendix Figure A.1. We noticed the running of a remote PowerShell session with wsmprovhost.exe in the peak activities of the PowerShell logs. Next, we observed an event where a password extraction script was executed, though less clearly marked by a peak. In the security logs, we found an event that specified that the credential manager's credentials were read, followed by an administrative logon. At last, we saw multiple authentication requests and the login of a user with username 'John' from workstation remnux.

Next, we also investigated the logs using our new approach. We saw most peaks occurring at the specified time in the overall FlexFringe output, as shown in Appendix Figure A.2. We discovered various malicious activities represented in these peaks when we zoomed in on the date and time. In the security logs, we found many authentication requests from a user "John" on the DC-APHRODITE system, as well as an attempt to change a privileged object by this John. Furthermore, in the PowerShell logs we saw the execution of a username + password extraction script, the download of a privilege escalation script from GitHub obfuscated by base64 code and the running of a remote PowerShell session via wsmprovhost.exe. Additionally, we also noticed several secondary malicious logs that were produced by primary malicious actions of the attackers in the system logs.

Figure 6.4 displays the addition of the extra contextual features. We could not easily determine on which specific logs they provide context due to this not being implemented yet, but similarly to the Case April 2022 dataset, they gave a good indication of when new anomalous events are happening that might be worth investigating. In the Figure, we clearly saw that between the 19th and

24th of January, there was a lot of activity based on the extra features logs. We saw that FlexFringe produced high peaks based on new username occurrences and admin keyword presence when we zoomed in on the specific activities during the important times.

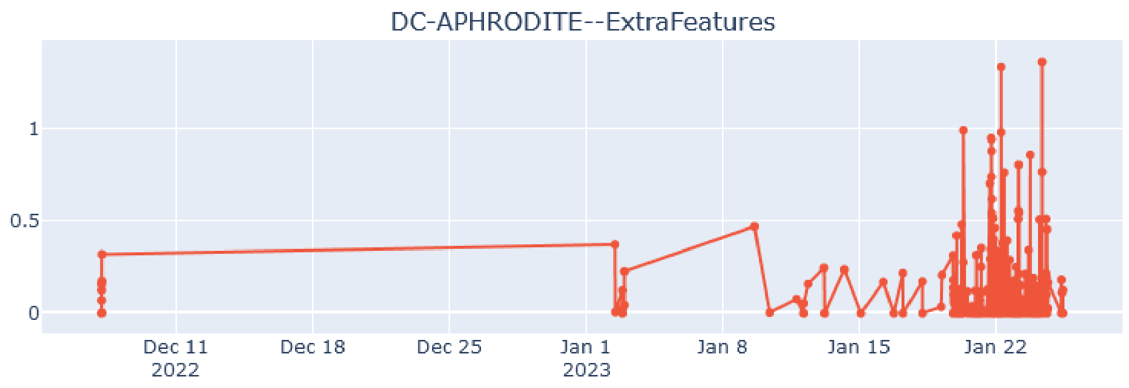


Figure 6.4: Extra contextual features of DC-APHRODITE in the Azure AD Labs (Test1) dataset.

6.2.3 Investigating a benign host

To see how sensitive each approach is to finding anomalous behavior, we performed some additional tests on hosts where no malicious behavior is found. Ideally, the logs should not have any peaks (false positives) that could trigger an investigation by a security analyst, but this can be hard to achieve since even normal behavior can be regarded as anomalous sometimes. However, as can be derived from Appendix Figure A.4 and A.5, we see that both approaches show peaks even when no malicious actions are taking place on the computer. The activities present in these peaks include anomalous behavior such as the enabling of the Minesweeper game on Windows, quick logoffs and logons on the WKS-FROUKJE machine as well as an administrative logon, new service installations and some warnings and errors.

Comparing the benign host graphs to the infected hosts graphs like the one in Figure 6.3, we see a difference in that when malicious anomalous behavior is present, there are more activities happening on the graph and the peaks stretch out over a longer time period. Therefore, this could potentially help a security analyst to identify when and where most of the anomalous actions are happening, possibly caused by various malicious actions of an adversary that has infiltrated the host.

7 Discussion

In this chapter, we present a comprehensive summary of the key findings derived from the results presented in Chapter 6. We delve into the context and rationale behind these findings. Additionally, we highlight the limitations encountered during the course of this research and provide recommendations for future studies.

7.1 Summarization of Key Findings

Experiment 1

1. **Cluster meaningfulness:** Our analysis showed that the RoBERTa-pretrained model produced clusters with the highest degree of meaningfulness. Conversely, the RoBERTa-scratch model generated the least meaningful clusters, often resulting in numerous isolated and unique clusters.

2. **Training language models with logs:** When using logs to train language models, we did not observe a significant improvement in the quality of embeddings and resulting cluster generation. It only made each cluster more specific instead of grouping similar looking logs together.

3. **Role of epsilon parameter:** The epsilon parameter of the DBSCAN clustering algorithm proved to be essential in optimizing the production of clusters based on the distance between embeddings. Careful selection of this parameter per log type is crucial for achieving meaningful results.

Experiment 2

4. **Similar performance by both approaches:** Both the current TF-IDF + Mini-batch K-means approach and new RoBERTa-pretrained + DBSCAN method demonstrated similar performance in identifying anomalous events and behaviors using FlexFringe, including malicious activities. Though, the resulting peaks in our new approach are more obvious.

5. **Contextual features:** The inclusion of extra contextual features provided valuable insights into the detection of attack time periods. However, these features did not consistently offer precise details regarding the nature of the events.

7.2 Interpretation and Summarization of the Results

In our first experiment, we conducted a detailed investigation of the clustering results, leveraging spread, T-SNE plots, and cluster contents to gain insights into the performance of the four language models. The *all-minilm-l6-v2* model demonstrated a tendency to group logs together that ideally should be separated. We suspect that this behavior may be caused by a suboptimal tuning of the epsilon parameter in the DBSCAN algorithm, that merges logs that are in close to each other in the embedding space. This also goes for the other models where the epsilon parameter has not been optimized either, but it seems that the default value is better suited for the RoBERTa models. In contrast to the *all-minilm-l6-v2* models, all RoBERTa models exhibited a higher spread value and produced more coherent cluster contents. The *RoBERTa-scratch* model, which lacked pretrained language knowledge, generated an excessive number of unique, isolated clusters. This over-fragmentation is likely due to the model's lack of language understanding, which could not be effectively compensated for within the limited data and available training time. The *RoBERTa-finetune* model also displayed a degree of specificity, while better than the *RoBERTa-scratch* that led to the creation of clusters that could have been merged with others. The model's training on new feature strings may have shifted its focus away from human language, creating a larger distance in the embeddings based on that, leading to the formation of separate clusters. Among all models, the RoBERTa-pretrained model achieved the most coherent and specific clusters. It balanced specificity without overgeneralizing based on human texts present in the logs. We attribute this to the model's pretraining on human language, which allowed it to create embeddings informed by this language. This underscores the importance of incorporating human language when performing information extraction on logs, as it significantly impacts the model's performance and explainability.

In our second experiment, we compared the performance of two different approaches: the current TF-IDF + mini-batch k-means approach and our new RoBERTa-pretrained + DBSCAN approach. Our results indicate that the new RoBERTa-pretrained + DBSCAN approach performs on par with the current TF-IDF + mini-batch k-means approach in terms of detecting (malicious) anomalous events. This suggests that our new approach is a viable alternative for anomaly detection in log data. We also assessed the number of false positive peaks generated by each method for a benign host and observed similar performance. Both approaches highlighted the same benign anomalous activities. Interestingly, the increased number of clusters produced by the RoBERTa-pretrained + DBSCAN approach appeared to assist the FlexFringe classifier in detecting anomalies more effectively, resulting in higher peaks. This suggests that allowing more clusters could be advantageous. We speculate that over a longer time period with more diverse logs, the performance of the RoBERTa-pretrained + DBSCAN approach may improve even further. This is because the model can adapt to changing patterns by creating new clusters automatically, whereas the current implementation is limited to a set maximum number of clusters chosen by the model designer. However, this assumption requires further testing and validation. Including extra contextual features did not assist in identifying specific malicious logs in this experiment. This limitation is attributed to the absence of this explorative feature in the FlexFringe APTA Dashboard. The potential effectiveness of these contextual features in highlighting malicious logs may be explored in the future, should they be better integrated into the dashboard. Nevertheless, these features did offer valuable insights by providing indications of when malicious events occurred. This suggests that while they may not directly identify malicious logs, their returned activities can serve as valuable indicators for further investigation.

7.3 Limitations

In this section, we list some limitations that we came across during our research. One of the primary limitations was the absence of clear labels for every log line in the dataset. This absence made it challenging to perform systematic benchmarking and compare model performance objectively using automated metrics like F1-score. The lack of clear labels hindered our ability to assess the models' accuracy effectively.

A second limitation came in the form of training budget and time. Training large transformer models, such as RoBERTa, is computationally expensive and resource-intensive. Due to budget and time constraints, we were limited in terms of the number of training epochs we could afford. This limitation also restricted our ability to perform extensive preprocessing and hyperparameter optimization, potentially impacting the final model's performance and the overall approach.

A third limitation was the time required for hyperparameter tuning of the clustering algorithm, specifically the epsilon parameter in DBSCAN. Although we aimed to optimize this parameter using the Density-Based Clustering Validation (DBCV) metric from Moulavi et al. (2014), practical constraints with the large volume of logs and files, led to prolonged execution times for this metric. Consequently, we could only work with the default value for epsilon, missing the opportunity to achieve well-optimized clustering based on the DBCV metric.

The last and fourth limitation arose from file size and hardware constraints. While attempting to process the SRV-CALYPSO host machine's logs, which were approximately three times larger than the average host log files, we encountered challenges. Our code was unable to process this host due to limitations in available RAM capacity, leading to inconsistent crashes. Future work may involve optimizing the code for more efficient processing or upgrading hardware to handle larger log files.

7.4 Recommendations for Future Research

Regarding future research, there are several areas that might be interesting to explore further to improve log-based anomaly detection methodologies.

A first recommendation would be to apply the methodology to different log sources besides EVTX files. These could involve sources such as Linux system logs or Android logs. Additionally, it would be interesting to investigate how to create source-agnostic language models, enabling their application across various log sources.

Secondly, a different log parser could be used to get better and cleaner features into the log string that is sent to the language model. In this research, we used EvtxECmd and were limited by the features the parser returned. Perhaps there are better parsers out there that can retrieve more and cleaner information from the logs, which would improve our language model's understanding of the log contents.

A third suggestion is to implement improved / more recent language models to extract information from the logs. During the writing of this thesis, the development of Large Language Models increased rapidly with the introduction of ChatGPT. It would be interesting to see how these newer and improved language models can be used in the pipeline as a replacement and improvement of roberta-base.

Hyperparameter tuning is also one of the areas worth investigating. Experimentation with different hyperparameters within the models, including parameters like the epsilon value in the DBSCAN clustering algorithm, could lead to improved clustering and anomaly detection outcomes.

Furthermore, exploring different data preprocessing techniques to enhance data quality and cleanliness before input into language models is desirable.

Lastly, considering alternative clustering algorithms within the log analysis pipeline could be explored as well. While DBSCAN was selected for specific reasons, exploring other clustering methods may return new insights into potential performance enhancements.

8 Conclusion

This thesis aimed to tackle the problem of coming up with an effective method for detecting anomalous behavior performed by adversaries on a system through the use of log messages. It focussed on coming up with a more complex method that better captures the relations between the features in the logs than the currently implemented traditional method called TF-IDF, which was already implemented in an anomaly detection pipeline. The aim of this research was to answer the following main research question:

How can embeddings and language models capture useful representations of log files for analysis in Cybersecurity?

The main research question was answered through four sub-questions:

SQ1 *What features from logs are relevant to train language models?*

SQ2 *Which language model and what training method are most suited for feature extraction?*

SQ3 *How does a transformer language model compare to a weighted word representation model as a feature extraction method in anomaly detection?*

SQ4 *Does adding additional contextual features improve anomaly detection?*

First, data was collected from hosts and we explored which features are relevant to train a language model and how they should be cleaned, processed and anonymized to remove unique identifiers and prevent bias. This data was then used to construct feature strings and create a dataset ready for the purpose of training the language models.

Secondly, we explored the use of the all-minilm-l6-v2 and RoBERTa architectures to see which model and training method is best suited for the purpose of feature extraction. We trained two RoBERTa language models with log data and performed several experiments to test which model created the most consistent and informative clusters. The RoBERTa-pretrained model emerged as the most effective in creating meaningful clusters, while the other models exhibited challenges in grouping logs optimally. The findings emphasize the necessity of including human language understanding when developing models for log analysis. Additionally, the results showed the importance of tuning clustering parameters for optimal results. All in all, we can say that inclusion of log files in the training process does not necessarily lead to a better understanding and grouping of log events.

Thirdly, the RoBERTa-pretrained model was then used together with DBSCAN clustering in a comparison against the current TF-IDF feature extraction method with Mini-batch K-means clustering implementation. Using this information as input to FlexFringe with these features, we observed that the approaches achieved similar performance in identifying (malicious) anomalies, thus rejecting our hypothesis that performance of anomaly detection improves using transformer-based language models as a feature extraction method over the baseline TF-IDF weighted word representation model. However, we did notice that, in our new approach, the anomalies were a bit more obvious in the resulting graphs having their peaks stand out more, making them easier to spot and identify.

At last, we observed that addition of extra contextual features did not necessary lead to an improvement in anomaly detection when using the FlexFringe classification model, but it proved to be a good indicator to identify the time period in which malicious actions took place.

This thesis research encountered several limitations throughout the research process. First, the absence of available labels hindered the ability to conduct a comprehensive comparative study, which would have allowed for the utilization of standardized metrics for performance assessment. Second, constraints in training budget and time imposed limitations on the extent of model training, leading to challenges in implementing thorough preprocessing and hyperparameter optimization. Additionally, due to time constraints, hyperparameter tuning for the epsilon parameter in the DBSCAN clustering algorithm could not be done, potentially impacting clustering performance. Lastly, hardware limitations were encountered when processing larger log files, leading to operational constraints.

Future research in this domain could explore several promising areas. These include extending the methodology to be able to handle different log sources, refining preprocessing techniques for cleaner feature extraction, and perhaps making use of the latest advancements in language models for improved performance. Additionally, efficient methods for optimizing critical hyperparameters, such as the epsilon parameter in the DBSCAN clustering algorithm, should be a priority. These directions offer opportunities to further enhance the effectiveness of log-based anomaly detection and contribute to the ongoing evolution of cybersecurity defense and recovery methods.

A
Appendix

A.1 Data Origins

Table A.1 shows the number of logs collected from each host origin and used in the process of training the language models.

File name	Row count
host_1	833.319
host_2	300.060
host_3	325.667
host_4	281.988
host_5	331.512
host_6	380.764
host_7	447.511
host_8	330.138
host_9	358.799
host_10	402.460
host_11	406.756
host_12	264.253
host_13	806.805
host_14	778.741
host_15	735.301
host_16	742.202
host_17	701.329
host_18	800.852
host_19	833.129
host_20	167.143
host_21	235.284
host_22	579.269
host_23	829.784
host_24	682.867
host_25	817.956
host_26	742.598
host_27	803.065
host_28	651.258
host_29	809.154
host_30	864.176
host_31	256.493
host_32	448.973
host_33	231.710
host_34	225.451
host_35	973.142
host_36	997.277
host_37	786.222
host_38	263.225
host_39	406.146
host_40	342.321
host_41	454.427
host_42	237.252
host_43	323.547
host_44	826.146
host_45	789.235
host_46	847.663
host_47	258.210
host_48	355.522
host_49	834.763
host_50	831.360
host_50	831.360
host_51	824.387
host_52	830.400
host_53	793.715
host_54	309.134
host_55	791.540
host_56	220.146
host_57	856.628
host_58	769.635
host_59	699.858
host_60	229.624
host_61	361.189
host_62	239.853
host_63	870.564
host_64	860.102
host_65	803.665
host_66	814.832
host_67	267.736
host_68	246.335
host_69	906.858
host_70	828.866
host_71	775.947
host_72	648.435
host_73	570.579
Total (n=73)	42.453.253

Table A.1: Number of log lines per host, adding up to a total of 42.453.253 logs.

A.2 Cluster Inspection

Here, we provide an overview of two of the cluster inspections that took place in Section 6.1.3. To see other comparisons, we refer you to our GitHub page [1].

A.2.1 Malicious Authentication Requests Inspection Example

Timestamp	ClusterID	EventID	Raw Feature String
2023-01-24 13:17:20.6885240	54	4776	Security.evtx Microsoft-Windows-Security-Auditing Security 4776 2023-01-24 13:17:20.6885240 LogAlways NTLM authentication request DC-APHRODITE.Akropolys.nl {”EventData”:{”Data”:[{”@Name”:”PackageName”,”#text”:”MICROSOFT_AUTHENTICATION_PACKAGE_V1_0”}, {”@Name”:”TargetUserName”,”#text”:”AKROPOLYS\\John”}, {”@Name”:”Workstation”,”#text”:”kali”},{”@Name”:”Status”,”#text”:”0xC0000064”}]}}
2023-01-24 13:27:38.9737497	54	4776	Security.evtx Microsoft-Windows-Security-Auditing Security 4776 2023-01-24 13:27:38.9737497 LogAlways NTLM authentication request DC-APHRODITE.Akropolys.nl {”EventData”:{”Data”:[{”@Name”:”PackageName”,”#text”:”MICROSOFT_AUTHENTICATION_PACKAGE_V1_0”}, {”@Name”:”TargetUserName”,”#text”:”John”}, {”@Name”:”Workstation”,”#text”:”kali”},{”@Name”:”Status”,”#text”:”0xC000006A”}]}}
2023-01-24 13:37:41.0740706	54	4776	Security.evtx Microsoft-Windows-Security-Auditing Security 4776 2023-01-24 13:37:41.0740706 LogAlways NTLM authentication request DC-APHRODITE.Akropolys.nl {”EventData”:{”Data”:[{”@Name”:”PackageName”,”#text”:”MICROSOFT_AUTHENTICATION_PACKAGE_V1_0”}, {”@Name”:”TargetUserName”,”#text”:”John”}, {”@Name”:”Workstation”,”#text”:”KALI”},{”@Name”:”Status”,”#text”:”0x0”}]}}
2023-01-24 13:13:24.6834688	54	4776	Security.evtx Microsoft-Windows-Security-Auditing Security 4776 2023-01-24 13:13:24.6834688 LogAlways NTLM authentication request DC-APHRODITE.Akropolys.nl {”EventData”:{”Data”:[{”@Name”:”PackageName”,”#text”:”MICROSOFT_AUTHENTICATION_PACKAGE_V1_0”}, {”@Name”:”TargetUserName”,”#text”:”AKROPOLYS\\John”}, {”@Name”:”Workstation”,”#text”:”remnux”},{”@Name”:”Status”,”#text”:”0xC0000064”}]}}
2023-01-24 13:11:15.9309497	96	4776	Security.evtx Microsoft-Windows-Security-Auditing Security 4776 2023-01-24 13:11:15.9309497 LogAlways NTLM authentication request DC-APHRODITE.Akropolys.nl {”EventData”:{”Data”:[{”@Name”:”PackageName”,”#text”:”MICROSOFT_AUTHENTICATION_PACKAGE_V1_0”}, {”@Name”:”TargetUserName”,”#text”:”John”}, {”@Name”:”Workstation”,”#text”:”EYE-4S1FGK3”},{”@Name”:”Status”,”#text”:”0xC000006A”}]}}
2023-01-24 13:18:14.8055269	96	4776	Security.evtx Microsoft-Windows-Security-Auditing Security 4776 2023-01-24 13:18:14.8055269 LogAlways NTLM authentication request DC-APHRODITE.Akropolys.nl {”EventData”:{”Data”:[{”@Name”:”PackageName”,”#text”:”MICROSOFT_AUTHENTICATION_PACKAGE_V1_0”}, {”@Name”:”TargetUserName”,”#text”:”john”}, {”@Name”:”Workstation”,”#text”:”EYE-4S1FGK3”},{”@Name”:”Status”,”#text”:”0xC000006A”}]}}
2023-01-23 14:32:31.7939953	96	4776	Security.evtx Microsoft-Windows-Security-Auditing Security 4776 2023-01-23 14:32:31.7939953 LogAlways NTLM authentication request DC-APHRODITE.Akropolys.nl {”EventData”:{”Data”:[{”@Name”:”PackageName”,”#text”:”MICROSOFT_AUTHENTICATION_PACKAGE_V1_0”}, {”@Name”:”TargetUserName”,”#text”:”Aphrodite”}, {”@Name”:”Workstation”,”#text”:”EYE-4S1FGK3”},{”@Name”:”Status”,”#text”:”0x0”}]}}
2023-01-24 13:53:51.7562240	110	4776	Security.evtx Microsoft-Windows-Security-Auditing Security 4776 2023-01-24 13:53:51.7562240 LogAlways NTLM authentication request DC-APHRODITE.Akropolys.nl {”EventData”:{”Data”:[{”@Name”:”PackageName”,”#text”:”MICROSOFT_AUTHENTICATION_PACKAGE_V1_0”}, {”@Name”:”TargetUserName”,”#text”:”John”}, {”@Name”:”Workstation”},{”@Name”:”Status”,”#text”:”0x0”}]}}
2023-01-23 03:03:51.0982386	148	4776	Security.evtx Microsoft-Windows-Security-Auditing Security 4776 2023-01-23 03:03:51.0982386 LogAlways NTLM authentication request DC-APHRODITE.Akropolys.nl {”EventData”:{”Data”:[{”@Name”:”PackageName”,”#text”:”MICROSOFT_AUTHENTICATION_PACKAGE_V1_0”}, {”@Name”:”TargetUserName”,”#text”:”HealthMailbox2f65eb98e79345249447f4860e723308@Akropolys.nl”}, {”@Name”:”Workstation”,”#text”:”EXC-CALYPSO”}, {”@Name”:”Status”,”#text”:”0x0”}]}}
2023-01-22 11:13:38.2607764	188	4776	Security.evtx Microsoft-Windows-Security-Auditing Security 4776 2023-01-22 11:13:38.2607764 LogAlways NTLM authentication request DC-APHRODITE.Akropolys.nl {”EventData”:{”Data”:[{”@Name”:”PackageName”,”#text”:”MICROSOFT_AUTHENTICATION_PACKAGE_V1_0”}, {”@Name”:”TargetUserName”,”#text”:”SRV-HERCULES$”},{”@Name”:”Workstation”,”#text”:”SRV-HERCULES”}, {”@Name”:”Status”,”#text”:”0x0”}]}}

Table A.2: Clusters inspection of malicious authentication requests (Security.evtx + eventID 4776) from user John on DC-APHRODITE in the Azure AD Labs (Test 1) dataset clustered by RoBERTa-pretrained.

[1] https://github.com/TimG-NL/cs_master_thesis

A.2.2 Benign Program Installation Cluster Inspection Example

Timestamp	ClusterID	EventID	Raw Feature String
2023-01-10 15:12:08.1390407	6	1033	Application.evtx MsiInstaller Application 1033 2023-01-10 15:12:08.1390407 Info A program was installed DC-APHRODITE.Akropolys.nl S-1-5-21-894095181-724868689-229023573-500 {”EventData”:{”Data”:”vs_communityx64msi, 17.4.33006, 1033, 0, Microsoft Corporation, (NULL)”,”Binary”:”...”}}
2023-01-10 15:12:01.0081446	6	1033	Application.evtx MsiInstaller Application 1033 2023-01-10 15:12:01.0081446 Info A program was installed DC-APHRODITE.Akropolys.nl S-1-5-21-894095181-724868689-229023573-500 {”EventData”:{”Data”:”vs_minshellinteropsharedmsi, 17.4.33006, 1033, 0, Microsoft Corporation, (NULL)”,”Binary”:”...”}}
2023-01-10 10:53:01.2131308	10	1033	Application.evtx MsiInstaller Application 1033 2023-01-10 10:53:01.2131308 Info A program was installed DC-APHRODITE.Akropolys.nl S-1-5-21-894095181-724868689-229023573-500 {”EventData”:{”Data”:”Python 3.10.9 Utility Scripts (64-bit), 3.10.9150.0, 1033, 0, Python Software Foundation, (NULL)”,”Binary”:”...”}}
2023-01-10 10:53:07.6251013	10	1033	Application.evtx MsiInstaller Application 1033 2023-01-10 10:53:07.6251013 Info A program was installed DC-APHRODITE.Akropolys.nl S-1-5-21-894095181-724868689-229023573-500 {”EventData”:{”Data”:”Python 3.10.9 Tcl/Tk Support (64-bit), 3.10.9150.0, 1033, 0, Python Software Foundation, (NULL)”,”Binary”:”...”}}
2023-01-10 15:18:03.1941393	13	1033	Application.evtx MsiInstaller Application 1033 2023-01-10 15:18:03.1941393 Info A program was installed DC-APHRODITE.Akropolys.nl S-1-5-21-894095181-724868689-229023573-500 {”EventData”:{”Data”:”Windows SDK for Windows Store Apps DirectX x86 Remote, 10.1.18362.1, 1033, 0, Microsoft Corporation, (NULL)”,”Binary”:”...”}}
2023-01-10 15:17:16.7262098	13	1033	Application.evtx MsiInstaller Application 1033 2023-01-10 15:17:16.7262098 Info A program was installed DC-APHRODITE.Akropolys.nl S-1-5-21-894095181-724868689-229023573-500 {”EventData”:{”Data”:”Windows SDK for Windows Store Apps Contracts, 10.1.18362.1, 1033, 0, Microsoft Corporation, (NULL)”,”Binary”:”...”}}

Table A.3: Benign activity log inspection on DC-APHRODITE in the Azure AD Labs (Test 1) dataset clustered by RoBERTa-pretrained.

A.3 FlexFringe Anomaly Detection Results

A.3.1 TF-IDF + Mini-batch K-means FlexFringe output on DC-APHRODITE Azure AD Labs (Test 1)

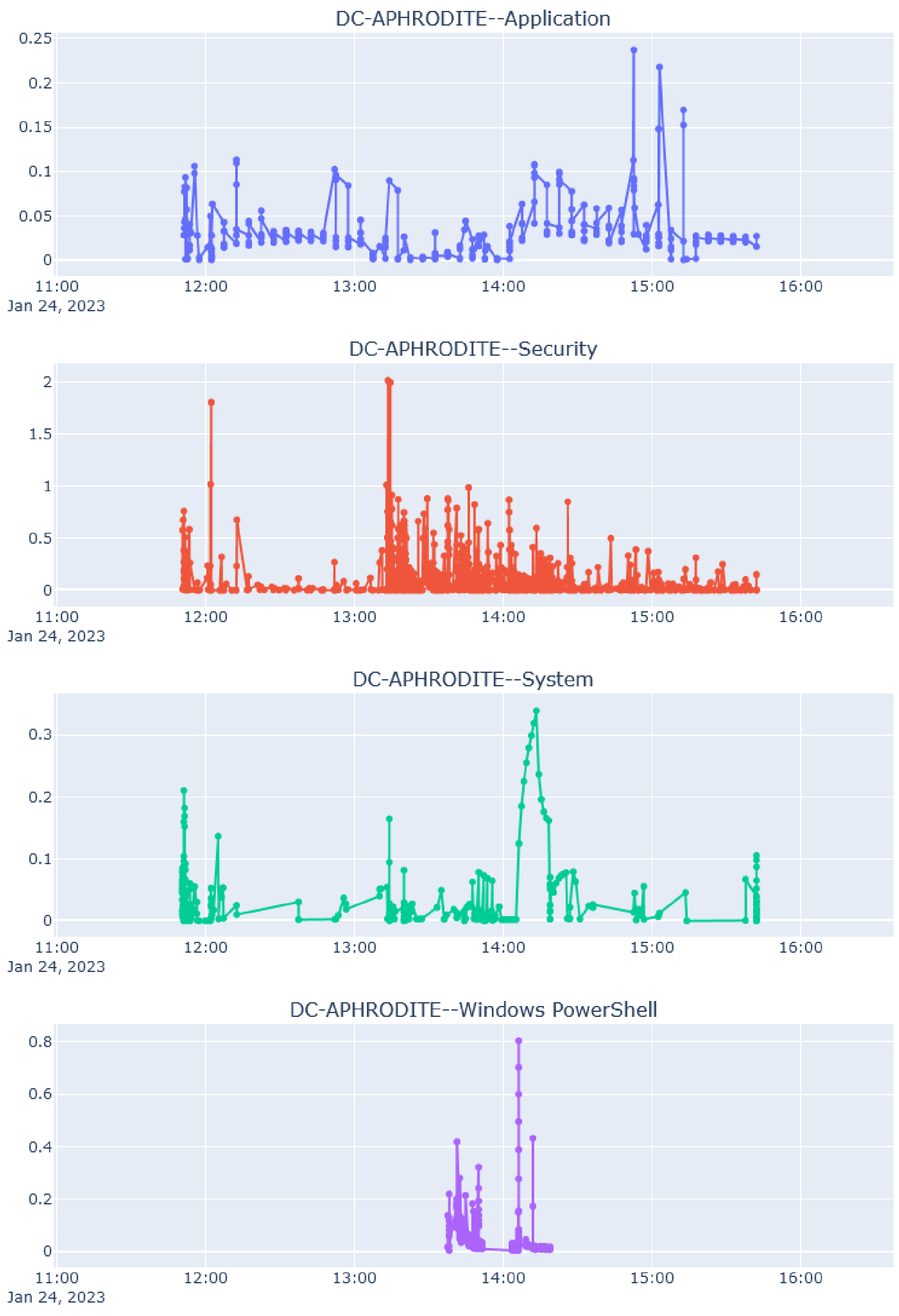


Figure A.1: FlexFringe graphs derived from the TF-IDF + Mini-batch K-means approach on DC-APHRODITE from Azure AD Labs (Test 1)

A.3.2 RoBERTa-pretrained + DBSCAN FlexFringe output on DC-APHRODITE Azure AD Labs (Test 1)

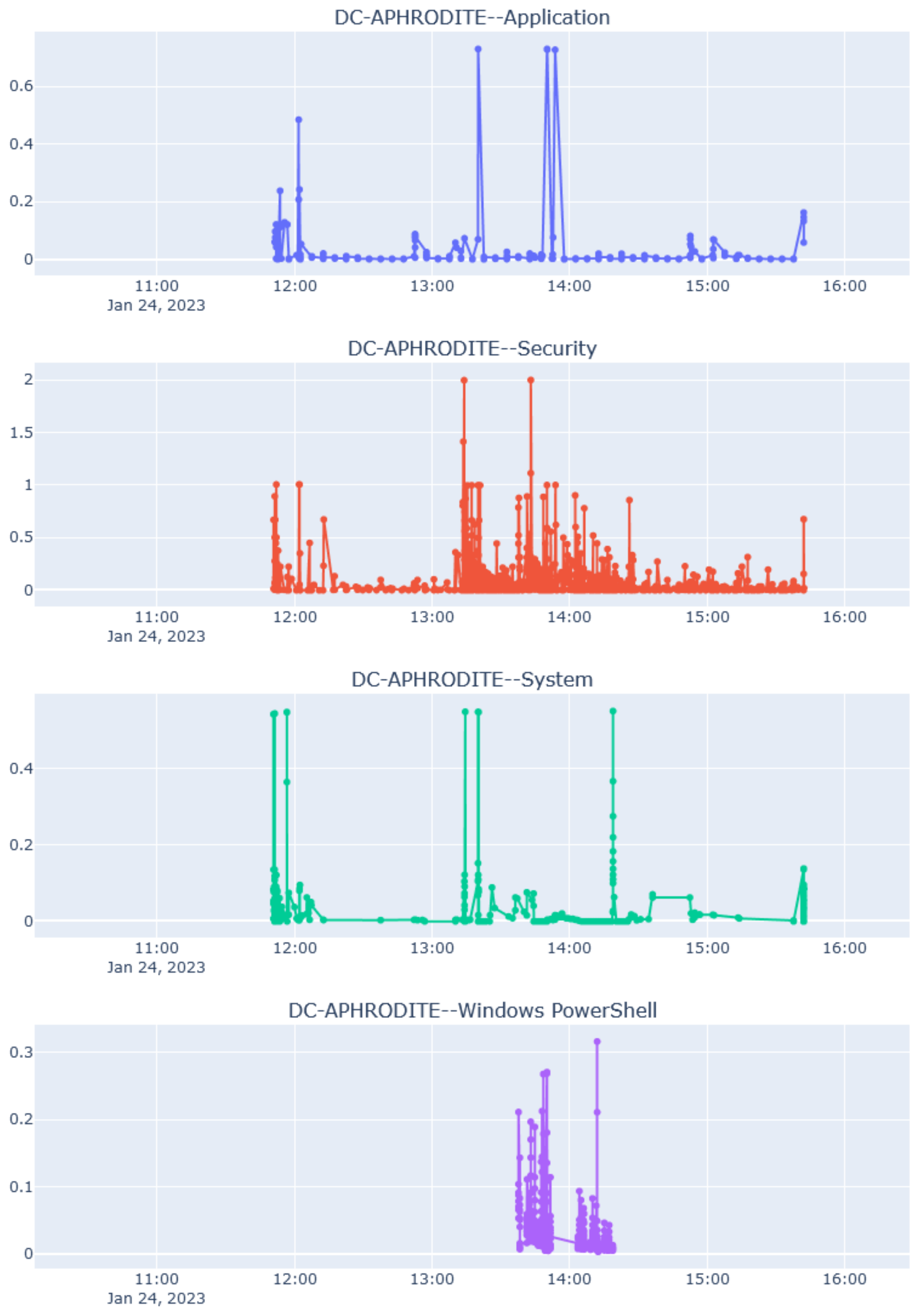


Figure A.2: FlexFringe graphs derived from the RoBERTa-pretrained + DBSCAN approach on DC-APHRODITE from Azure AD Labs (Test 1)

A.4 Extra Features Figures

A.4.1 Case April 2022: Infected Host

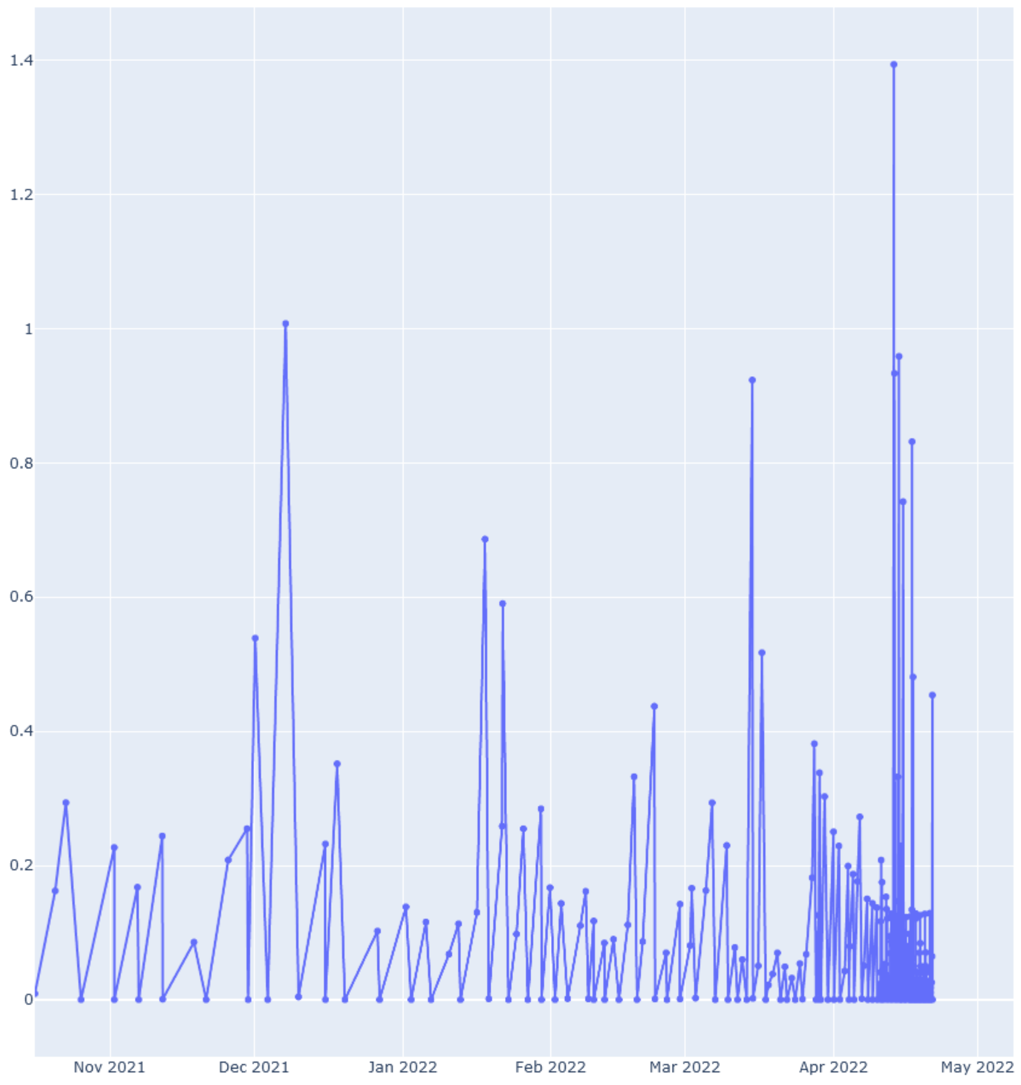


Figure A.3: Anomaly detection graph of the extra generated contextual features for the infected host for the Case April 2022 dataset.

A.5 FlexFringe Benign Host Investigation Results

A.5.1 TF-IDF + Mini-batch K-means FlexFringe output on WKS-FROUKJE Azure AD Labs (Test 1)

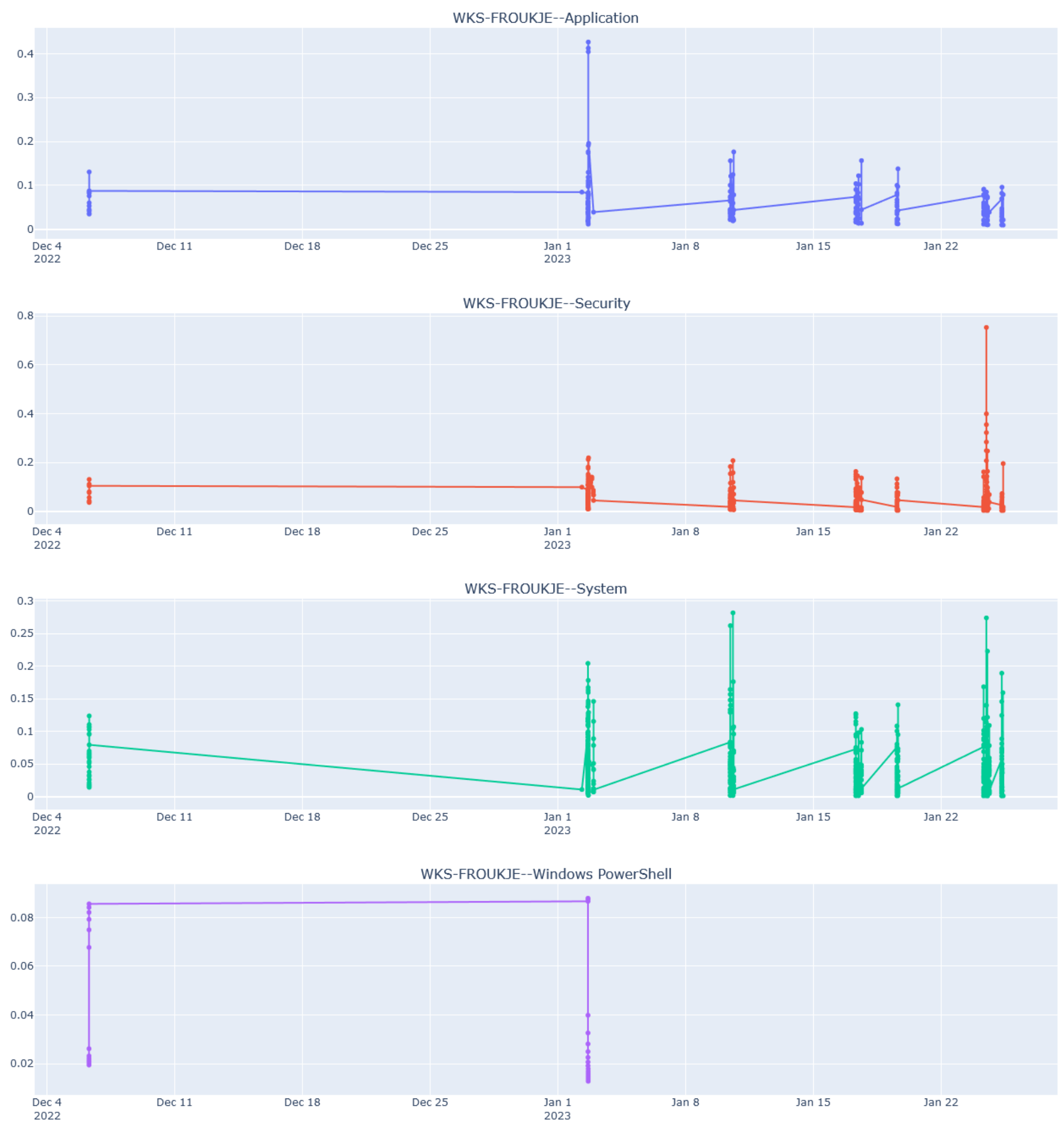


Figure A.4: FlexFringe graphs derived from the TF-IDF + Mini-batch K-means approach on WKS-FROUKJE from Azure AD Labs (Test 1)

A.5.2 RoBERTa-pretrained + DBSCAN FlexFringe output on WKS-FROUKJE Azure AD Labs (Test 1)

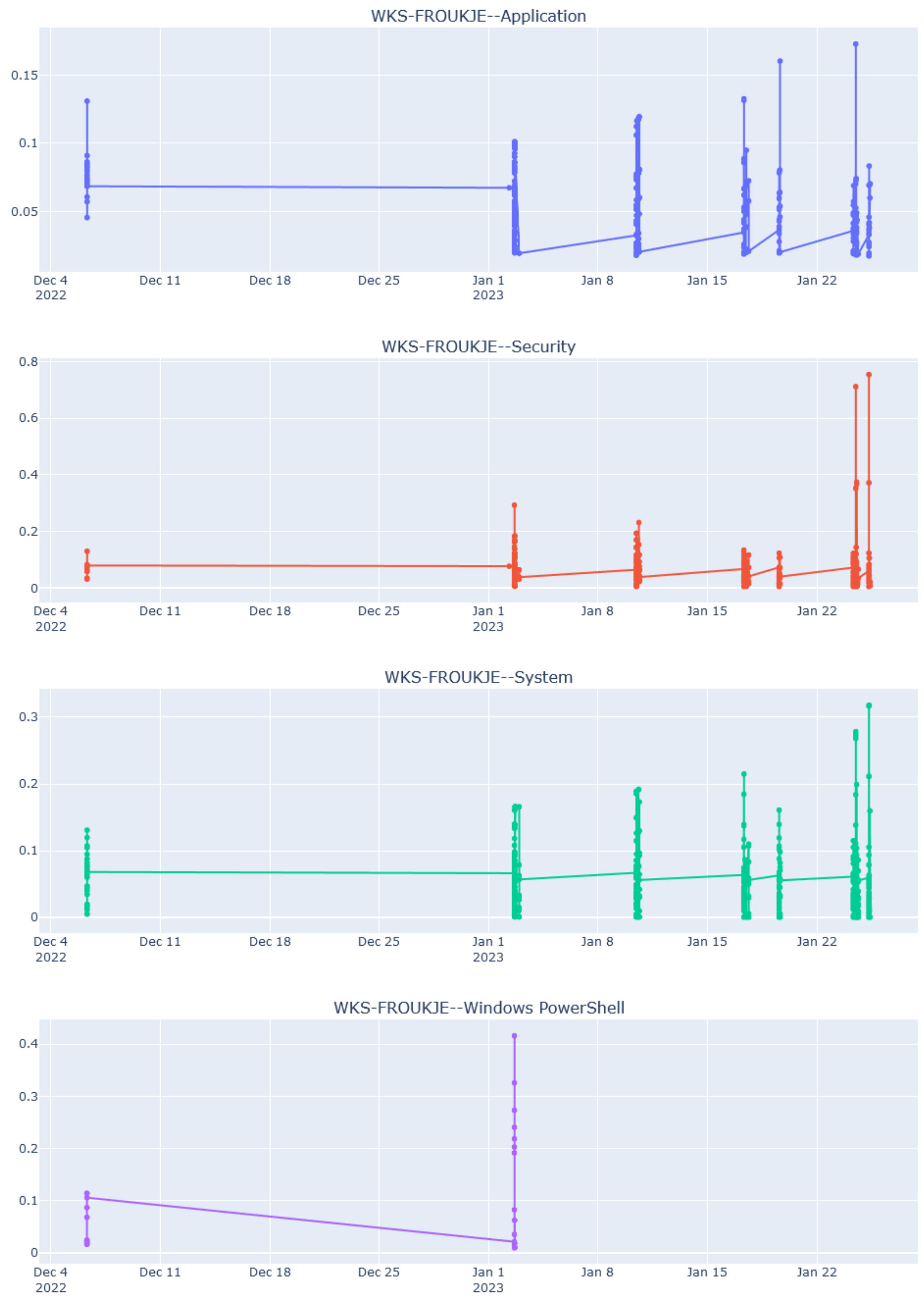


Figure A.5: FlexFringe graphs derived from the RoBERTa-pretrained + DBSCAN approach on WKS-FROUKJE from Azure AD Labs (Test 1)

Bibliography

Ankit, U. (2022, Jun). Transformer neural networks: A step-by-step breakdown. https://builtin.com/artificial-intelligence/transformer-neural-network.

Baum, L. E. and T. Petrie (1966). Statistical Inference for Probabilistic Functions of Finite State Markov Chains. *The Annals of Mathematical Statistics 37*(6), 1554 – 1563.

Bengio, Y., R. Ducharme, and P. Vincent (2000). A neural probabilistic language model. *Advances in neural information processing systems 13*.

Bradley, S. (2020, Jun). The most important windows 10 security event log ids to monitor.

Brown, T., B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei (2020). Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin (Eds.), *Advances in Neural Information Processing Systems*, Volume 33, pp. 1877–1901. Curran Associates, Inc.

Camacho-Collados, J. and M. T. Pilehvar (2017). On the role of text preprocessing in neural network architectures: An evaluation study on text categorization and sentiment analysis. *arXiv preprint arXiv:1707.01780*.

Charter, B. (2008). Evtx and windows event logging. *SANS Institute, InfoSec Reading Room*.

Devlin, J., M.-W. Chang, K. Lee, and K. Toutanova (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Duboue, P. (2020). *The art of feature engineering: essentials for machine learning*. Cambridge University Press.

Egersdoerfer, C., D. Zhang, and D. Dai (2022). Clusterlog: Clustering logs for effective log-based anomaly detection. In *2022 IEEE/ACM 12th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*, pp. 1–10. IEEE.

Ester, M., H.-P. Kriegel, J. Sander, X. Xu, et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, Volume 96, pp. 226–231.

Geng, X. and H. Liu (2023, May). Openllama: An open reproduction of llama.

Guo, H., S. Yuan, and X. Wu (2021). Logbert: Log anomaly detection via bert. In *2021 international joint conference on neural networks (IJCNN)*, pp. 1–8. IEEE.

Hammouchi, H., O. Cherqi, G. Mezzour, M. Ghogho, and M. El Koutbi (2019). Digging deeper into data breaches: An exploratory data analysis of hacking breaches over time. *Procedia Computer Science 151*, 1004–1009.

Jaccard, P. (1901). Distribution de la flore alpine dans le bassin des dranses et dans quelques régions voisines. *Bull Soc Vaudoise Sci Nat 37*, 241–272.

Kampakis, S. (2022). 3 Types of Anomalies in Anomaly Detection | HackerNoon — hackernoon.com. https://hackernoon.com/3-types-of-anomalies-in-anomaly-detection. [Accessed 14-Mar-2023].

Karl-Bridge-Microsoft (2021, Aug). Channeltype complex type - win32 apps.

Lee, Y., J. Kim, and P. Kang (2021). Lanobert: System log anomaly detection based on bert masked language model. *arXiv preprint arXiv:2111.09564*.

Liu, Y., M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov (2019). Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

MacQueen, J. (1967). Classification and analysis of multivariate observations. In *5th Berkeley Symp. Math. Statist. Probability*, pp. 281–297. University of California Los Angeles LA USA.

McInnes, L. and J. Healy (2017). Accelerated hierarchical density based clustering. In *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, pp. 33–42. IEEE.

Menon, P. (2023, Mar). Introduction to large language models and the transformer architecture. https://rpradeepmenon.medium.com/introduction-to-large-language-models-and-the-transformer-architecture-534408ed7e6

Merriam-Webster (2023). Anomaly. https://www.merriam-webster.com/dictionary/anomaly.

Moulavi, D., P. A. Jaskowiak, R. J. Campello, A. Zimek, and J. Sander (2014). Density-based clustering validation. In *Proceedings of the 2014 SIAM international conference on data mining*, pp. 839–847. SIAM.

Mullen, T. and N. Collier (2004). Sentiment analysis using support vector machines with diverse information sources. In *Proceedings of the 2004 conference on empirical methods in natural language processing*, pp. 412–418.

Mysiak, K. (2020, Jul). Explaining dbscan clustering.

Na, Y. (2021). Finite State Machine | nana.log — yrnana.dev. https://yrnana.dev/post/2021-03-14-finite-state-machine/. [Accessed 20-09-2023].

Nguyen, C. (2022). Sad: State machine-based anomaly detection in user behavior.

nxlog (2022). Nxlog documentation.

OpenAI (2023). Gpt-4 technical report.

Radford, A., K. Narasimhan, T. Salimans, I. Sutskever, et al. (2018). Improving language understanding by generative pre-training.

Radford, A., J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. (2019). Language models are unsupervised multitask learners. *OpenAI blog 1*(8), 9.

Rajasekharaiah, K., C. S. Dule, and E. Sudarshan (2020). Cyber security challenges and its emerging trends on latest technologies. In *IOP Conference Series: Materials Science and Engineering*, Volume 981, pp. 022062. IOP Publishing.

Rajendran, G., R. R. Nivash, P. P. Parthy, and S. Balamurugan (2019). Modern security threats in the internet of things (iot): Attacks and countermeasures. In *2019 International Carnahan Conference on Security Technology (ICCST)*, pp. 1–6. IEEE.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). Learning representations by back-propagating errors. *nature 323*(6088), 533–536.

Sculley, D. (2010). Web-scale k-means clustering. In *Proceedings of the 19th international conference on World wide web*, pp. 1177–1178.

Sennrich, R., B. Haddow, and A. Birch (2015). Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Sharif, A. (2021, Dec). What is an event log? contents and use - crowdstrike.

SolarWinds (2023). What Is a Windows Event Log? - IT Glossary | SolarWinds — solarwinds.com. https://www.solarwinds.com/resources/it-glossary/windows-event-log. [Accessed 14-Feb-2023].

Sparck Jones, K. (1972). A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation 28*(1), 11–21.

Steinhaus, H. et al. (1956). Sur la division des corps matériels en parties. *Bull. Acad. Polon. Sci 1*(804), 801.

Toman, M., R. Tesar, and K. Jezek (2006). Influence of word normalization on text classification. *Proceedings of InSciT 4*, 354–358.

Touvron, H., T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. (2023). Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Van der Maaten, L. and G. Hinton (2008). Visualizing data using t-sne. *Journal of machine learning research 9*(11).

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin (2017). Attention is all you need. *Advances in neural information processing systems 30*.

Verdonck, T., B. Baesens, M. Óskarsdóttir, and S. vanden Broucke (2021). Special issue on feature engineering editorial. *Machine Learning*, 1–12.

Verwer, S. and C. A. Hammerschmidt (2017). Flexfringe: a passive automaton learning package. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 638–642. IEEE.

Wolf, T., L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush (2020, October). Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Online, pp. 38–45. Association for Computational Linguistics.

Yasar, K. and A. S. Gillis (2023, Mar). What is windows event log?: Definition from techtarget.

Zhang, W., Q. Yang, and Y. Geng (2009). A survey of anomaly detection methods in networks. In *2009 International Symposium on Computer Network and Multimedia Technology*, pp. 1–3. IEEE.