Natural Language Processing with machine learning for anomaly detection on system call logs

Submitted in partial fulfilment
of the requirements for the degree of

Master of Science

of Rhodes University

Christo Goosen

Grahamstown, South Africa
August 2023

Abstract

Host intrusion detection systems and machine learning have been studied for many years especially on datasets like KDD99. Current research and systems are focused on low training and processing complex problems such as system call returns, which lack the system call arguments and potential traces of exploits run against a system. With respect to malware and vulnerabilities, signatures are relied upon, and the potential for natural language processing of the resulting logs and system call traces needs further experimentation.

This research looks at unstructured raw system call traces from x86_64 bit GNU Linux operating systems with natural language processing and supervised and unsupervised machine learning techniques to identify current and unseen threats. The research explores whether these tools are within the skill set of information security professionals, or require data science professionals.

The research makes use of an academic and modern system call dataset from Leipzig University and applies two machine learning models based on decision trees. Random Forest as the supervised algorithm is compared to the unsupervised Isolation Forest algorithm for this research, with each experiment repeated after hyper-parameter tuning.

The research finds conclusive evidence that the Isolation Forest Tree algorithm is effective, when paired with a Principal Component Analysis, in identifying anomalies in the modern Leipzig Intrusion Detection Data Set (LID-DS) dataset combined with samples of executed malware from the Virus Total Academic dataset. The base or default model parameters produce sub-optimal results, whereas using a hyper-parameter tuning technique increases the accuracy to within promising levels for anomaly and potential zero day detection.

Acknowledgements

I would like to refer to a Nigerian Igbo proverb "It takes a village to raise a child!" [1]. Indeed it feels like it took a village of Rhodes MSc staff, my family and some previous colleagues and friends for me to have finished this masters. I dedicate this to all who stuck by me during this challenge.

My wife, Tiana Goosen, who slept many a night alone, while I typed away at this masters.

Family for backing me and for patience.

Alan Herbert for supervising me through a very busy and difficult time for focus.

Karen Bradshaw for being a phenomenal supervisor and very patient and wise counsel on this journey.

Liam Smit for advising, spell checking and for reminding me to do my work.

Virus Total for giving me access to the academic dataset, which included a large number of Linux malware samples.

The group behind the *Leipzig Intrusion Detection Data Set*, for the excellent dataset used in this research.

[1] http://www.ngopulse.org/article/it-takes-village-raise-child

Glossary

AI artificial intelligence. 48

API application programming interface. 14, 21

AUC area under the curve. 36, 38, 72, 73, 75, 77, 92

BoSC bag of system calls. 30, 31, 37, 38

botnets A network of bots (Yamaguchi, 2020). 18

BOW Bag-Of-Words. x, 29, 30, 31, 32, 60

c&c command-and-control servers. 20, 46

CIA confidentiality integrity availability. 6, 7

CVE Common vulnerabilities and exposures. 51

DARPA Defense Advanced Research Projects Agency. 1

DoS denial of service attack. 7, 15, 20, 24

ELF executable and linking format. 55

HIDS host intrusion detection system. xii, 51, 53

IDS intrusion detection systems. 1, 2, 3, 7, 8, 9, 10, 11, 12, 22, 23, 25, 31, 41

IPC inter-process communication. 22

JSON Javascript object notation. xii, 55

LID-DS Leipzig Intrusion Detection Data Set. i, xii, 4, 40, 42, 50, 51, 52, 57, 69, 94

MAC message authentication code. 21, 22

MD5 message-digest algorithm 5. 17, 54

ML machine learning. 28, 33

NAT network address translation. 19

NLP natural language processing. 1, 3, 4, 6, 26, 27, 40, 41, 44, 49, 58, 59, 60, 93, 94, 95

PCA principal component analysis. 42, 49, 62, 63, 64, 66, 67, 95, 97

RAM random access memory. 50

regex regular expressions. 58

ROC receiver operating characteristic. 72, 73, 88

seccomp secure computing mode. 22

SHA256 secure hash algorithm 256-bit. 54

sinkholed Manipulating data flow in a network; traffic is redirected from its intended destination to the server of choice (Newman, 2018). 19

STP statistical pattern-based feature extraction method. 37, 38

syscall system call. 25, 41, 47

tf-idf term frequency–inverse document frequency. ix, x, 27, 28, 29, 32, 38, 49, 60, 61, 62, 63, 64, 66, 67, 91, 94, 95, 96, 97

tokenizer Tokenization is a way of separating a piece of text into smaller units called tokens. 28

Unix a family of multitasking, multi-user computer operating systems that derive from the original AT&T Unix. 55

VirusTotal VirusTotal is a website created by the Spanish security company Hispasec Sistemas. ix, xii, 41, 46, 47, 53, 54, 55, 56, 69

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context of the Research

The concept of applying machine learning and data mining techniques in information security has been researched for a couple of decades. This goes back to when Denning (1986) proposed using audit records to detect anomalies and when, in 1998, the Defense Advanced Research Projects Agency (DARPA)[1] was testing intrusion detection systems (IDS) with training data and machine learning (Lippmann *et al.*, 2000). Though early studies by researchers like Denning were correct in their hypothesis that "exploitation of a system's vulnerabilities involves abnormal use...security violations could be detected from abnormal patterns of system usage", the hardware limitations strained these efforts (Denning, 1986).

The increasing adoption of cloud resources and the digitization of business critical systems are matched with increasing information security threats. Increasing threats are managed with intrusion detection, but the sheer number of events in modern systems, does not scale with the small agile teams favoured in fast moving environments. The further adoption of techniques like anomaly detection and natural language processing of network data provides scalable network detection, but device execution still remains an area in need of novel applications of the same techniques.

Natural language processing (NLP) provides new possibilities in security as it is capable of "computationally interpreting written language" (Shropshire, 2018). Audit and development logs in computers follow human readable formats which provide a great opportunity for applying NLP. An NLP IDS could allow the detection of advanced persistent threats or zero days, as the algorithms classify and detect previously unknown vectors of attack in systems (Thejaswini and Indupriya, 2019). NLP and machine learning systems are also able to scale up beyond the abilities of current information security teams.

NLP can be used to enhance the existing capabilities of small skilled teams, while growing the number of available systems. Further development of hardware and techniques like NLP gives security researchers and practitioners new capabilities to detect anomalies in human readable logs.

[1] https://www.ll.mit.edu/r-d/datasets/1999-darpa-intrusion-detection-evaluation-dataset

1.2 Research Motivation

The motivation for this research is well articulated in one of the earliest studies of an automated IDS. Denning (1986) stipulated four factors for consideration in a real-time and automated IDS.

The first factor was stated as follows (Denning, 1986, p.118): "most existing systems have security flaws that render them susceptible to intrusions, penetrations, and other forms of abuse; finding and fixing all these deficiencies is not feasible for technical and economic reasons". Any information security system or policy is subject to the economic and business priority limitations, and thus wholesale replacing of a company's infrastructure and applications to remove all threats, is not an economically viable solution (Denning, 1986).

The second factor as identified by Denning (1986) is that current system's replacement is hampered by missing *attractive* features, inherent in the current system. Known flaws are tolerated in current systems as they might not be able to be replaced for economic reasons for example, when features present potential revenue over security (Denning, 1986). Denning is well aware in the first two factors that economic and profit considerations would overrule a business' perceived need for total security.

The third factor is a well known fact in information security, in that "developing systems that are absolutely secure is extremely difficult, if not generally impossible" (Denning, 1986). Herley and Oorschot (2017) make a case for growing calls to make security and security research more *scientific*, with added difficulties of applying the scientific method to develop secure systems. With a lack of consensus on security as a *scientific pursuit*, to apply a theory or hypothesis of a secure system would be difficult (Herley and Oorschot, 2017). To adhere to the minimal standards for a scientific discipline, a science of security would need consistency, falsifiability and predictive power and progress (Herley and Oorschot, 2017). The pursuit of a perfectly secure system is further refuted by a lack of rigid science and a statement like "inconsistent or unfalsifiable theory, providing neither new understandings nor accurate predictions, or a field that does not progress, is unscientific" (Herley and Oorschot, 2017, p.103).

Denning (1986, p.118) states as her fourth factor that "even the most secure systems are vulnerable to abuses by insiders who misuse their privileges". She correctly identifies the insider threat, but further that the operating system and other auxiliary systems could post a threat as well. The insider threat further highlights the need for an IDS and the detection of anomalies. As stated by Denning (1986, p.118) "exploitation of a system's

vulnerabilities involves abnormal use, of the system", a legitimate user of the system could exploit vulnerabilities, which would be detected as abnormal to the usual patterns of usage.

One of Denning's goals for the research into an intrusion detection model is stated as: "The model is independent of any particular system, application environment, system vulnerability, or type of intrusion, thereby providing a framework for a general-purpose intrusion-detection expert system" (Denning, 1986, p.118). This highlights two motivations for NLP in an IDS: firstly being applied to more general systems and logs; and secondly enabling expert involvement instead of full automation. The enhancement of existing security staff is highlighted in a potential weakness identified by Denning (1986, p.130): "Although we believe that the approach can detect most intrusions, it may be possible for a person to escape detection through gradual modifications of behavior or through subtle forms of intrusion that use low-level features of the target system that are not monitored (because they would produce too much data)".

1.3 Research Statement

The aim of this research is to determine whether NLP techniques can be applied to logs, such as syscall logs, to classify and identify anomalies in previously undetected incidents or datasets.

These datasets would have used signature based intrusion detection, but could provide training data to group and classify anomalies for human intervention. Unsupervised machine learning models can detect previously unseen events and do not require a large amount of expertise beyond existing knowledge workers. Thus, we hypothesize that this can be done.

The hypothesis will be tested with the following research questions:

1. Can NLP detect anomalies in human readable logs, such as system call traces?
2. Can NLP assist in detecting previously undetected anomalies in applications, with easy-to-maintain and train models?
3. Can NLP detect malicious activity or malware from system call traces?
4. Which of unsupervised or supervised machine learning models produces more accurate results?

5. Are unsupervised machine learning models able to detect unseen incidents?
6. Are Isolation Forest, Random Forest or both models necessary to detect anomalies?
7. Are security professionals able to use the methods of NLP and tree model anomaly detection without needing to be machine learning experts?

1.4 Research Objectives

The objectives of this research are threefold:

1. Investigate system calls as human language using NLP to tokenize and classify unstructured log data.
2. Investigate what the most effective classification algorithm is to answer the research questions.
3. Detect and identify anomalies in the form of security vulnerabilities (previously unseen) in tokenized system call data.

1.5 Approach

The high level approach for testing the hypothesis is to collect data, train a machine learning model and test it against potentially malicious system call traces.

To allow for reproducible experiments, some of the training data and samples are collected from sources like VirusTotal academic access[2] and LID-DS[3].

The VirusTotal data repository consists of only malware binaries samples and metadata from static analysis and signature detection. To be able to use the dataset, malware execution is necessary to collect the system call traces.

1.6 Structure of this Thesis

- Chapter 2 comprises a literature review and overview of background concepts informing the research on anomalies, Linux system calls, machine learning and NLP.

[2]https://go.chronicle.security/virustotal-private-services
[3]https://www.exploids.de/lid-ds/

- Chapter 3 presents the research methodology and details on the experimental setup and data handling, preprocessing, enriching, scaling and validation.
- Chapter 4 evaluates and discusses the results gathered from the study and experiments.
- Chapter 5 provides a summary of the research and its findings, a discussion on the validity of the posed hypothesis, as well as future work.

Chapter 2

Literature Review

This chapter is divided into ten sections covering NLP and host based intrusion detection. The first section covers the high-level confidentiality integrity availability (CIA) model for discussing information security. The next section covers seminal research in the field of intrusion detection and systems for the detection thereof. Section 2.3 covers the aspects of untrusted code or binaries while Section 2.4 elaborates on the aspects of static and dynamic analysis of code. Section 2.5 elaborates on the critical aspects of malware analysis and prudent experimental design. Section 2.6 elaborates on the nature of Linux system and network calls. Linux malware, as well as the segmentation of malware by operating system, is covered in the next section. Section 2.8 gives in-depth definitions and insights into anomalies and their statistical representation. Finally the usage and application of machine learning in anomaly detection is discussed in the last section.

2.1 CIA Model

The CIA model or triad for secure information provides a firm foundation for any discussion around information security with the model divided into three parts or a triad of *confidentiality*, *integrity* and *availability* (Lundgren and Möller, 2019).

2.1.1 Confidentiality

Confidentiality is defined as the "property that information is not made available or disclosed to unauthorized individuals, entities, or processes" (Lundgren and Möller, 2019, p. 420). This definition can be simplified to state that only authorized users and processes should have access to data, and be able to modify or write data (Fruhlinger, 2020). The simpler definition is suitable for this research, as it deals with processes and not only human intervention. Untrusted or unauthorized processes' in a system should not gain access to other processes data or systems.

2.1.2 Integrity

Integrity is defined as the property of both completeness and accuracy (Lundgren and Möller, 2019, p.421). Fruhlinger (2020, p.1) further builds on this definition: "Data should be maintained in a correct state and nobody should be able to improperly modify it, either accidentally or maliciously".

2.1.3 Availability

The definition for availability is the "property of being accessible and usable upon demand by an authorized entity" (Lundgren and Möller, 2019, p.421). This can be simplified as authorized users or entities should have access to systems and data when required. Availability is simply explained through an example where a denial of service attack (DoS) or crypto malware[1] attack brings down a legitimate system and customers cannot transact, thus rendering the business unavailable (Fruhlinger, 2020).

2.1.4 Suitability of the model

The model is ideal as it is suitable for any information security perspective and is able to cater for both technical issues and soft or social issues of information securities with such examples as people, organizations, culture, ethics and policies (Lundgren and Möller, 2019).

It should be noted that the CIA model has its critics, but "theorists argue that CIA supplies goals for a secure system rather than serving as a definition supplying a demarcation between secure and insecure information (systems)" (Lundgren and Möller, 2019, p.421). The defining of goals for a secure system is ideal, as it aligns with the goals of this research. The research is not investigating a perfect model for describing information security, but rather applying the CIA model to describe the goals of keeping a system secure by detecting anomalies.

2.2 Intrusion Detection

Sarmah (2021, p.4) defines an IDS as the following: "a security system that monitors computer systems and network traffic and analyzes that traffic for possible hostile attacks

[1]https://www.f-secure.com/v-descs/articles/crypto-ransomware.shtml

originating from outside the organization and also for system misuse or attacks originating from inside the organization".

For the purpose of this research, we refer to the Denning (1986) model of an IDS. This model is preferred as it was designed with goals such as real-time intrusion detection and the ability to "detect a wide range of security violations ranging from attempted break-ins by outsiders to system penetrations and abuses by insiders" (Denning, 1986, p.1). The author further states that the model is independent of environment, application and vulnerability that is the subject of intrusion detection.

2.2.1 Intrusion detection system model components

Denning (1986) breaks down the IDS model into the following six components:

1. Subjects: Initiators of activity on a target system, normally the users.
2. Objects: Resources managed by the system such as files, commands, devices, and so on.
3. Audit records: Generated by the target system in response to actions performed or attempted by subjects on objects, for example user login, command execution and file access.
4. Profiles: Structures that characterize the behavior of subjects with respect to objects in terms of statistical metrics and models of observed activity. Profiles are automatically generated and initialized from templates.
5. Anomaly records: Generated when abnormal behavior is detected.
6. Activity rules: Actions taken when some condition is satisfied, which update profiles, detect abnormal behavior, relate anomalies to suspected intrusions, and produce reports.

2.2.2 Subjects and objects

Subjects and objects are defined together by Denning (1986), as they represent the senders and receivers of activity in the system. The subjects are initiators of events or actions within a target system and all activity in the model arises from commands initiated by subjects. It is important to note that the model's use of subjects, groups both legitimate

and illegitimate users of the system together and describes a typical subject as: "typically a terminal user, but might also be a process acting on behalf of users or groups of users, or might be the system itself" (Denning, 1986, p. 3). The author further states that users could be grouped into classes, with the purpose of managing control over access system objects, as well as the fact that these groups could overlap. Objects on the other hand, are on the receiving end of events and actions (Denning, 1986). Objects may include programs, files, messages, user or program created objects, etc. (Denning, 1986). Objects are a large group of receptors and could even include being subjects if they are recipients of actions. Once a subject is classified as being able to receive actions, the subject is viewed as an object within the model. For the model, objects are grouped into classes by their type and groupings can be more granular based on characteristics like database relations. For the most part, with the model proposed by Denning (1986), file or directory level grouping and classes will suffice.

2.2.3 Audit records

Audit records within the model constitute six tuples, representing actions performed by subjects on objects. For example an audit record of <Subject, Action, Object, Exception-Condition, Resource-Usage, Time-stamp> is described as follows (Denning, 1986, p.3):

1. Action: An event or operation that was performed by the subject with or on the object.
2. Exception condition: an exception condition, if any is raised by the system, not the exception returned to the subject.
3. Resource usage: the list of quantitative elements with each element being an amount of resource such as CPU time or records returned.
4. Timestamp: an obvious data point in an audit log, described as a unique timestamp indicating when the action occurred.

The audit record's specification of subjects and objects conceptually makes it similar to models such as the *access-matrix* model (Denning, 1986). The IDS differs from the access-matrix model through the substitution of the concept of action performed for action authorized.

A critical factor of the IDS model is that it deconstructs or decomposes activity into single actions of objects (Denning, 1986). This reflects the fact that many system operations

comprise multiple simple steps, such as a user copying a file. Denning further provides the following example, *COPY GAME.EXE TO <Library>GAME.EXE*, decomposed to multiple simpler instructions or steps:

```
(Smith, execute, <Library>COPY.EXE, 0, CPU=00002, 11058521678)
(Smith, read, <Smith>GAME.EXE, 0, RECORDS=0, 11058521679)
(Smith, write, <Library> GAME.EXE, write-viol, RECORDS=0,
11058521680)
```

The advantages of decomposing actions or instructions is relevant to this research, as most actions in a system comprise multiple follow up actions (Denning, 1986). Other than providing context of actions, Denning further identifies three advantages of decomposing. Firstly, objects are parts of the system needing protection, thus decomposition is consistent with the system's protection mechanisms, or more simply stated, specific steps or actions to protect and prevent future unauthorised actions.

The next advantage is that single-object audit records simplify the model and how it is applied. Any research is aided by simplification so that a model can be applied in real world scenarios. The third and final advantage identified by Denning (1986), is that existing systems at the time were based on single object actions.

Potential drawbacks are identified by Denning (1986) in the audit records. A potential strength and drawback of the auditing records is that the target system is responsible for auditing and transmitting the audit records. Though the target system can identify problems as they occur, a compromised system could be logging unreliable or false audit records. A compromised system could alter events, leading to altered audit records and malicious events go unnoticed on a system.

2.2.4 Profiles

The IDS model's profiles are described as the following by Denning (1986, p.5): An activity profile characterizes the behaviour of a subject, in respect to a object, providing a signature or example of normal activity for its perspective subjects or objects. Thus the model would identify a anomaly of abnormal observed behaviour using an statistical metric and model. Denning further describes a metric as x, a random variable, representing an accumulated quantitative measure over a period. Three types of metrics defined by Denning (1986) are as follows:

1. Event counter or the counter of audit records within the period that equals some condition or property.
2. Interval timer or the span of time between two events.
3. Resource measure or amount of resources consumed during a period, by an action.

The periods described by Denning (1986) are either a fixed time interval, such as a minute, or the time between two audit-related events such as a login and subsequent logout. Observations from audit records are used with a statistical model to determine whether a new observation is abnormal. The statistical models identified by Denning are:

1. Operational model
2. Mean and standard deviation model
3. Multivariate model
4. Markov process model
5. Time series model

Denning further provides a profile structure of ten components that can identify the statistical model and a random variable as a metric, as well as the audit events which the variable measures. The profile structure captures such properties as the variable name, resource usage patterns and others as referenced in Listing 2.1 (Denning, 1986, p.5).

The use of the profile allows the IDS to update a variable's distribution on an audit record that corresponds or matches a variable's pattern and check for an anomaly (Denning, 1986). Furthermore by adding profiles for classes and aggregating individual activity, the IDS can identify if behaviour of an object or user is consistent with other objects or users.

2.2.5 Anomaly records

Periodically or whenever an audit record is generated, an interactive development environment would update its activity profiles and check for anomalous behaviour (Denning, 1986). The detection of an anomaly or abnormal behaviour would then create an *anomaly record* in the Denning model, which has the format <Event, Time-stamp, Profile> and the three components, namely:

Listing 2.1: Denning model profile

```
<Variable-Name, Action-Pattern, Exception-Pattern,
Resource-Usage-Pattern, Period, Variable-Type, Threshold,
Subject-Pattern, Object-Pattern, Value>
    Example:
    Variable-Name: SessionOutput
    Action-Pattern: 'logout'
    Exception-Pattern: 0
    Resource-Usage-Pattern: 'SessionOutput=' # -> Amount
    Period:
    Variable-Type: ResourceByActivity
    Threshold: 4
    Subject-Pattern: 'Smith'
    Object-Pattern: *
    Value: record of ...
```

1. Event: is a record of the event leading to the abnormality. The type of record is broken into two categories: *audit* for the data in an abnormal audit record and *period* for the data accumulated over the current interval that was found to be abnormal.
2. Time-stamp: a unique time-stamp or interval stop time which allows tracking an anomaly back to an audit record.
3. Profile: the profile is simply the profile created from the activity of the abnormality as it was observed. The author states that this could simply be an ID or database relationship to the profile record.

2.2.6 Activity rules

An IDS finally provides activities and rules on which an information security team can base reactions to anomalies (Denning, 1986). As stated by Denning (1986, p. 14): Activity rules define actions taken on the creation of anomaly or audit records, or when a period of time ends. Denning, to prevent confusion with previous discussions of action, calls the action in a rule the body, which is illustrated in Listing 2.2. The IDS model's activity rules consist of two parts: conditions that satisfy a rule and cause actions to be triggered or *fired*, and an action that corresponds to a rule (Denning, 1986). The conditions match patterns in events and have four types Denning (1986):

1. Audit-record rule is triggered by a new audit record matching the patterns on an activity profile.

Listing 2.2: Denning model example rule

```
Condition:  Clock mod p = 0
Body:    Start = Clock - p;
         A = SELECT FROM Anomaly-Records WHERE
         Anomaly-Record.Time-stamp > Start;
         generate summary report of A;
END
```

2. Periodic-activity-update rule is a rule triggered by the end of an interval matching the period component of an activity profile, which in turn updates the profile and checks for an anomaly.

3. Anomaly-record rules are immediate triggers to notify a security officer, brought on by the generation of an anomaly record.

4. Periodic-anomaly-analysis rule is triggered by the end of an interval and generates summary reports of the anomalies during the current period.

Listing 2.2 shows an example of an periodic-anomaly-analysis rule as given by Denning (1986, p.15). The listing shows a time based rule which generates a report of all vulnerabilities within a time frame.

2.2.7 Application and limitations of the model

The components of the Denning model, form part of a rule-based pattern matching system which is applied to monitoring standard operations of systems. Once a deviation occurs within the standard operation, profiles can be used to apply rules or report anomalies to a security officer.

The limitations of the model are the lack of context of the target system's security measures as well as complex actions used to exploit a flaw in the system (Denning, 1986). The Denning model does propose that flaw-based detection, could have value in such a proposed system, but would not account for insider threats, complexity of implementation as well as deficiencies not suspected. Denning further proposes that for detecting an *intrusion* or anomaly, a security officer may be better equipped to locate and deal with a vulnerability or host of vulnerabilities or alternatively, an expert system, which would enhance the existing capabilities of a security team with anomaly detection and reporting, with the proposed profiles in the previous sections.

2.3 Untrusted Code

Untrusted code is the code run in an environment, trusted by the user or operating system, but which cannot be trusted due to its introduction from an external source (Hoopes, 2009). This leads to untrusted code running in a sandbox, for instance Javascript engines in browsers, to prevent the untrusted code from introducing vulnerabilities to the application or system in which it is executed.

Untrusted code is a reality in cloud computing environments, client side *Javascript*[2] and developer tool sets where the developer and system cannot fully trust the source of the code. Sabahi (2013) describes cloud computing as technologies for virtualization, with multiple virtual servers on the same physical infrastructure, and the self service technologies access over networks such as the Internet.

Untrusted code and its pitfalls are expressed by Janczuk (2015) as custom code executed by a user, but is untrusted by the platform operator. The author further states that extensibility and reuse of code and libraries is essential to the user or developer in providing features to clients, but the operator needs to limit the risk. The risk is mitigated by running untrusted code in an isolated environment, potentially virtually isolated, to limit the impact of this code on the other users and platform.

Multi-tenancy or cloud environments, as well as rapid development of features in a code base necessitate the isolation and separation of applications (Zerouali *et al.*, 2018). In modern computing architectures and especially in cloud environments, virtualization technology is frequently applied to isolate untrusted processes. The author further states that isolation of processes enables controlling the access of processes to sensitive resources. Virtualization and isolating resources have a downside as these incur performance cost with less resource sharing incurring more overhead and inter-process communication.

Virtualization of system resources allows for the isolation of applications, preventing compromise of another interconnected or completely separate code base (Sabahi, 2013). The author describes the various virtualization approaches of operating system based, application based and hypervisor based virtualization with mention of virtualization of network hardware. Additionally the author mentions the importance of authentication, authorization and network separation. A combination of these building blocks with a cloud application programming interface (API) allows customers to scale and isolate within multi-tenancy and untrusted environments (Sabahi, 2013).

[2]https://www.javascript.com/

Ratazzi *et al.* (2019) state that hypervisor and virtual machine based virtualization incurs a cost, in a large dynamically scaled environment, as well as raising various design and architecture challenges with the additional platform complexities. These platform complexities add overhead even though they are not directly related to improved security. Containers have enabled developers to write code more quickly and remove some of the complexities of hypervisor based virtualization (Zerouali *et al.*, 2018). As their adoption in enterprises and general usage increases, containers are accelerating the pace of innovation, but that pace can cause problems for security teams and securing applications when the tools present a quick and easy solution.

Sandboxing of untrusted code is only viable while a signature exists of known vulnerabilities and the system's known attack vectors are mitigated (Ratazzi *et al.*, 2019). Sandboxing is an active measure loaded before the execution of code, and thus a new vulnerability's mitigation would have to be added after the fact.

2.4 Static Analysis with Dynamic Analysis of Code

Research by Xu *et al.* (2013) uses static analysis as a starting point and the source of data for a model based on dynamic analysis. The authors combine static and dynamic analysis with machine learning to present a novel anomaly detection methodology that can in realtime provide a quantitative measure for software or system assurance. They define assurance as confirmation from evidence that the software is executing and behaving as intended, and that it has not been tampered with. The combination of static and dynamic analysis is to balance the overhead and noise generated by the static analysis of programs and attempt to gather more than binary predictions of executions (Xu *et al.*, 2013). Binary predictions are not sufficient according to the authors, as on their own, they pose minimal threat, but multiple anomalous events could relate to a DoS attack.

The focal point of the research by Xu *et al.* (2013) is on probabilistic machine learning based black-box techniques, but the observed methods were started from scratch, where the data from static analysis could jump-start training and predictions. The authors further propose the use of static analysis to jump-start a hidden Markov model, for the purpose of anomaly detection. They give an approach to strengthen a learning-based model, with control flow dependence information, derived from program static analysis. Strengthening the probabilistic dynamic learning model is achieved with call and control flow graph dependence information, which is derived from static analysis.

The authors proposed and tested the use of static analysis, and the artifacts produced by these methods, to train machine learning models around normal behaviour. Thus they

had to produce normal behaviours from trusted sources, and combined data from client and server-side applications. The authors further tested various lengths of traces and also analysed system calls and library call sequences to prove their hypothesis.

2.5 Malware Analysis

Rossow *et al.* (2012) describe malware analysis as the observations that researchers make in executing malicious code in various experiments, to study behaviours for the generation detection models and algorithms. The authors provide a rigorous methodology or guidelines for determining relevant malware execution studies. The guidelines or methodology is used to select studies or experiments which adhere to areas such as: safety, realism, transparency and correct datasets. They also address common issues in malware dynamic analysis and data collection, to strengthen research and conclusions.

Rossow *et al.* (2012) identified issues in malware research by discovering in research and analysis that well performing and efficient experiments in the lab perform poorly in real world evaluations. They also identify that prudent experimentation is difficult and prudent practices are needed towards solidifying the science of malware research. Characteristics are identified for research through experience and other relevant literature, not for the purpose of criticizing malware execution, but to provide guidelines for prudent research with these datasets and reproducing results. These characteristics are grouped into four categories: *correct datasets*, *transparency*, *realism* and *safety*.

2.5.1 Correct datasets

The first aspect or characteristic highlighted by Rossow *et al.* (2012) is checking whether legitimate samples used in malware research should be removed from the dataset. Legitimate software samples (or goodware samples, as defined by the authors) are necessary in experiments to measure false alarms, but are not desirable when measuring false negative rates in a experiment (Rossow *et al.*, 2012). The authors advocate that datasets and samples are gathered from sources that remove goodware samples from their datasets or compile sample subsets based on the malware family labels. This is based on the fact that public sample permissions can contain samples executed that had no malware component at all, which puts it in a third category of unknown or uncertain. This allows for study of anomalies where data can be predicted as malicious or not.

The following two concerns and characteristics are focused on the malware families. The first characteristic is that a dataset should be balanced as aggressively polymorphic malware families could dominate the sample uniqueness of the dataset, especially the message-digest algorithm 5 (MD5) hashes (Rossow *et al.*, 2012). The second characteristic is on the use or exclusion of distinct families of malware between training and evaluation datasets. Malware families in both datasets can assist in generic detection models, whereas experiments aiming to study unseen or newly discovered types of malware would need to separate the malware families in the datasets.

A crucial characteristic in evaluating a dataset for correctness is in the dynamic analysis privileges of the malware (Rossow *et al.*, 2012). Malware, especially the kind that has rootkit[3] functionality can affect OS data structures and can affect the kernel and higher up in the system. An additional threat not apparently clear in Denning (1986)'s research, is the affect of the malware or intrusion on the monitoring components (Rossow *et al.*, 2012). Thus consideration is needed in two aspects, firstly in the privileges the malware is allowed to run, and secondly at which privilege the monitoring system runs to be able to observe the gained privileges and affects of the malware (Rossow *et al.*, 2012). It is critical that the sensors are placed or located at a level where they cannot be modified or tampered with and the authors suggest running these at a level where they can monitor an emulator or virtual machine.

Another characteristic given by Rossow *et al.* (2012) is to discuss and if necessary mitigate analysis artifacts and biases. The authors identify that software configuration of an analysis environment or artifacts like strings containing usernames or operating system serial keys, can be recorded as part of the manifestation of a particular execution of the malware. This is especially crucial where research derives models for detection, and requires discussing the particulars of the artifacts the model leverages, for the replication of such research. The authors further highlight the potential biases around malware behaviour as containment policies can affect artifacts. A real life infected system and analysis environment can produce vastly different artifacts, adversely affecting the research or detection model.

Finally Rossow *et al.* (2012) identify a characteristic and caution in combining malware trace activity into normal or benign background activity. As mentioned in the previous paragraph on biases, behaviour of malware in victim machines in the wild could differ substantially from dynamic analysis[4] environments (Rossow *et al.*, 2012). A further aspect

[3] a term commonly used to describe malware—such as Trojans, worms, and viruses—that actively conceals its existence and actions from users and other system processes (McAfee, 2006)

[4] Dynamically executing a process/malware/program

is identified in that the performance aspects of a system can affect the activity observed and some malware families employ mitigations based on system resource specifications (Rossow *et al.*, 2012; Vidas and Christin, 2014). A mismatch between background activity and the performance characteristics of real world scenarios, may lead to evaluation results that appear to be excellent, but fair poorly in real-world settings (Rossow *et al.*, 2012). The authors propose that researchers discuss these issues explicitly when combining real world traces with harmless background activity.

2.5.2 Transparency

Transparency is correctly identified by Rossow *et al.* (2012) as a key characteristic in malware dynamic analysis. Transparency not only gives insights into the methods of the researcher, but allows for procuring the same samples; as well as producing similar results. Despite mentioning consistency in naming malware families, Rossow *et al.* (2012) identifies that the names of employed malware samples are the first characteristic in transparency. The labeling of malware allows readers to correctly identify malware and which methodology works with the malware family. Furthermore a large sample of malware does not guarantee family diversity, as binary-level polymorphism[5] could be present in the dataset. It is also suggested that where the information would be too verbose and increase page count, authors can use links to websites and reference accordingly.

A second critical aspect of transparency highlighted by Rossow *et al.* (2012) is the listing of malware usage by time or timestamp. The authors further suggest adding detail or a summary external to the paper that describes the dataset in detail for researchers to repeat exactly. Time is an essential aspect, especially with the ephemeral nature of some malware. The observed behaviour within the correct context (including time) is essential in observing its interactions with *botnets*, as take-downs or changing versions affect the observable behaviour of the malware.

As with any research, the researcher should be transparent on the sample selection, in this case that of the malware samples (Rossow *et al.*, 2012). The authors state that researchers often only study a subset of the malware available and that for example only a random sampling of statistically valid samples may prove necessary. A researcher could also decide to focus on more recent analysis and results, instead of historic data (Rossow *et al.*, 2012). The authors insist that researchers should adequately describe the selection criteria for the malware samples as well as the potential bias introduced by the criteria.

[5]techniques by which malware change their identity (Vidas and Christin, 2014)

Random selection of samples could also produce imbalances, which is addressed at a later stage in the paper through guideline A.2 which states that balancing training datasets should be done in terms of malware families and not individual specimens.

The authors, following two aspects of transparency, recommend that researchers describe the system as well as the network connectivity of the analysis environment. As malware and assumptions around a standard operating system change with time, it is crucial to describe the system and software versions (Rossow *et al.*, 2012). Furthermore the behaviour of the malware could change based on the type of system detected and its configuration. This is especially relevant as different malware families assign different roles of activity dependent on the connectivity of the system. Botnets and malware systems can have the functionality to live off the land and in the example of the Waldec botnet, a system behind a network address translation (NAT) or direct Internet connection both have different roles (Rossow *et al.*, 2012).

Analyzing the reasons for both false positives and false negatives is a critical characteristic to address in transparency (Rossow *et al.*, 2012). As the authors state false classification rates alone do not provide clarification of a system's performance. The authors insist that researchers need to describe the conditions leading to false classification rates after exploring and analyzing the cause and effect of observed errors. Finally not only negative results, but also analysis is needed on the composition and diversity of true positive classifications. The authors describe a classic problem of positive rates as seen with an over trained model that detects a single family of malware or focuses on an environment artifact. A model's usefulness could be replaced by searching for string artifacts if it focuses on a single malware family. Instead it is necessary to evaluate the diversity of the data or samples in correctly identified detections, to assess the general classification accuracy of a model or method.

2.5.3 Realism

Rossow *et al.* (2012) identified a number of characteristics related to the realism of a study, the dangers and moral issues in obtaining realistic datasets. Firstly the authors identify evaluating malware families that are relevant to the study by prevalence or time, instead of using older and sinkholed samples.

They further define performing realistic or real world evaluations as: a scenario or data from a scenario incorporating multiple affected host computers in active use, other than those of the authors. The authors do not make suggestions or extensive recommendations

around real-world evaluations, but do emphasise their importance in evaluating the divide between an academic study and real world observations and environments.

The following two characteristics, caution in generalization and appropriate malware stimuli speak to the inputs needed to create realistic behaviours and traces (Rossow *et al.*, 2012). Generalizing on aspects like a single operating system such as Windows XP, may fail a study as malware families could exclusively run on a platform or disregard a specific operating system version. As with this research, the authors are aware of the nature of malware written for specific architectures, operating systems or software versions, and thus careful analysis is needed towards realism. With regard to appropriate stimuli Rossow *et al.* (2012) explain that malware can expose or activate further behaviour, based on longer execution time or requiring realistic user interactions. It is therefore pertinent that researchers describe stimuli and execution time, which may vary based on the study i.e. whether initialization or a long running process or DoS attack.

The final and a contentious point Rossow *et al.* (2012) make about realism, is related to Internet access for malware. The authors defer the moral, ethical and legal standpoint of an Internet connected malware analysis as increasingly malware needs to connect to command-and-control servers (c&c) servers first, before exposing and observing malicious behavior. To mitigate the risk of Internet exposure or harm, a researcher can have a simulated Internet environment where appropriate, but must describe limitations resulting from the research (Rossow *et al.*, 2012).

2.5.4 Safety

To achieve a realistic study, as compared with a real-world analysis, Rossow *et al.* (2012) identified many potential pitfalls, cautions and dangers with realistic research. They identify the trade-off of safety and containing malware, with the impact on the fatefulness of the experiments. The potential harm and spread of researching malware must be contained in realistic experiments, and employ basic mitigations such as containment, redirecting spam and suppressing DoS attacks.

According to Rossow *et al.* (2012) in the purpose of research, authors should discuss their containment polices. Furthermore authors need to monitor, identify and disclose security breaches in their containment. Unintended malware breakout from containment can further bolster the research and understanding of the severity and mechanisms of the malware.

2.6 Linux System and Network Calls

Bagherzadeh *et al.* (2018) have a unique insight into Linux System calls, based on a study of 25 years of system calls in the kernel. Logging and measuring Linux system calls is a unique challenge as in 25 years 18 million lines of code have been added to the kernel, with an average of 25 lines of code that were added to the system calls per day. From another study the authors noted that 80% of breaking changes in the kernel was refactoring, so from a behavior perspective the API changes often, but the underlying syscall according to the documentation remains the same to the developer or application.

Further study of the Linux kernel API changes, allows for tracking new functionality, but also inferring dangerous behavior from security fixes and deprecations. A great wealth of knowledge was gained from Bagherzadeh *et al.* (2018) classifying the syscalls with the largest commit counts, which include the following commits: *ptrace*, *signal*, *ioctl*, *futex*, *ipc*, *mmap*, *perf_ event_ open*, *readdir* and *splice*. The aforementioned syscalls had between 23% and 45% of commits dedicated to bugfixes, with 2009 alone accounting for 45 commits related to security fixes from reports, in syscalls. The authors also noted that in 25 years an additional three system calls were added including such calls as *seccomp* and *bpf*. A final take away from the work by Bagherzadeh *et al.* (2018) is that not all Linux syscalls are activated for every architecture. Different architectures complicate implementing syscalls and causes this research to be limited and scoped to x86_amd64[6].

Within the context of Zero Trust and multi-tenant environments, lies the research by Rajagopalan *et al.* (2005) that encompasses *Authenticated System Calls*. As stated by the authors, system call monitoring is a technique for observing and controlling suspected compromised applications at runtime by checking that each system call conforms to a policy outlining the program's expected or normal behaviour. The research builds on this concept with monitoring based on authenticated system calls. An authenticated system call is defined as a system call that is "augmented with extra arguments that specify the policy for that call and a cryptographic message authentication code (MAC) that guarantees the integrity of the policy and the system call argument" (Rajagopalan *et al.*, 2005, p.1). This allows the kernel to verify each system call and the modified system calls are generated by an installer that verifies, profiles and rewrites the binary after a static analysis is done. The research by Rajagopalan *et al.* (2005) adds a MAC and profile to the application as well as the crypytographic key in the kernel, which was used to

[6]Google's gvisor implements 211 of the 319 x86_amd64 system calls and many virtualization technologies and cloud platforms are based on x86 Virtualization. (Ding *et al.*, 2015, p.192)

generate the MAC. On a system call the kernel recomputes the MAC from the key and allows or blocks the syscall based on its MAC. This prevents a malicious program or user from adding a new system call (in the application) or man-in-the-middle attacks on an application's calls to the kernel.

Within the Linux kernel the system call *secure computing mode (seccomp)* implements policies for executing certain system calls, and terminates system calls outside of the policy (Anderson, 2017). Originally *seccomp* only allowed the system calls read, write, sigreturn and exit, with any other syscall triggering process termination. The original version of *seccomp* had the benefits of clarity and permitting processes to operate as filters, but was deemed too restrictive and prevented most programs from performing meaningful work (Anderson, 2017). Chrome used to operate in a pure *seccomp* mode, but this required a large codebase in assembly language to forward system calls from a sandbox process to a trusted process that executed system calls for the sandbox. Modern *seccomp* allows each program to have its own policy and white-list of system calls, but is complicated and not effective on its own to build a sandbox for executing arbitrary code. In addition to *seccomp* a separate inter-process communication (IPC) namespace is needed for networking and system calls to be passed by a proxy to the process in the separate IPC namespace.

2.7 Linux Malware and Exploiting x86_64 Systems

Microsoft Windows has an approximate 83% market share for desktop computers, thus malware writers primarily target the Windows operating system historically (Cozzi *et al.*, 2018). Malware targeting Linux has grown in number due to the proliferation of Linux in the cloud, embedded and mobile computing and the Internet of Things. The vast number of CPU architectures and available hardware makes Unix-like operating systems attractive to developers, but security can be a secondary focus in a low-cost and limited hardware environment (Cozzi *et al.*, 2018).

2.8 Anomaly Detection

Mutz *et al.* (2006) describe how anomaly-based IDS systems function, by analyzing and building models of normal behaviour. Normal behaviour models are created by processing and analyzing events from network or host based auditing systems in expected normal behaviour. The authors further make the distinction that network based detection systems monitor and analyze network traffic's payloads and headers, while host based systems

focus on user or program activity and behaviour, monitored on the application or operating system level. Further distinction is made between individual and sequenced system calls, where sequenced system calls are subject to mimicry and could categorize denial of service behavior (Mutz *et al.*, 2006). These types of systems are otherwise described as IDS, where an anomaly within the program and network behavior constitutes a malicious intrusion into the system.

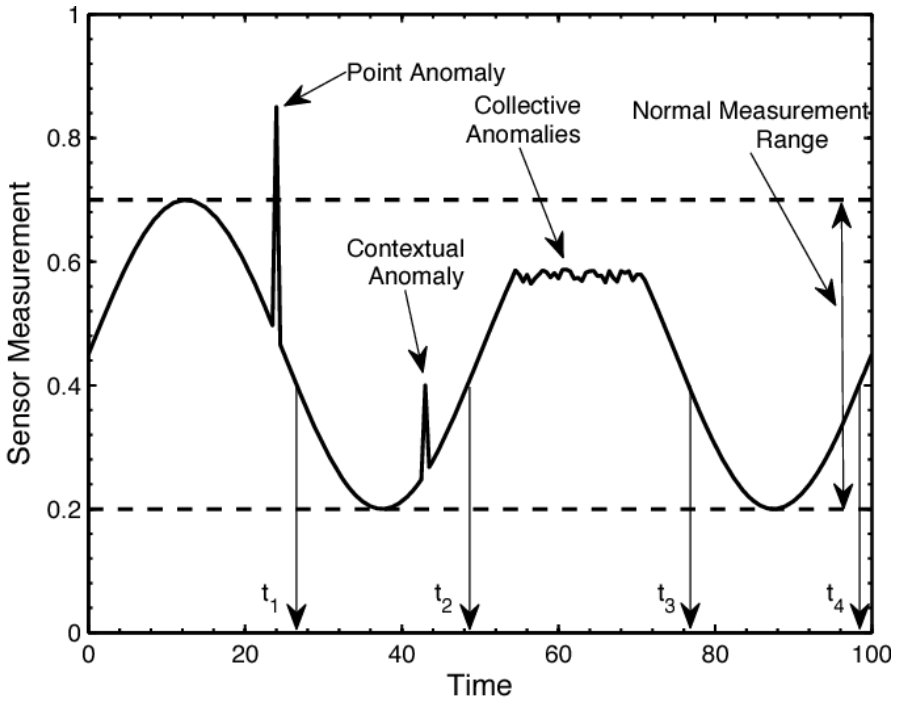

Figure 2.1: Anomaly detection types taken from O'Reilly *et al.* (2014)

2.8.1 Point anomaly detection

Point anomaly detection in host based IDS is critical, as signature based systems are dependent on having signatures before an attack occurs (Grimmer *et al.*, 2018). Goldstein and Uchida (2016) describe point anomaly detection as the process of detecting a single anomalous instances or occurrence within a larger dataset. Point anomaly is illustrated in Figure 2.1, before time point t_1. Point anomaly detection algorithms do not need signatures beforehand and enable novelty and outlier detection (Grimmer *et al.*, 2018). Extracted features can be plotted in two dimensions and anomalous points identified with classifiers, such as Axis Aligned Bounding Box Classifier, K-centers and Nearest Neighbor.

2.8.2 Contextual anomaly detection

Contextual anomalies are described as a data point which in isolation indicates a normal observation, but within the context is observed as an anomaly (Goldstein and Uchida, 2016). O'Reilly *et al.* (2014) illustrate a contextual anomaly in Figure 2.1, the context that if it occurred at any of the periods t_1 to t_4, would not have been an anomaly. Context anomaly detection can be applied to network calls and data, based off a baseline or context (Garje *et al.*, 2014). The potential for detecting and responding to Zero-Day attacks makes contextual anomaly detection a powerful tool for intrusion detection (Aleroud and Karabatis, 2013). In a multi-tenant system we can track the context of a user in the system, or events such as a DoS (Garje *et al.*, 2014). Context-aware security utilizes external data, such as time of the day or user information, to improve its capability of detecting a security breach (Sánchez and Aracil, 2017).

Contextual anomaly detection can use both static and dynamic analysis of network traffic to configure a baseline and detect anomalies (Aleroud and Karabatis, 2013). One risk of contextual anomalies in unsupervised anomaly detection techniques is the large numbers of false positives and alerts produced.

2.8.3 Collective anomaly detection

Chandola *et al.* (2009) defines *collective anomalies* as a collection of related data points, that with respect to the entire dataset, are observed as anomalous. Each event within a dataset is not classified as an anomaly, but collectively their occurrence in the dataset can be grouped as an anomaly. Figure 2.1 illustrates an example of collective anomalies, in which each data point on its own is not anomalous, but collectively is anomalous to the entire dataset. An example by Chandola *et al.* (2009) provides a sequence of events, familiar in a web based attack or intrusion: where an attack like a ssh bruteforce by a remote machine over the web, would be followed by data ex-filtration to the same or a different remote machine. To differentiate a collective anomaly from a point anomaly, the author states that point anomalies can be found in any datasets, but collective anomalies only occur datasets with related events or points. Furthermore a collective anomaly can be a contextual anomaly when enriched with context information.

2.8.4 Application of anomaly detection

Chandola *et al.* (2009) identify many anomaly detection applications and describe them through characteristics or aspects, including the notion of anomaly, nature of the data, challenges associated with detecting anomalies and existing anomaly detection techniques.

Though the authors mention many different applications, such as fraud detection and sensor networks, for this research we focus on the application in intrusion detection.

Chandola *et al.* (2009) identify that an intrusion differs from normal behaviour observed in a system, but huge volumes of data are a challenge. Computational efficiency is key for anomaly detection to handle large inputs and frequent events. The authors identify the need for online analysis to process streaming sources of data. The large volume of streamed events are not just computationally expensive, but are susceptible to a high false alarm rate and overwhelming and confusing analysts. The authors propose semi-supervised or unsupervised anomaly detection, as labeled normal datasets are frequently available, as opposed to unlabeled intrusions. The authors rely on the definition by Denning (1986) to further classify intrusion detection into either network or host based.

Host based intrusion detection systems

Host based IDS or *system call intrusion detection systems* detect events via system call traces (Chandola *et al.*, 2009). Of interest to this research, the authors identify these intrusions by the subsequences of anomalies (collective) within the traces.

This forms the basis of the research question that a collection of anomalies could be traced back to a malicious program, which in between normal traces would be identical to normal behaviour on its own. The execution of a program and malicious program would share a common language, but as identified by the authors, the grouped occurrence of these words would be out of place. Tracing anomalies with *system call (syscall)* traces is characterised by an alphabet consisting of individual or system calls and sequential data points as illustrated in Figure 2.2.

open,	read,	mmap,	mmap,	open,	read,	mmap	...
open,	mmap,	mmap,	read,	open,	close	...	
open,	close,	open,	close,	open,	mmap,	close	...

Figure 2.2: Example system call traces taken from Chandola *et al.* (2009)

Tracing system calls is complicated by varying execution sequences, varying length of sequences and can be profiled at various levels, such as a user access level. Due to the complicated sequences such traces could have, a host based IDS requires sequential data processing. Chandola *et al.* (2009) identify that within this domain point anomaly

detection is not applicable as the techniques are required to compute the similarity of sequences or model the sequence data.

Network based intrusion detection systems

Though the focus of this research is on host based intrusion detection, network traces can be a source of intrusion detection and trace back to specific syscall traces. A Network Intrusion Detection System uses network traces as a dataset and intrusions occur or are observed as point anomalies and through certain techniques the data can be modeled in a sequential fashion to detect the collective anomalies in the subsequences of the data (Chandola *et al.*, 2009).

The anomalies are primarily caused by attackers external to the organization and network, to gain access, disrupt or compromise hosts. The data sources for these intrusion detection systems vary and can include packet level traces and CISCO netflow data (Chandola *et al.*, 2009). The authors identify that detecting intrusions on a network level is challenging, as the attackers constantly adapt their attacks to existing mitigations and data is highly dimensional. Unsupervised methods are used to adapt to changing attacker behaviour, but high false alarms can be prevalent. Some network attacks can mimic normal behaviour within the dataset and thus be collective anomalies as mentioned before.

2.9 Natural language processing for anomaly detection

Natural language processing is a field combining the fields of artificial intelligence and the field of natural language (Jurafsky and Martin, 2009). Other descriptions of the field give further explanation of its capabilities, such as speech recognition and synthesis, computational linguistics and human language technology. Jurafsky and Martin (2009) describe the goal of the field to perform various human language tasks such as: improved human to human communication, improved human to machine communication or even processing and transforming speech and text. The algorithms and inference capabilities of NLP are essential as human language contains complexities such as ambiguity, different dialects, phonetic, phonology and incorrect spelling (Jurafsky and Martin, 2009).

As stated by Rao and McMahan (2019) all NLP methods start with a text dataset otherwise known as a corpus. The corpus contains raw data or text as well as metadata relating to or associated with the text (Rao and McMahan, 2019).

2.9.1 Pre-processing

Before the raw text can be used, it needs to be pre-processed. Pre-processing is the term for the process of cleaning, filtering, refining and turning unstructured text based data into valuable data. (Lourdusamy and Abraham, 2018). The author describes the rationale of pre-proccessing as textual data processed with various refinement methods to produce cleaned text, from which knowledge and context can be extracted. Pre-processing encompasses many different techniques as described by Lourdusamy and Abraham (2018):

1. Tokenization
2. Part of speech tagging
3. Grammatical parsing and chunking
4. Stemming
5. Document indexing
6. Stop word filtering
7. Word sense disambiguation
8. Lemmatization
9. Text summarization
10. tf-idf

2.9.2 Tokenization and vectorizing text

An essential part of NLP is tokenization or breaking up a stream of characters into words, punctuation marks, numbers and other discrete items (Kibble, 2013). Tokenization is explained by Rai and Borah (2021) as the process of splitting or breaking up words and sentences into the smallest or simplest morpheme defined as a token. A morpheme would be the smallest possible word, and could not be broken down any further (Rai and Borah, 2021). A simplified explanation of tokenization is put forward by Manning *et al.* (2008): a document and character sequence split or broken up into tokens (tokenization), with the possibility of discarding punctuation or characters.

Once the sentence is broken into tokens and text is cleaned, the text needs to be presented in a numerical format for machine learning and statistical algorithms to process. As

explained by Bengfort *et al.* (2018, p.55): machine learning (ML) algorithms operate on numeric features, with inputs of two dimensional arrays where columns provide features for learning and rows the examples for the model. The authors state that sentences to entire documents can be represented as instances for machine learning algorithms to operate on, but are turned into vectors. These vectors are of uniform length in the representation, and each property within the vector representation is a feature of the dataset. Bengfort *et al.* (2018) further state that the content and meta attributes of a document, attributes such as document length, all form part of the features of a document. All the features of a document represent a multidimensional feature space, which allows the application of machine learning algorithms to natural language documents.

A simple example of a tokenizer is the WhitespaceTokenizer as seen in Figure 2.3. This simple tokenizer creates a token after each whitespace within the sentence, but as seen in this example of an application or system log, this tokenizer would not be optimal. Another example in Figure 2.4 uses a statistical model trained on news text, to tokenize text, with fairly good results on application logs (Perkins, 2018). This is another example of application or system logs being based on human language.

Figure 2.3: Example of whitespace tokenization (Perkins, 2018)

Some of the models or methods commonly used for text to vector representation include: Frequency, One-Hot Encoding, Tf-idf and Distributed Representations (Bengfort *et al.*, 2018). The methods are further described in Figure 2.5. For the purpose of this research, frequency based methods such as Bag of Words and TF-IDF are of interest.

Figure 2.4: Tokenization using a Treebank Tokenizer tokenization (Perkins, 2018)

Vectorization Method	Function	Good For	Considerations
Frequency	Counts term frequencies	Bayesian models	Most frequent words not always most informative
One-Hot Encoding	Binarizes term occurrence (0, 1)	Neural networks	All words equidistant, so normalization extra important
TF–IDF	Normalizes term frequencies across documents	General purpose	Moderately frequent terms may not be representative of document topics
Distributed Representations	Context-based, continuous term similarity encoding	Modeling more complex relationships	Performance intensive; difficult to scale without additional tools (e.g., Tensorflow)

Figure 2.5: Common vectorization methods for NLP (Perkins, 2018)

Bag of words

The simplest method of vectorizing or encoding is the BOW model. The BOW model represents the documents of text as a vector of equal length to the vocabulary of the corpus (Bengfort *et al.*, 2018). This model is simple, but is the starting point of other more complex models such as tf-idf, as identified by the author. The model further decreases computational load by the vector token positions being arranged or sorted alphabetically (Bengfort *et al.*, 2018) as seen in Figure 2.6, where the Y axis labels are alphabetically ordered. This does not contain the original sequence of the input, but rather alphabetically ordered counts of word occurrences. An example bag-of-words representation for systemcalls or strace log, is presented in Figure 2.6. The figure shows that Document inputs, show on the X axis, and the corresponding word labels (in alphabetical order) and counts on the Y axis. Each document has the same labels, but a incremental count will exist if the word is found within a document.

Out[151]:

	_exit	_llseek	af_unix	ar	b384	bad	bash	bash_history	bashn	bfffff89n	...	v1	var	vuln	wait4	whoaminexitnexitnftp	write	ws_col	ws_row	ws
Doc0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
Doc1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
Doc2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
Doc3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
Doc4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
Doc377	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
Doc378	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
Doc379	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
Doc380	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
Doc381	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	

382 rows × 193 columns

Figure 2.6: Bag of words representation of strace log, with system calls on X axis and documents as integers in Y Axis.

Bag of system calls

As mentioned in the previous section, BOW models are often combined to form more complex models. A lot of research and applications of a *Bag-Of-Systems* calls method were identified in the research by Kang *et al.* (2005, p.1). The authors identified that modelling anomalous behaviour methods were based on either user or process behaviour. The authors further identified that system call data is the most common type of dataset used to study the behaviour of processes. The size of the data collected during a system call trace is identified as a stumbling block for storing data and training models and thus a better representation is needed.

Kang *et al.* (2005, p.1) identified that methods and detection algorithms such as *STIDE* use observations of "fixed-length, contiguous subsequences of system calls". Using these sequences, anomaly detection of subsequences of traces, are matched to normal traces in a dataset or database. The authors' motivation for a new method of representation and classification is that a dataset or database would grow exponentially with the number of traces, which could limit the size of the dataset and adoption of these methods. A theoretical calculation of such a database size for 200 traces with 6 subsequences is given as follows: $200^6 = 64 \times 10^{12}$ (Kang *et al.*, 2005).

A *bag of system calls (BoSC)* alternative representation of the data is proposed by Kang *et al.* (2005). The authors note that intrusion detection of system calls, is a classification problem, thus using a bag of system calls approach allows modelling the system call traces based on frequency. The approach is derived from the "bag of words representation that has been demonstrated to be effective in text classification problems" (Kang *et al.*, 2005,

p.1). A BoSC is a method based on integer frequency of a term and can be represented as an ordered list such as: "$X_i = \langle c_1, c_2, c_3, ..., c_m \rangle$". In this mathematical representation of BoSC, where $m = |\Sigma|$ and c_j is the number of distinct occurrences of s_j a system call. In this representation an input sequence is converted into BoSC where the input sequence is lost and the frequency of system calls are preserved.

The one potential drawback of the proposed classification method and representation is that the ordering metadata of system calls is not preserved, as opposed to the frequency of each system call which is preserved for each input sequence (Kang *et al.*, 2005). The authors note the above drawback, but using well known datasets, the research proved that frequency of system calls or the bag of system calls is sufficient to differentiate between abnormal and normal sequences or traces.

A further application of the Bag of System calls method was proposed by Abed *et al.* (2015). The proposed use of the model was applied with a real-time IDS monitoring a docker container, with strace, a system call parser and sliding window of size of ten. Figure 2.7 illustrates the system call processing with the sliding window. The sliding window and frequency is transformed into a bag of system calls, or BOSC, which is stored in a normal-behaviour database as seen in Figure 2.8.

Syscall	Index	Sliding window	BoSC
pwrite	6	[futex, futex, sendto, futex, sendto, pwrite]	[2,0,3,0,0,0,**1**,0,...,0]
sendto	0	[futex, sendto, futex, sendto, pwrite, sendto]	[**3**,0,**2**,0,0,0,1,0,...,0]
futex	2	[sendto, futex, sendto, pwrite, sendto, futex]	[3,0,**2**,0,0,0,1,0,...,0]
sendto	0	[futex, sendto, pwrite, sendto, futex, sendto]	[**3**,0,2,0,0,0,1,0,...,0]

Figure 2.7: System call parsing with sliding window and BoSC (Abed *et al.*, 2015)

BoSC	Frequency
0,1,0,2,0,0,0,0,1,0,3,0,1,0,0,0,1,0,0,1	15
0,1,0,1,0,0,1,0,1,0,3,0,0,0,0,0,1,1,0,1	8
0,1,0,2,0,0,5,0,0,0,0,0,0,0,0,0,1,0,0,1	2
0,1,0,2,0,2,0,0,1,0,2,0,0,0,0,0,1,0,0,1	1

Figure 2.8: Normal behaviour database BoSC representation (Abed *et al.*, 2015)

TF-IDF - Term frequency inverse document frequency

The BOW model is limited, in the sense that it describes a single document and fails to account for the corpus context (Bengfort *et al.*, 2018). Term frequency inverse document

frequency (tf-idf), as described by the author, allows the representation of "relative frequency or rareness of tokens in the document against their frequency in other documents" (Bengfort *et al.*, 2018, p.62). The authors further describe the computation of tf-idf as occurring on a "per-term basis, such that the relevance of a token to a document is measured by the scaled frequency of the appearance of the term in the document, normalized by the inverse of the scaled frequency of the term in the entire corpus.". Figure 2.9 shows that the term *libc* in the X axis, and we refer to 8 on the Y index, which we shall refer to as Document 8. The systemcall *libc* in Document 8 has been assigned a high value for its uniqueness within the corpus and frequency of of its own document. A term such as 'null' in document 0 in Figure 2.9, is assigned the value 0.0 as it is not present in the document.

Out[119]:

	isig	kayacik	kill	ld	lets	lib	libc	libnsl	libnss_files	libnss_nis	...	nscd_socket	nsswitch	null	o_append	o_rdonly	o_rdwr
0	0.000000	0.0	0.0	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.0	...	0.0	0.0	0.000000	0.0	0.000000	0.0
1	0.000000	0.0	0.0	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.0	...	0.0	0.0	0.000000	0.0	0.000000	0.0
2	0.000000	0.0	0.0	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.0	...	0.0	0.0	0.368774	0.0	0.000000	0.0
3	0.000000	0.0	0.0	0.347954	0.0	0.000000	0.000000	0.0	0.0	0.0	...	0.0	0.0	0.000000	0.0	0.281897	0.0
4	0.000000	0.0	0.0	0.481318	0.0	0.000000	0.000000	0.0	0.0	0.0	...	0.0	0.0	0.000000	0.0	0.389943	0.0
5	0.000000	0.0	0.0	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.0	...	0.0	0.0	0.000000	0.0	0.000000	0.0
6	0.000000	0.0	0.0	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.0	...	0.0	0.0	0.488318	0.0	0.000000	0.0
7	0.000000	0.0	0.0	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.0	...	0.0	0.0	0.000000	0.0	0.000000	0.0
8	0.000000	0.0	0.0	0.000000	0.0	0.532951	0.612473	0.0	0.0	0.0	...	0.0	0.0	0.000000	0.0	0.414828	0.0
9	0.000000	0.0	0.0	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.0	...	0.0	0.0	0.000000	0.0	0.000000	0.0
10	0.000000	0.0	0.0	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.0	...	0.0	0.0	0.000000	0.0	0.000000	0.0
11	0.000000	0.0	0.0	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.0	...	0.0	0.0	0.402210	0.0	0.000000	0.0
12	0.000000	0.0	0.0	0.000000	0.0	0.000000	0.000000	0.0	0.0	0.0	...	0.0	0.0	0.000000	0.0	0.000000	0.0

Figure 2.9: A tf-idf representation of system calls, with system calls on X axis and documents as integers in Y Axis.

The difference between BOW and tf-idf is the lack of context within all the documents. A BOW model only counts the occurrence of a word within a single document, tf-idf expands this model by counting the occurrence of the word within the document and inversely its occurrence of the word within all of the documents. As described by Bengfort *et al.* (2018, p.125) "text normalization techniques presented in previous chapters such as stop-words removal and lemmatization" could create the exact same BOW representation within 2 documents, with different meaning. The latter model, tf-idf, is able to describe, by comparing a frequency of terms, as compared to its rareness within the entire set of documents. This is very advantageous within the context of anomaly detection as infection or an exploit could happen over multiple system calls within the document.

The tf-idf representation or mapping of text to a vector combines the BOW method's count of token frequency with the context of that occurrence within the entire set of

documents or dataset. This is ideal when representing text based system call logs or traces for analysis with machine learning tools.

2.10 Machine Learning and Anomaly Detection

Anomaly detection with statistical models has been used historically, through a statistical inference testing to classify the data as belonging to the model (Nassif *et al.*, 2021). ML is increasingly being used for anomaly detection as it has the characteristic that allows automating knowledge acquisition from data, through presenting examples to models and classifying these samples. The authors identify that anomaly detection is an important issue of study and that it is used in fields such as cyber intrusion detection, medicine and, as relevant to this research, textual anomaly detection. They also classify anomaly detection, including machine learning, into three broad categories according to the function used to build or train the model: supervised anomaly detection, semi-supervised anomaly detection and unsupervised anomaly detection. The focus of the paper presented by the authors was to review literature and do a systematic literature review analyzing and reviewing ML models used in anomaly detection. The literature review allows researchers to see the use of ML models in anomaly detection, as well as the most prevalent or effective models to apply in the research.

2.10.1 Supervised anomaly detection

Supervised anomaly detection comprises both normal and abnormal datasets containing labels for the instances of events. The approach for the model is to train and build a predictive model for normal and anomalous classes for comparison (Nassif *et al.*, 2021). Supervised models require the researcher or author to label the data accurately and with representative labels, which can be tedious and difficult. A dataset with existing labels could be used, but the authors identify that the training dataset could contain far fewer anomalies than a real world dataset could. A supervised model could represent good lab results, but lack in real world inference and applicability. From the literature review, 18% of papers selected, applied supervised anomaly detection, making supervised models the second most used model in this research (Nassif *et al.*, 2021).

2.10.2 Semi-supervised anomaly detection

Semi-supervised machine learning models use training where ordinary cases are presented to the model. Thus the model infers and classifies anything anomalous that cannot be

classified within the ordinary. Nassif *et al.* (2021) state that these models are more preferred than supervised, as the anomalous events do not require labels and data is presumed to be labeled only for ordinary cases. Though these methods and models are preferred, only 5% of articles cited by the authors in their literature review, contained semi-supervised methods.

2.10.3 Unsupervised anomaly detection

Nassif *et al.* (2021) describe how unsupervised anomaly methods do not require training datasets and that it is implied that normal data points outnumber anomalies in datasets. The authors further suggest that many semi-supervised models or techniques can be adapted to unsupervised, by training with the unlabeled datasets. The assumption made by the authors is that few anomalies exist in the datasets and that the model will robustly train with these anomalies.

According to the literature review by Nassif *et al.* (2021, p.34), 27% of reviewed and selected papers were based on the application of unsupervised anomaly detection, as the sole model or method. This was the most prevalent detection model for anomaly detection, and papers studying this method appeared from the year 2002.

2.10.4 Models

As stated before in discussing unsupervised machine learning, for the purpose of this research we wish to use unsupervised learning models. Generally machine learning methods fall under three main categories: clustering, classification and hybrid methods or techniques (Liu *et al.*, 2020). For this research two models were identified for anomaly detection, both for classification through anomaly detection.

Random Forest provides the supervised learning classification, while Isolation Forest is a unsupervised algorithm. Both are commonly used in detecting anomalies in data and both models are based on decision trees (Liu *et al.*, 2020).

The common problem with decision trees is overfitting, where a low bias and high variance is implied, or rather does not generalize the data well (Ying, 2019). As described by the author the model memorizes all training data and struggles to infer on the test data, memorizing some of the noise of the training set and does not learn to infer from unseen data. This is primarily an issue with supervised learning and will be considered during the research.

Random Forest

Anton *et al.* (2019, p.3) describes a random forest as follows: "Random Forests are collections of Decision Trees, binary classifiers consisting of one root node, several internal split nodes and leaf nodes that are used to classify events". A classification of data occurs on a majority vote from all the trees. Resende and Drummond (2018, p.2) further highlight the Random Forest's "resilience to deal with imbalanced datasets; embedded feature selection method and intrinsic metrics to rank features by importance; and able to deal natively with categorical and continuous features". Both authors describe the strengths of a Random Forest as its robustness in over fitting, low training times, low training complexity and fast prediction compared to other models.

Random Forest models are frequently used as classifiers for anomaly detection in problems such as network intrusion detection. As both authors mention, the Random forest model often outperforms other models in its effectiveness when performed on datasets such as *KDD99* and others. In Figure 2.10 an example tree illustrates the way a trained random forest tree votes.

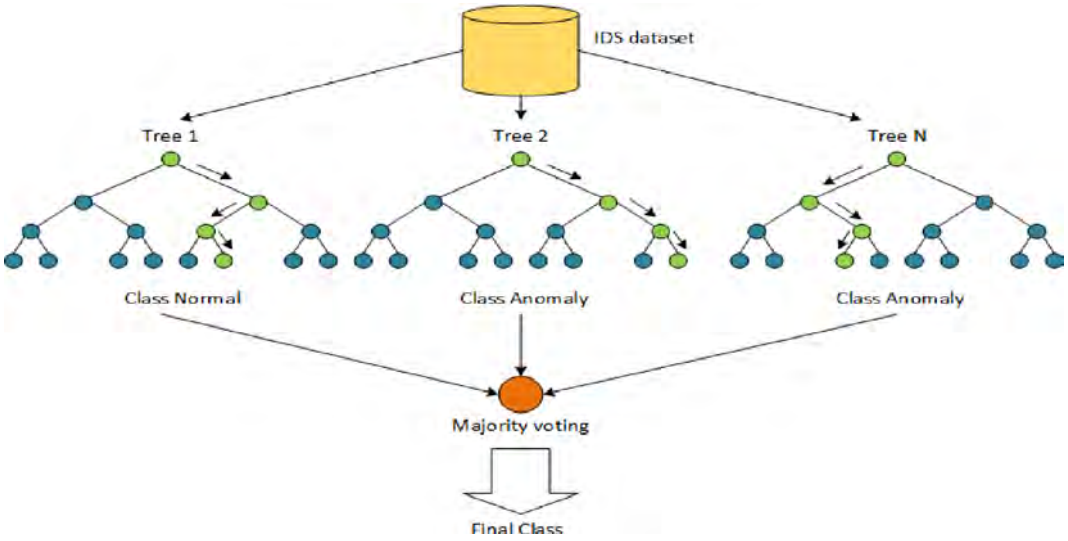

Figure 2.10: Random forest decision between trees on a data point taken from Primartha and Tama (2017, p.6)

Isolation Forest

The Isolation Forest machine learning or anomaly detection algorithm was proposed by Liu *et al.* (2008). The algorithm was a deviation from the usual methods for anomaly detection, which focus on modelling normal instances, but are not optimised for detecting anomalies. The authors noted that existing methods will have lower efficiency as too few

anomalies are detected, as compared to false positives. The authors also identified that existing methods "are constrained to low dimensional data and small data size because of their high computational complexity" (Liu *et al.*, 2008, p.1).

The method proposed by the authors focuses on isolated anomalies rather than profiling normal instances of data. This is illustrated in Figure 2.11 where an anomaly is isolated compared to normal data. The proposed method achieves this through focusing on quantitative properties of two anomalies (Liu *et al.*, 2008, p.1):

1. They are the minority consisting of fewer instances.
2. They have attribute-values that are very different from those of normal instances.

The authors simplified the explanation of the above properties with the description of anomalies as infrequent and different to other samples, which enables the isolation of these properties (Liu *et al.*, 2008). To isolate these instances a tree structure is constructed, and due to their nature the anomalies are isolated "closer to the root of the tree", as opposed to a point traversing further down the tree for normal behaviour (Liu *et al.*, 2008, p.1). Thus the authors' method creates a number of isolation trees for a dataset where the anomalies are instances with "short average path lengths on the iTrees", as seen in Figure 2.12 (Liu *et al.*, 2008, p.1).

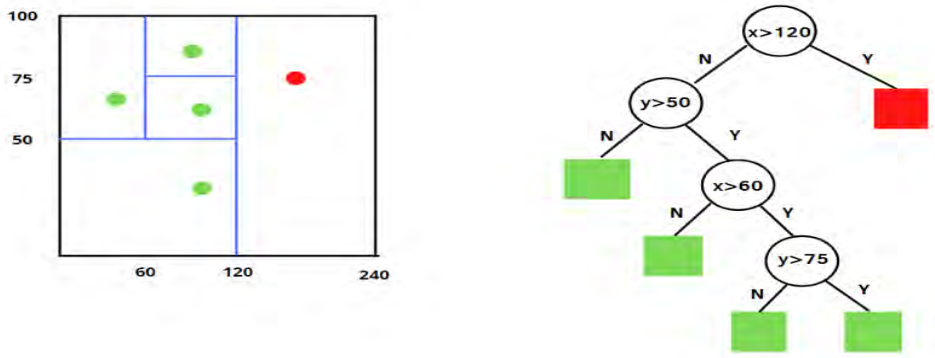

Figure 2.11: Isolation Forest tree with the anomaly isolated taken from Anello (2021).

The authors validated their model on two interesting datasets, with relevance to information security: "HTTP (KDDCUP99)" and "SMTP (KDDCUP99)" (Liu *et al.*, 2008). These datasets are from a 1999 competition to develop a network intrusion detector based on models trained with intrusion examples. The authors compared their algorithm with algorithms such as LOF and achieved high area under the curve (AUC) scores as seen in Figure 2.13. Using these datasets and others the authors identified a high degree of accuracy, low computation complexity and efficiency in varying sized datasets.

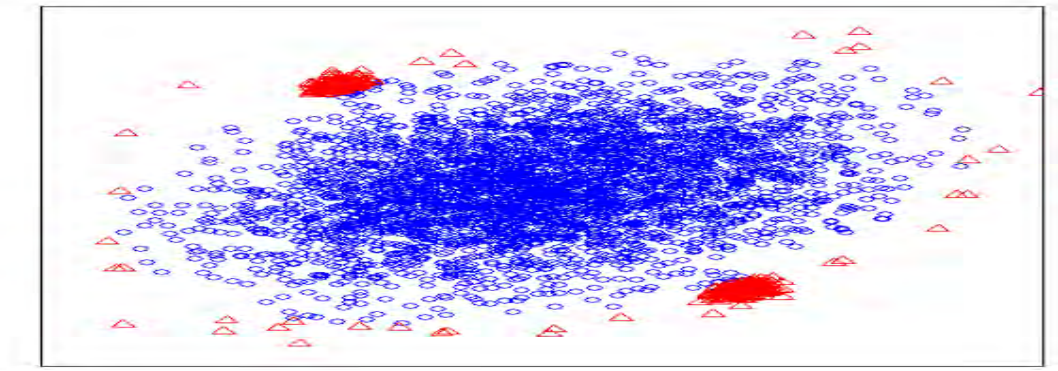

Figure 2.12: iForest anomalies graphed with normal instances taken from Liu *et al.* (2008).

	AUC				Time (seconds)					
	iForest	ORCA	LOF	RF	iForest			ORCA	LOF	RF
					Train	Eval.	Total			
Http (KDDCUP99)	**1.00**	0.36	NA	NA	0.25	15.33	**15.58**	9487.47	NA	NA
ForestCover	**0.88**	0.83	NA	NA	0.76	15.57	**16.33**	6995.17	NA	NA
Mulcross	**0.97**	0.33	NA	NA	0.26	12.26	**12.52**	2512.20	NA	NA
Smtp (KDDCUP99)	**0.88**	0.80	NA	NA	0.14	2.58	**2.72**	267.45	NA	NA
Shuttle	**1.00**	0.60	0.55	NA	0.30	2.83	**3.13**	156.66	7489.74	NA
Mammography	**0.86**	0.77	0.67	NA	0.16	0.50	**0.66**	4.49	14647.00	NA
Annthyroid	**0.82**	0.68	0.72	NA	0.15	0.36	**0.51**	2.32	72.02	NA
Satellite	**0.71**	0.65	0.52	NA	0.46	1.17	**1.63**	8.51	217.39	NA
Pima	0.67	**0.71**	0.49	0.65	0.17	0.11	0.28	**0.06**	1.14	4.98
Breastw	**0.99**	0.98	0.37	0.97	0.17	0.11	0.28	**0.04**	1.77	3.10
Arrhythmia	**0.80**	0.78	0.73	0.60	2.12	0.86	2.98	**0.49**	6.35	2.32
Ionosphere	0.85	**0.92**	0.89	0.85	0.33	0.15	0.48	**0.04**	0.64	0.83

Figure 2.13: iForest results on various datasets (Liu *et al.*, 2008, p.9)

2.10.5 NLP and isolation forest

Liu *et al.* (2020) also identified that system call sequences or methods like BoSC or other frequency based methods could be combined with four different anomaly detection algorithms. The algorithms identified and studied by the author for anomaly detection were LOF (Local Outlier Factor), Isolation Forest, One-class Support Vector Machine and K-Nearest Neighbors global.

The authors propose a new feature extraction method they had developed, which they named statistical pattern-based feature extraction method (STP) (Liu *et al.*, 2020). The goal of the author's new method is to study a platform independent set of features that could be used in research and anomaly detection. The method "transforms the system calls in a trace into frequency sequences of n-grams and then explores a fixed number of statistical features on the frequency sequences" (Liu *et al.*, 2020, p.1). A similar goal is envisioned with BoSC, where the method must achieve a combination of lightweight as well as stable and accurate detection. A one class classification is applied to account for

normal data being abundant and abnormal data sparse and difficult to obtain. Hence the author would only use normal data to extract features for training and that abnormal data would be any differing patterns to the normal classifications.

The authors focused on host-based anomaly detection and chose well known academic datasets for the paper. The three datasets chosen by the authors ADFA-LD, ADFA-WD, NGIDS-DS were used with the four anomaly detection algorithms, to test the efficiency of the STP classification or feature extraction method (Liu *et al.*, 2020). The authors also evaluated hybrid and frequency based methods for feature extraction, including the BoSC and tf-idf methods, against the STP method.

The results of the research by Liu *et al.* (2020) showed an increase in performance in the AUC and a strong performance when applied to Isolation Forests. The authors note that the STP method takes more time processing than other methods such as BoSC or tf-idf.

2.10.6 Hyper-Parameter tuning

Training and evaluating machine learning models involves tuning various parameters to achieve a model that fits the data and problem the data scientist aims to solve. Elgeldawi *et al.* (2021) describe how machine learning models use data to adjust their internal parameters and these are referred to as model parameters or just parameters. These parameters are adjusted by the model during the fitting of the data, but as described by the author the model itself has parameters tuned before the learning process. These parameters are known as *hyper-parameters* and Elgeldawi *et al.* (2021, p.2) describes the difference as the model parameters indicate the conversion of input data to desired output data, as compared to hyperparameters that indicate the model structure itself.

The importance of choosing hyperparameters is highlighted by Li *et al.* (2018, p.1) stating that: "The values of hyperparameters configure the structure and other characteristics of the model and can significantly influence the training result...given the complexity of the model algorithms and the training processes, identifying a sweet spot in the hyperparameter space for a specific problem can be challenging". Hyper-parameters affect the performance of the algorithms in the model and the authors describe the time consuming and difficult process of adjusting these parameters and repeating the learning process to test for hyper-parameter selection (Elgeldawi *et al.*, 2021). As noted by the authors the defaults of the models when implemented in the software package or library are often sufficient or accurate enough for a dataset and problem, but to increase further accuracy

and tweak the model, automated methods are required to find the most accurate hyper-parameters from a range of possible values. This process is referred to as hyper-parameter optimisation.

Yu and Zhu (2020) identify multiple algorithms for hyper-parameter optimization and the various advantages and disadvantages of the algorithms. Table 2.1 from the authors identifies various algorithms and their attributes to various problems and issues such as a lack of efficiency. As described by Boyle (2019), Grid search and Random search are methods that can be used to optimize hyperparameters, where the former iterates over an exhaustive grid of parameters and the latter uses random samples of a range to determine parameter optimization. The yellowbrick [7] library also contains many visualization tools within its suite of Feature Analysis Visualizers to aid in the optimization and selection of parameters (scikit yb.org, 2021):

1. Rank Features: rank single and pairs of features to detect covariance
2. RadViz Visualizer: plot data points along axes ordered around a circle to detect separability
3. Parallel Coordinates: plot instances as lines along vertical axes to detect classes or clusters
4. PCA Projection: project higher dimensions into a visual space using PCA
5. Manifold Visualization: visualize high dimensional data using manifold learning
6. Direct Data Visualization: (aka Jointplots) plot 2D correlation between features and target

Hyper parameter tuning can be used to find the most optimal hyperparameters in unstructured text data with high dimensionality. This will allow discarding unnecessary features and improve the efficiency and accuracy of the model. The sklearn library has a *GridsearchCV* algorithm which combines a Gridsearch of parameters with cross-validation. Cross-validation is described by Payam *et al.* (2016) as a "statistical method of evaluating and comparing learning algorithms by dividing data into two segments: one used to learn or train a model and the other used to validate the model". Thus the sklearn library combines the Gridsearch algorithm with cross-validation, which is configurable, to achieve higher accuracy optimizations.

[7] https://www.scikit-yb.org/en/latest/api/text/freqdist.html

Table 2.1: Comparison between major hyper-parameter optimization algorithms (Yu and Zhu, 2020)

	Advantage	Disadvantage
GridSearch	* Simple * Parallelism	* Curse of dimensionality
Random search	* Parallelism * Easy to combine with early stopping methods	* Low efficiency * Cannot promise an optimum
Bayesian optimization	* Reliable and promising * Foundation of many other algorithms	* Difficult for parallelism * Conceptually complex
Multi-bandit methods	* Conceptually simple * Computationally efficient	* Balance between budget and number of trials
PBTmethods	* Combine HPO and model training * Parallelism	* Constant changes to computation graph * Not extendable to advanced evolution

2.11 Summary

This chapter reviewed of topics such as malware analysis, prudent experimentation, system calls analysis, natural language processing, anomaly detection and supervised and unsupervised machine learning models.

A large body of work and research exists on system calls and count vectorizing system calls for further analysis. Good academic guidance is also provided for prudent and real-world malware dynamic analysis, which produces system call logs for analysis. Additionally most academic datasets used in research are dated or do not have sufficient hyper parameters, but a dataset exists in the form of the LID-DS dataset.

The tree classifiers of Isolation Forests and Random Forests are well studied in anomaly detection and studies with NLP and these models are sufficiently represented in the literature.

Finally hyper-parameter optimization is discussed with algorithms like Gridsearchcv allowing exhaustive searches of parameters influencing the fitting of the model. These parameters are tested before training, to improve the accuracy of the classification model.

Chapter 3

High Level Research Design and Methodology

This chapter explains the research approach and gives a high level overview of the experimental design. It also provides justifications for some of the design decisions.

3.1 High Level Overview of Research

This research set out to explore anomaly detection with NLP and machine learning on Linux system syscalls to study the viability of detecting security intrusions and incidents. The usage of collective anomaly detection, through NLP and machine learning techniques, is studied for its viability as a IDS. To achieve the research objectives, supervised and unsupervised machine learning are bench-marked on a dataset containing anomalies and normal system call traces.

3.1.1 Foundation for research design

The work by Rossow *et al.* (2012, p.1) was used as the guiding framework in designing an experiment that adheres to the following aspects: "transparency, realism, correctness, and safety for collecting and using malware datasets". The VirusTotal student datasets accessed by the researcher allows for transparent and realistic sources of data, which, with additional study and experimentation, could provide further strengthening in the field of NLP of syscall logs and traces. The safety of using malware data sources is paramount, as the VirusTotal dataset only includes malware samples, hashes and some sources of metadata. The VirusTotal malware samples need dynamic analysis on a Linux based system to acquire syscall traces. The safety of the virtual environment, host and connected networks are of utmost importance. This research adds to the body of work studied by Rossow *et al.* (2012), as only a quarter of the papers studied were based on execution-driven datasets and the authors provided no details of the security precautions taken to ensure safety.

3.1.2 High-level experimental steps

For the purpose of this research Figure 3.1 illustrates the design and flow of the experiment. The figure and the below steps form the basis of the experimental work for this research.

The high-level steps for the malware and machine learning experiment are as follows:

1. Gather system call trace data sources from the public data source Leipzig Intrusion Detection dataset (LID-DS) download page.
2. Gather and download Virustotal malware samples. Then execute malware in prudent, safe and realistic environment and capture traces.
3. Combine public data sources with executed malware syscall traces. The malware traces are added to the public data source to further test the effectiveness of the models. The LID-DS traces are divided between benign and malicious, whereas the malware traces are all benign. These sources combined will form a single dataset for this research, hereafter referred to as the research dataset.
4. Load text based data as documents, each log trace gathered represents a single document.
5. Tokenize and vectorize the data by term frequency and inverse document frequency. During the tokenization step, the *TfidfVectorizer* from sklearn automatically cleans the data and has a token pattern that removes punctuation and stop words. Document parameters or number of vocabulary selected was 50000.
6. Tag document as malicious or benign. This tag, target or otherwise Y value is the actual classification or output expected of the model. This step forms the Y variable for the models in the form of a target or Y value.
7. Remove and store the target or Y value for model training (in the case of supervised learning) and accuracy. The target or Y value, is the tag or target value described in the previous step.
8. Apply dimensionality reduction with principal component analysis (PCA) and graph PCA variance to identify the optimal number of components. A PCA component number of eighty percent of variance is selected to preserve a large majority of the variance that will affect the model, while discarding features that have little effect on the outcome of the model.

9. Fit data to the two machine learning models i.e. Random Forest and Isolation Forest decision trees.

10. Compare model accuracy scores, training time and AUC scores.

11. Repeat above three comparisons on the test data to validate the model on unseen data.

12. Apply different parameters to GridsearchCV to identify the most optimal parameters for model fitting.

13. Change model initialization parameters with the outputs of the GridsearchCV.

14. Repeat steps from 8 to 10 and compare all four experiments. These four experiments are namely: Random Forest, Isolation Forest, Random Forest with optimal parameters and Isolation Forest with optimal parameters.

15. Compare two models for the most accurate result.

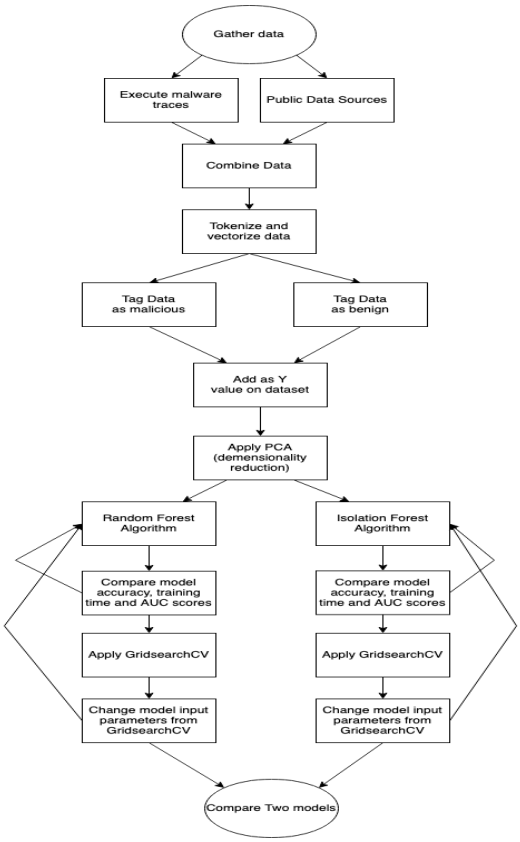

Figure 3.1: Flow chart of the high-level experiment

3.1.3 Justification for choice of technologies

Research on the application of unsupervised machine learning with NLP in security has a long history in academia. Supervised learning on security systems and host based intrusion detection has an even longer history with datasets and promising results. Anomaly detection in host systems continues to be a promising avenue of research and more tools are available for efficient models and processing of text based unstructured data.

The seminal work by Liu *et al.* (2008) on modelling anomalies in a tree through isolation, rather than modelling normal data, provides a strong basis for this research. When modelling the frequency and location of events within a log, using term frequency and inverse document frequency, anomaly detection is predicted to be efficient and accurate (Ioannidou, 2021). Thus frequency of words with Isolation Forests provides a solid base for detecting and isolating anomalies.

It is further expected that anomalies in a log, much like in human language, will present as infrequent and out-of-place speech. This allows identifying potential anomalies as they would be infrequent and not match the frequent system calls.

Research by Younis *et al.* (2016) identifies various levels and the names of dangerous Linux system calls. The usage of a table of dangerous Linux system calls can further enrich the data or be used to remove benign samples of system call traces, that would otherwise be marked as malicious. Thus the dataset can be enriched and it is expected that a model can be trained with only the dangerous system calls.

3.2 Tooling, Transparency and Safety

To adhere to principles set out by Rossow *et al.* (2012), the safety, transparency and correctness of the study includes a detailed description of the tooling and methods applied to the research. Dynamic analysis of malware requires prudent care to protect both hosts and networks, but still requires enough real world setup to allow the malware sample to execute under expected conditions.

3.2.1 Data analysis environment

For the analysis of the data, an Ubuntu Desktop 20.04 based computer was used. The desktop was upgraded to Ubuntu Desktop 21.04 version as the research progressed. This

Table 3.1: Data analysis machine screenshot of OS and hardware information

Item	**Value**
Device Name	goose-starship
Memory	15.6 GiB
Processor	AMD Ryzen 7 5800x 8-core processor x 16
Graphics	llvmpipe (LLVM 11.0.1, 256bits)
Disk Capacity	2.0TB
OS Name	Ubuntu 21.04
OS Type	64-bit
Gnome Version	3.38.5

computer is normally used for various Python based projects and thus ideal for Python data analysis with Jupyter Notebooks[1].

To adhere to the transparency aspect of the work by Rossow *et al.* (2012), as much info on the analysis environment is provided, as seen in Table 3.1.

None of the execution of malware or gathering of normal / abnormal data was done on this computer to prevent malware altering the data recorder or introducing noise into the dataset from existing services installed.

Once the initial setup for the analysis machine was tested on the previously mentioned Ubuntu desktop machine, an Amazon AWS EC2 instance was employed to handle larger data volumes with more CPUs and larger RAM allocations than the desktop PC could handle. The Amazon EC2 instance was allocated an Amazon EPYC CPU to stay on the same platform, with 16 vCPUs and 64GB of RAM.

3.2.2 Host machine

Ubuntu 20.04 LTS Desktop[2] was used on a Lenovo Thinkpad T460p. This host machine was used to install Virtualbox and run virtual machines for malware analysis only. The Ubuntu Desktop version was installed, as the virtual machines would also use Ubuntu Desktop as well as the display.

To protect the machine during malware execution on virtual resources, the latest updates were installed and Inetsim (simulating common internet services in a lab environment) [3] was used to mimic Internet services from one virtual machine to another.

[1]https://jupyter.org/
[2]https://wiki.ubuntu.com/FocalFossa/ReleaseNotes
[3]https://www.inetsim.org/index.html

3.2.3 Virtual machine

The execution environment consists of a Ubuntu 16.04 LTS Desktop[4]. The Ubuntu install is run in a Virtualbox virtual machine, running on the host machine mentioned in the previous section. Some assumptions have been made with the version of the OS. Considering that the malware samples in the VirusTotal Academic access are from 2017, 2018 and 2019, Ubuntu 16.04 LTS Desktop was chosen to create an environment most suitable for the malware to execute in. As seen in Table 3.2, Ubuntu 16.04 LTS was released on 21 April 2016, with end of life at the end of April 2024. This falls within the time frame of the malware sample discovery, so any malware checking the kernel and version with *uname −a* or the Ubuntu version with *lbs_release −a*, would determine it to be current and thus unlikely to resort to anti-forensic behaviour.

Table 3.2: Ubuntu 16.04 release dates (Ubuntu, 2021)

Ubuntu **16.04.7 LTS**	Xenial Xerus	Changes	August 13, 2020	April 2021	April 2024
Ubuntu 16.04.6 LTS	Xenial Xerus	Changes	February 28, 2019	April 2021	April 2024
Ubuntu 16.04.5 LTS	Xenial Xerus	Changes	August 2, 2018	April 2021	April 2024
Ubuntu 16.04.4 LTS	Xenial Xerus	Changes	March 1, 2018	April 2021	April 2024
Ubuntu 16.04.3 LTS	Xenial Xerus	Changes	August 3, 2017	April 2021	April 2024
Ubuntu 16.04.2 LTS	Xenial Xerus	Changes	February 16, 2017	April 2021	April 2024
Ubuntu 16.04.1 LTS	Xenial Xerus	Changes	July 21, 2016	April 2021	April 2024
Ubuntu 16.04 LTS	Xenial Xerus	Release Notes	April 21, 2016	April 2021	April 2024

Additionally the Desktop version was used to allow the execution of any malware to execute that required, checked for, or exploited X.org[5] server running on Linux. An additional step taken to ensure the execution of the malware was the allocation of appropriate virtual resources allocated to the virtual machine. Malware authors have become aware of the fact that malware researchers and sandboxes have limited resources on host machines, thus allocating 1 CPU core and between 1 and 4GB of RAM (Kovacs, 2015). Kovacs (2015) states, numerous malware families have sandbox detection and evasion techniques, with some leveraging multiple methods. Thus to create a realistic environment for execution, and also a realistic dataset, multiple considerations are to be taken by the researcher of a malware sample.

For the purpose of this research, the executed malware datasets need to adhere to the principles of realistic, transparent and safe as stated by Rossow *et al.* (2012). Finally we need to isolate the malware from the Internet, yet we need to simulate realistic network activity. As Rossow *et al.* (2012) states: certain malware requires being connected to c&c

[4]https://releases.ubuntu.com/18.04.5
[5]https://www.x.org/wiki/

servers, which then enables or activates benign or malicious behavior. This behaviour is a mitigation built in to detect analysis of the malware. For this purpose the decision was made to use Inetsim, instead of disabling network access. For additional safety, the virtual machine was updated to the latest version of Ubuntu to prevent the malware breaking out of the virtual machine.

3.2.4 Capturing traces

This section explains the tools and techniques used to capture and process traces for this research.

Sysdig

Sysdig[6] is a debugging, troubleshooting or tracing tool, released open-source, developed to solve the link between host and network which is often missing in other tools (Degioanni, 2014a). The authors of the tool offer many open source tools and paid for services in the visibility of large systems and cloud tools, such as containers and Kubernetes[7]. The sysdig tool allows tracing both network and host events with customizable outputs and can trace containers without being installed on the client. Degioanni (2014a) and the sysdig team aim to solve the question of which process is involved in the sending of data on the network, with additional functionality for monitoring program execution.

Though the output from sysdig can be customized, the default output for tracing a process is illustrated in Figure 3.2. Columns logged by sysdig include the command executing the process, which processor core it is running on, the thread id, event type and arguments. The time, sequence, event type (or syscall) and arguments passed to the syscall are crucial to the dataset for this research. An example of a sysdig trace of syscalls, from a VirusTotal sample given in Listing 3.1 illustrates the execution and captured information needed to obtain all the context from the process.

[6]https://sysdig.com/

[7]Kubernetes is an open-source container-orchestration system for automating computer application deployment, scaling, and management

Listing 3.1: Example and partial output of sysdig trace of a VirusTotal sample

```
cat 2017-10-20-28d5f75e289d652061c754079b23ec372da2e8feb1066a3d57381163b614c06c-x64.scap
dirfd=20(<d>/root/etc) name=localtime flags=4161(O_NONBLOCK|O_RDONLY|O_CLOEXEC) mode=0 dev=0
09:48:08.573331834 exe 1919 > sendmsg fd=6 size=11 tuple=NULL fd=6 size=11 tuple=NULL
09:48:08.573334594 exe 1919 < sendmsg res=11 data=..... res=11 data=.....
09:48:08.573338549 exe 1919 > recvmsg fd=6 fd=6
09:48:08.573339108 exe 1919 < recvmsg res=-11(EAGAIN) size=7 data= tuple=NULL res=-11(EAGAIN) size=7
    data= tuple=NULL
09:48:08.573339931 exe 1919 > ppoll fds=6:?1 14:e1 timeout=none sigmask= fds=6:?1 14:e1 timeout=none
    sigmask=
09:48:08.573341620 exe 1919 > switch next=1940(exe) pgft_maj=0 pgft_min=9 vm_size=121160 vm_rss=12732
    vm_swap=0 next=1940(exe) pgft_maj=0 pgft_min=9 vm_size=121160 vm_rss=12732 vm_swap=0
09:48:08.573342803 exe 1940 < ppoll res=1 fds=8:?1 res=1 fds=8:?1
09:48:08.573344004 exe 1940 > recvmsg fd=8 fd=8
09:48:08.573344500 exe 1940 < recvmsg res=7 size=7 data=..... tuple=NULL res=7 size=7 data=.....
    tuple=NULL
09:48:08.573348028 exe 1940 > recvmsg fd=8 fd=8
09:48:08.573348644 exe 1872 < futex res=0 res=0
09:48:08.573348855 exe 1940 < recvmsg res=4 size=4 data=.... tuple=NULL res=4 size=4 data=.... tuple=
    NULL
09:48:08.573351966 exe 1872 > futex addr=C00004AF48 op=128(FUTEX_PRIVATE_FLAG) val=0 addr=C00004AF48
    op=128(FUTEX_PRIVATE_FLAG) val=0
```

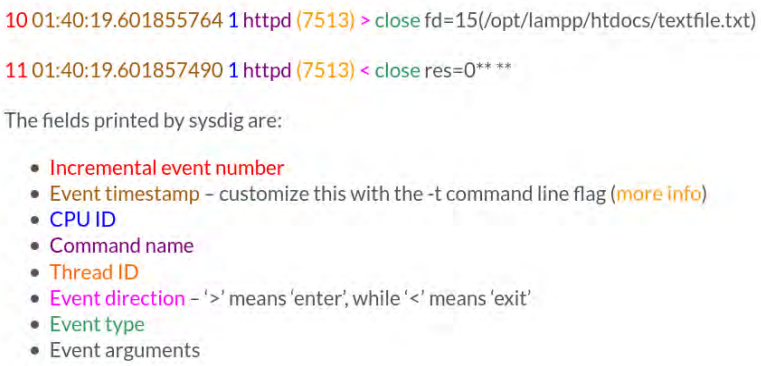

Figure 3.2: Sysdig process trace format example taken from Degioanni (2014b)

The instrumentation in sysdig allowing container and Kubernetes instrumentation for tracing host and network calls, is essential to the safety of the research experiment. The safe extraction of artifacts and the consistency of the data is ensured by sysdig not being present in the container or within the user access level of the container. Sysdig and the tracing is run on a level of the host or virtual machine, not within the container, preventing the malware from detecting or potentially modifying the tracing.

3.2.5 Python data processing

For the purpose of importing the raw data, processing it, tokenizing it and running unsupervised machine learning techniques, *Python3*[8] and *Jupyter Notebooks*[9] were used. The Python machine learning and artificial intelligence (AI) ecosystem is rich with many libraries and tools, making it easy for researchers outside of the data science field to engage

[8]https://docs.python.org/3/
[9]https://jupyter.org

Listing 3.2: Scikitlearn imports

```
from sklearn.metrics import classification_report,accuracy_score
from sklearn.ensemble import IsolationForest
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import LocalOutlierFactor
from sklearn.svm import OneClassSVM
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error
from sklearn.feature_extraction.text import TfidfVectorizer
```

with these tools. This aligns with the purpose of this research, to empower information security professionals and other defenders to utilize unsupervised machine learning techniques to strengthen incident response and detection.

For data importing and data frames, the *pandas*[10] library was used. Data is imported as a list, then added to a pandas data frame, for pre-processing, analysis, model training and validation. A pre-processing step was also used to turn normal text based data to labeled and syscall tagged CSV files, for easier and more efficient processing.

For NLP of the logs, the library Text Hero[11] is used for pre-processing and cleaning the raw text data. Text Hero's requirements are explained in the two sentences in its documentation overview: firstly it is difficult to get a quick understanding of a dataset comprising of text only and secondly that Text Hero provides functions for cleaning and converting text into vector space for further insights (Besomi, 2021). Text Hero is designed for NLP and includes features such as cleaning, PCA, k-means clustering, tf-idf vectorizing, word clouds and graphing data. Text Hero allows custom pipelines for preprocessing and mapping documents to a vector. Most of the Text Hero library is a wrapper for existing Python libraries such as *numpy*, *space*, *nltk*, *pandas*, *wordcloud*, *plotly* and *matplotlib*.

The *scikit-learn*[12] Python library, also referred to as *sklearn*, was extensively used in this research. The library has all the tools needed for NLP, and unsupervised and supervised learning models. Some of the features of sklearn that were used are listed in Listing 3.2.

[10]https://pandas.pydata.org/
[11]https://texthero.org/
[12]https://scikit-learn.org/stable/

Listing 3.3: Random and Isolation Forest with n_jobs parameter

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import IsolationForest

RandomForestClassifier(n_jobs=-1)
IsolationForest(n_jobs=-1)
```

3.2.6 Cloud processing of large datasets

System call traces, being unstructured and large volume data-sources, produce large datasets. These large datasets in their initial form pose major costs for storing, preprocessing, loading in training and comparing machine learning models: these are storage, CPU, random access memory (RAM), input and output amongst others.

To handle large datasets and reduce time and hardware costs, the usage of cloud resources was applied to these problems. Amazon EC2 instances [13] (virtual machines) were utilized for short periods of time to run Jupyter Notebooks. Many of the machine learning algorithms are able to do model training in parallel, using all the CPU cores. Isolation Forest and Random Forest models in Listing 3.3, have the following settings to initialize the model with all available computing resources.

The usage of virtual machines with more than 8 CPU cores and more than 64GB of RAM allowed for tweaking the TF-IDF vector representation's maximum features. The more features selected from the corpus, the larger the data needed to be processed by the model.

3.3 Data Sources

Data was gathered from multiple resources for this research. Firstly, LID-DS dataset from academic study, mixed with VirusTotal malware traces, were used allowing other researchers to recreate or verify the study. The public dataset can also be combined to create synthetic datasets for training models, as in the case of this research.

3.3.1 LID-DS dataset

The limitations identified by Grimmer *et al.* (2019) of existing datasets such as KDD 199, UNM, NGIDSDS and ADFALD not only related to the age and relevance of these datasets.

[13] https://aws.amazon.com/ec2/

The authors also found that thread information, or rather the lack of child processes or threads were not captured in these data sources, moreover there was limited metadata omitting such important indications of intrusion such as the arguments, return values and timestamps of system calls. Datasets such as UNM and AFDAlD only had numerical representations for the process ID or system call name only. Figure 3.3 illustrates the example references cited by Grimmer *et al.* (2019).

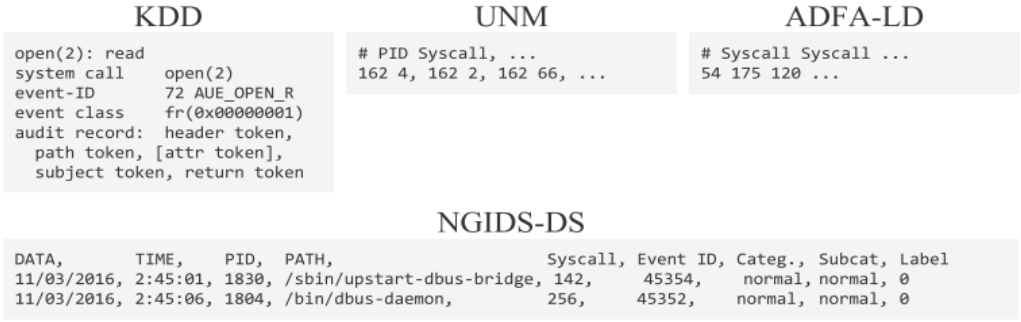

Figure 3.3: Examples of public academically cited and used datasets taken from Grimmer *et al.* (2019)

Grimmer *et al.* (2019) created new datasets using multiple attacks from Common vulnerabilities and exposures (CVE), on modern up-to-date systems. The CVEs selected by the authors included: Shellshock, Heartbleed, SQL injection and many more. The authors scripted the simulation scenarios to allow multiple users connecting to a web server, with many other scenarios simulated as well. Though normal user behaviour is difficult to model, the authors modeled behaviour on "On Off User behaviour", where the assumption is that a user waits while a request over the network is being fulfilled, for instance uploading a file. The authors also used tools such as *sqlmap* to generate data.

Grimmer *et al.* (2019) used Docker containers and sysdig to automate the tracing process as well as the victim and attacker scenarios. To achieve this the authors developed a framework named the *LID-DS framework* [14], not to be confused with the dataset of the same name, for recording HIDS datasets. The authors simulated 1000 runs of a scenario, for normal behaviour; as well as 100 runs for abnormal behaviour. The number of runs recorded over 30 seconds, produced 7000 normal sysdig traces and 1100 abnormal traces. The result of a single trace could include ten thousand or more system calls. These traces are captured and published as the LID-DS 2019 dataset.

[14]https://dbs.uni-leipzig.de/file/BSI-LID-DS.pdf

Table 3.3: List of scenarios in the LID-DS

Folder Name	Size	Scenario
CVE-2014-0160	148,2 MB	Heartbleed
PHP_CWE-434	649,1 MB	PHP file upload: unrestricted upload of file with dangerous type
Bruteforce_CWE-307	208 MB	Bruteforce login: improper restriction of excessive authentication attempts
SQL_InjectionCWE-89	689,4 MB	SQL injection with sqlmap various
ZipSlip EPS_CWE-434	6,6 GB	Unrestricted upload of file with dangerous type
CVE-2012-2122	155,1 MB	MySQL authentification bypass
CVE-2017-7529	41,1 MB	Nginx integer overflow vulnerability
CVE-2018-3760	388,7 MB	Sprockets information leak vulnerability
CVE-2019-5418	369 MB	Rails file content disclosure vulnerability

Listing 3.4: LID-DS 2019 dataset Bruteforce CWE307 run.csv log indicating runs and benign or malicious activity

```
image_name,scenario_name,is_executing_exploit,warmup_time,
recording_time,exploit_start_time
victim_bruteforce:latest,crashing_hamilton_4459,False,10,35,-1
victim_bruteforce:latest,salmon_bartik_3442,False,10,40,-1
victim_bruteforce:latest,flabby_lehmann_7321,False,10,45,-1
victim_bruteforce:latest,ancient_feistel_3121,False,10,35,-1
victim_bruteforce:latest,raspy_curie_2357,False, 10,50,-1
victim_bruteforce:latest,sweet_sutherland_3362,False,10, 40,-1
victim_bruteforce:latest,thousands_khayyam_2375,False,10,55,-1
...
```

Grimmer and Max (2019) grouped all the traces by scenario, as in Listing 3.3, where different scenarios created larger traces than others. Each scenario includes normal and abnormal traces. The authors of the dataset separated abnormal traces, thus making labelling easy for the dataset. In the dataset a*run.csv* file is provided for each scenario, as illustrated in Listing 3.4, with each run's name and whether the exploit was run or not during the trace. All files are text based, from 2019 and in the sysdig system call tracing format as in the example in Listing 3.5. The two traces in the example indicate both the capture of the system call and its arguments, as well as the event direction of the trace.

Grimmer and Max (2019) produced Table 3.4 to show the difference between their LID-DS 2019 dataset and previously cited academic papers in host based intrusion detection (Grimmer *et al.*, 2019, p.9). The dataset provides various scenarios for training models

Listing 3.5: Sysdig trace format and example trace as used in the LID-DS 2019 dataset (Grimmer and Max, 2019)

```
event_number event_time cpu user_uid process_name thread_id
event_direction event_type event_arguments

1159 00:27:12.310259734 6 33 apache2 15862 > writev
fd=12(<4t>172.17.0.21:46790->172.17.0.19:443) size=31
1160 00:27:12.310311837 6 33 apache2 15862 < writev
res=31 data=.....j*.@s.&.....4.
```

Table 3.4: HIDS datasets compared (Grimmer *et al.*, 2019, p.8)

	LID-DS	NGIDDS	ADFA-LD	UNM	KDD99
Process IDs	x	x		x	x
Arguments	x				x
Return Values	x				x
Timestamps	x	(x)			x
Not outdated	x	x	x		
Data buffers	x				
Amount of data	x	x			x

and are already labeled, which is ideal for this research. These are also large trace files, which can test the suitability of training these models.

3.3.2 VirusTotal

For the purpose of this research, a study was done of available malware or exploit datasets that could be verified and reproduced by other researchers. VirusTotal allows researchers to apply for academic access, which if granted, gives access to a vast number of malware samples across multiple operating systems and architectures (Virustotal.com, 2021). The dataset was chosen as the malware samples could allow dynamic analysis and verification against VirusTotal's metadata and extensive labelling of samples. To test unknown and real world data, access was obtained to the VirusTotal Academic dataset, covering virus samples between 2017 and part of 2019. The VirusTotal samples were run to create anomalous or outlier datasets to test the accuracy of the models to predict outlier system traces.

The samples and data provided by VirusTotal[15] are submitted by customers and the wider

[15]https://www.virustotal.com

Table 3.5: Example of dataset categories and structure VirusTotal (Virustotal.com, 2021)

Filename	Year
7ZIP.7Z	2019
ACE.7Z	2019
Android.7Z	2019
Apple software_package.7z	2019
C.7z	2019
C++.7Z	2019
DOS_EXE.7Z	2019
ELF.7Z	2019
HTML.7Z	2019
ISO_image.7Z	2019
MS_Excel.Spreadsheet.7Z	2019
Shell script.7Z	2019
Win32 EXE.7Z	2019
XML.7Z	2019

public of real-world evaluations and monitoring, thus Rossow *et al.* (2012) provide the correct practices for this research study. Rossow *et al.* (2012, p.65) provide a comparative Hippocratic oath[16] for malware research in the following statement: "For such research to reflect prudent science, the work needs to address a number of concerns relating to the correct and representative use of the datasets, presentation of methodology in a fashion sufficiently transparent to enable reproducibility, and due consideration of the need not to harm others".

Brief overview of dataset

The VirusTotal academic dataset was acquired on Monday, 10 June 2019 at 7.43am, and included samples from 2017 until just before the acquisition time. As with a responsible and ethical malware research institution, the dataset was archived with the password 'infected' to protect the client and prevent antivirus removing the file (Rossow *et al.*, 2012). The dataset is divided by year and specific language or mechanism used to deliver the payload to a client machine, as seen in Table 3.5.

The dataset, within a language/delivery classification, consists of both a file and a JSON metadata file as observed in Table 3.6. The filename of the sample is derived from its hash and in the metadata contains a MD5 and secure hash algorithm 256-bit (SHA256) hash, as well as the various partnered contributor's detection analysis of the sample

[16] `https://www.nlm.nih.gov/hmd/greek/greek_oath.html`

Table 3.6: Example of VirusTotal's dataset including the files named by hash as well as the JSON metadata (Virustotal.com, 2021)

Filename/Hash	Size
3ca2f08028fc9a09C9646f85421898844ab1ef9ec62d37e33eae8c0c3d633cff	177 bytes
3ca2f08028fc9a09c9646f85421898844ab1ef9ec62d37e33eae8coc3d633cff.json	7.4 kB
7e95f95f60ea8e08895750a89f4cfcec90930649b7a42d9f81642f9b2c28de60	177 bytes
7e95f95f60ea8e08895750a89f4cfcec9093C649b7a42d9f81642f9b2c28de60.json	7.7 kB
scb84f1ec3obd2732437b2118597ba19aa4a46050e1a667b12bcdb1f13aa5f15	1.8 kB
8cb84f1ec30bd2732437b2118597ba19aa4a46050e1a667b12bcdb1f13aa5f15.json	37.7kB
18abd809ec5661c00018d1e6eef625dfda70893C1d92766a3de215743ec4da1c.json	10.9 kB

(Developers.virustotal.com, 2021; Liu *et al.*, 2009). VirusTotal's detection samples are bolstered by contributor detections and evidence adding to the richness of the data (Developers.virustotal.com, 2021).

Subset of data collected

The VirusTotal dataset has many choices including Windows, Mac OSX and Android (Virustotal.com, 2021). For the purpose of this study, the subset of ELF binaries, specifically Linux X86_64 kernel ELF binaries were selected (Virustotal.com, 2021).

VirusTotal provided access to Windows and a family of multitasking, multi-user computer operating systems that derive from the original AT&T Unix (Unix) based malware through their academic research access process. The dataset includes signatures, some logs and the binary made available to VirusTotal in the process of potentially identifying a malware incident.

Executable and linking format (ELF) is the common standard for binary file formats on Unix and Unix-like systems (Bhatt and Ahmed, 2018). The header in ELF binaries allows the identification of Linux only binaries, for selection and running in a dynamic environment (Bhatt and Ahmed, 2018). Figure 3.5 shows example output from reading an ELF binary's header.

The VirusTotal academic resource of detected malware is significant, as it includes ELF binaries. See Figure 3.4 for an example of a detected ELF binary based malware on VirusTotal[17].

[17] https://www.virustotal.com/gui/file/e9fc1441bb88dd8cc7fcc2e176e53084ccd28f8766a017b3a07474b7e6b72ab9

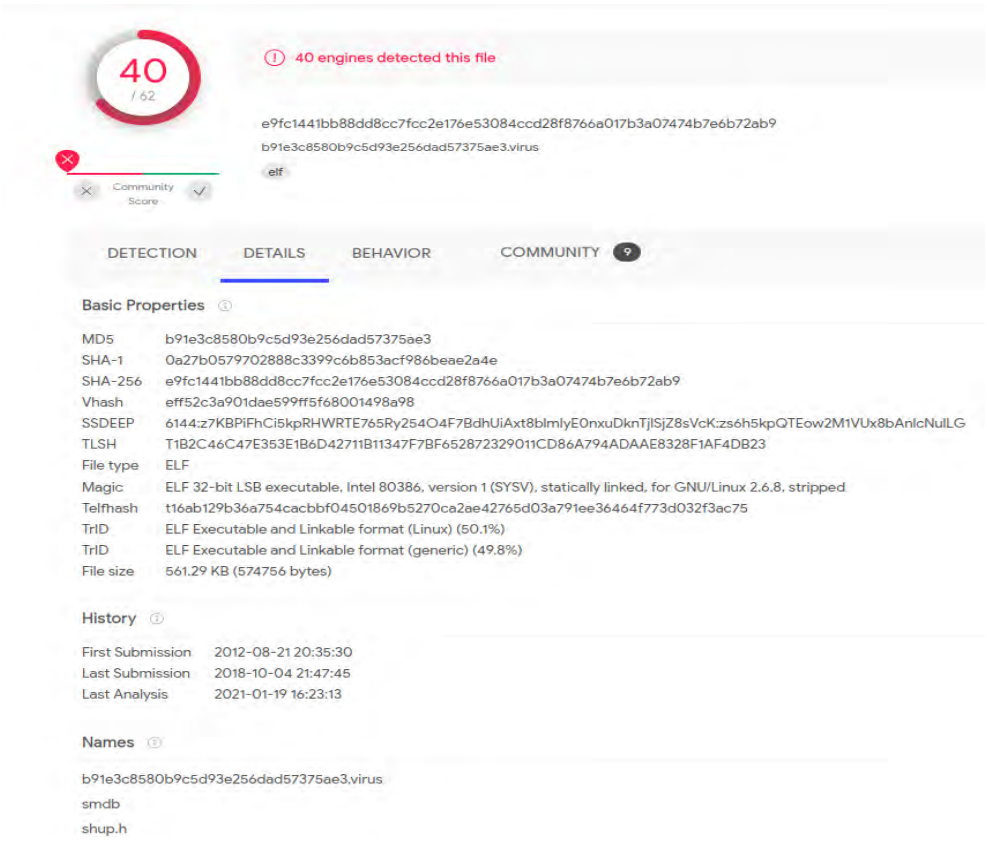

Figure 3.4: Example ELF malware on VirusTotal (VirusTotal, 2021)

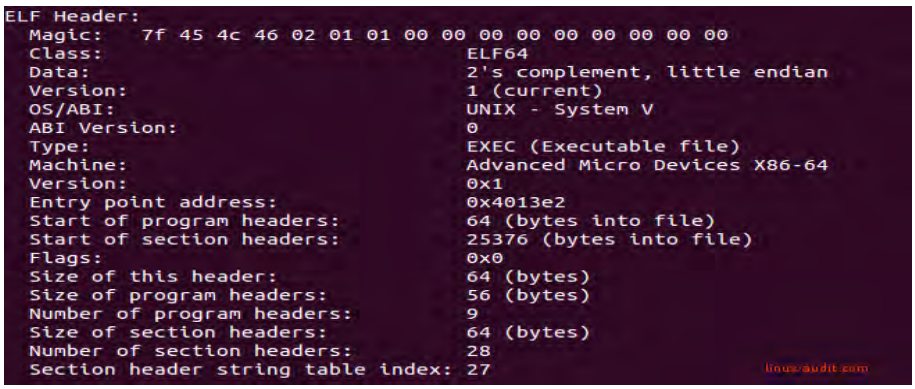

Figure 3.5: Example header of a ELF binary (Boelen, 2019)

The analysis of syscalls of a malicious binary or process requires the execution and dynamic analysis of a sample. Thus care must be taken in both the experimental setup as well, as in the methodology used. For this purpose the guide for this research was to "explore issues relating to prudent experimental evaluation for projects that use malware-execution datasets" (Rossow *et al.*, 2012, p.65).

With the use of Sysdig or strace[18], the system call execution logs or artifacts of the application were gathered and stored.

The malware samples were run in a virtual machine and isolated to prevent host compromise as well as to create repeatable environments with snapshots. The only difference between environments was the specific malware binary to be run.

3.3.3 Combined dataset - The research dataset

The system call traces of LID-DS and VirusTotal were combined to test the effectiveness of the models and the academic dataset. Both datasets have the same sysdig trace format, allowing the combination of the traces into a single dataset. For the duration of the experiment, the dataset will be referred to as the research dataset. This dataset could be split into training and test subsets, but would form part of the same dataset.

For the purpose of this research a single dataset was derived from two raw sources. Files were processed from the LID-DS dataset as well as execution system call sysdig traces from the VirusTotal academic data source. The dataset is derived from LIDS logs and VirusTotal total trace logs derived from executing 30 ELF binaries. The LID-DS logs were selected from 7 vulnerabilities/scenarios with 30 benign and 2 anomalous trace logs selected per scenario. Thus the dataset was derived from 224 files from the LID-DS dataset and 30 VirusTotal traces.

3.4 Processing of the Data

The research dataset in its raw form consists of unstructured text based data. To harness the research dataset most effectively, the unstructured data needs pre-processing, data labelling, tokenization and feature extraction.

3.4.1 Data pre-processing

As we were dealing with unstructured text based data, in the format of strace or sysdig output files, pre-processing the data into a predictable format that could be analyzed was required. For the purpose of this research on system logs, care in the pre-processing stage was needed, as processes such as lemmatization and normalization could affect the results.

[18]https://man7.org/linux/man-pages/man1/strace.1.html

This was due to the fact that some of the text data could refer to specific system calls or APIs, instead of abbreviations, as might be encountered in the NLP of tweets.

For the pre-processing and cleaning of the text, the Python library Text Hero was extensively used. The cleaning features of Text Hero include:

1. Filling NaN (Not a number), otherwise known as replacing not assigned values with spaces.
2. Changing text to lowercase.
3. Removing digits.
4. Removing punctuation.
5. Removing diacritics, which is described as "marks above, through, or below letters — are used in many orthographies to remedy the shortcomings of the ordinary Latin alphabet" (Wells, 2000). An example would be *déjà vu*.
6. Removing stopwords such as *but*, *again*, *is*, *an*, *and*, *the*.
7. Remove whitespace, including tabs and spaces.

The first step in the pre-processing of data, is the removal of noise from the logs. To assess the level of text preprocessing needed we refered to the table proposed by Ganesan (2019) and replicated in Table 3.7.

As the processing of application logs are domain specific and potentially noisy, with a lot of data, moderate pre-processing and text enrichment as stated in Table 3.7 was needed.

As a start, brackets and some special characters were removed using regular expressions (regex) on the data frames. With the removal of special characters we also removed hexadecimal numbers as part of some of the arguments for system calls. Additional noise was identified in backslash characters.

The expected removal of punctuation and numbers was facilitated by the Text Hero library functionality. As this research proposed the use of NLP on system call logs, we applied as many of Text Hero's cleaning and pre-processing tools that did not remove necessary context within the logs. For stopwords, only English stopwords were removed, while retaining necessary context for the logs.

Table 3.7: Level of text preprocessing needed (Ganesan, 2019)

	Domain Specific / Noisy Texts	**General / Well written Texts**
Lots of data	- Moderate pre-processing - Text enrichment could be helpful	- Light pre-processing - Text enrichment could be helpful, but not critical
Sparse data	- Heavy pre-processing - Text enrichment is important	- Moderate pre-processing - Text enrichment could be helpful

Listing 3.6: System call trace example

```
819   19:04:53 [400a9d7b] execve("/usr/sbin/traceroute",
["exploited", "-g",
"1.2.34", "-g",
"1.2.3.17
sex\374\377\377\377\270\307\4\10\211\377\377\277"...,
"\353\n\353\n\353\n\353\n\353\n\353\n\353\n\353\n\353\n"...],
[/* 0 vars */]) = 0 <0.000297>
```

Noise appeared in the form of special characters like / or * as well as hexadecimal, values passed to system call arguments in the traces. An example is given in the extract from a *strace* log in Listing 3.6. As NLP tools would be applied to these logs, this study focused on the language and words, not the characters that could form part of the exploit or anomaly.

3.4.2 Data labelling

For the purpose of comparing supervised and unsupervised machine learning models for the suitability of host-based intrusion detection on unstructured logs, labeled data was required for the supervised models.

The research dataset is tagged with target values to allow validating the accuracy of the model prediction. For the research dataset, the traces are tagged as normal and benign behaviour. The research dataset in its raw form has a column indicating benign traces as true or false, allowing for easy tagging of the document as 1 for benign and -1 for malicious. One simply adds a target (Y) or labeled abnormality column with either a -1 or 1 to indicate an abnormal or normal document. There is a larger proportion of benign to malicious traces.

3.4.3 Tokenizing and numerical feature vectors

The greatest challenge with text based data such as logs, is that it represents unstructured data. To process text based data with NLP, the data source is converted to numerical

feature vectors (Ioannidou, 2021). Thus we had to convert the unstructured text data into numerical representations for statistical and machine learning analysis.

To facilitate the conversion to vectors, the text is applied to a Bag-of-Words technique first. To use NLP methodology we processed the log data, including the cleaned data, as documents and sentences. From the cleaned logs we create an occurrence matrix of the words in sentences in the documents. Similarly to what Kang *et al.* (2005) proposed with a "bag of system calls", in this research we convert system calls and arguments into a matrix of occurrences. With this approach the occurrences are modeled in a matrix, as seen in the example in Figure 3.6, but context of position is lacking especially within the entirety of the document.

Figure 3.6: BOW representation of the data

To account for the context of system calls within the entire log, the technique *tf-idf* is applied with the BOW technique. As the technique models term frequency within the sentence and inversely in the document, it is useful as stated by Yse (2019, p.1): “frequent terms in the text are rewarded...they also get punished if those terms are frequent in other texts...On the contrary, this method highlights and rewards unique or rare terms considering all texts”. The previous statement by the author highlights the crucial component of this research, as unique and rare items, or anomalies, would be evident in the tf-idf representation of the text. An example of the output of tf-idf from the dataset is seen in Figure 3.7.

Listing 3.7: TfidfVectorizer vectorizer imported in python with the sklearn library

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf_vect = TfidfVectorizer(
    max_features=max_tf_idf_features,
    lowercase=True,
    use_idf=True,
    ngram_range=(1,1),
    smooth_idf=True,
    token_pattern=r"(?u)\b\S{2,}\w+\b",
    strip_accents='unicode',
    norm='l1')
```

	0.,.(.S.....k.j.i.h.9.8.7	0...0	0.0ms	0.9ms	0.cpu0	0000	0d107d09f5bbe40cade3de5c71e9e9b7./hackable/us	1',this	1.0ms	1.1ms	...	where	windows
0	0.0	0.0	0.0	0.0	0.0	0.002929	0.0	0.0	0.0	0.0	...	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.003626	0.0	0.0	0.0	0.0	...	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.003685	0.0	0.0	0.0	0.0	...	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.002641	0.0	0.0	0.0	0.0	...	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.004218	0.0	0.0	0.0	0.0	...	0.0	0.0
5	0.0	0.0	0.0	0.0	0.0	0.003194	0.0	0.0	0.0	0.0	...	0.0	0.0
6	0.0	0.0	0.0	0.0	0.0	0.003745	0.0	0.0	0.0	0.0	...	0.0	0.0
7	0.0	0.0	0.0	0.0	0.0	0.002653	0.0	0.0	0.0	0.0	...	0.0	0.0
8	0.0	0.0	0.0	0.0	0.0	0.003022	0.0	0.0	0.0	0.0	...	0.0	0.0
9	0.0	0.0	0.0	0.0	0.0	0.003163	0.0	0.0	0.0	0.0	...	0.0	0.0

10 rows × 50000 columns

Figure 3.7: A tf-idf representation of the data with 10 rows of documents, with the column names (50000) being the terms tokenized by the vectorizer

Feature extraction

The term frequency inverse document frequency method for extracting numerical features into a vector from text, maps each tokenized word or term into a feature between 0 and 1, based on its term frequency and inverse document frequency. Though system call traces or logs are based on human language, some of the tokens or characters do not represent human speech and can include arguments for the system calls. Preprocessing would have changed some of the non-Unicode or hex data, but feature engineering is still necessary to make sure that an adequate number of features are selected to train the model, without overwhelming the resources available for training and prediction.

The tf-idf vectorizer in sklearn allows the max_features parameter as shown in Listing 3.7, which limits the number of features used to fit the vector.

It is also important to note the `ngram_range`, which indicates the lower and upper boundary of n-values for different word n-grams to be extracted. An `ngram_range` of (1,1) extracts only unigrams which are one-word sequences.

According to the Chromium OS Documentation, a total of 384 Linux System calls are located in the Linux system call table for x86 or 32bit Intel/AMD arch of Linux (Google-source.com, 2021). For Intel/AMD 64 bit Linux a total of 332 system calls are identified by the website. This would necessitate a minimum number of 384 features for the converted text vector, to fit every Linux system call and its frequency.

3.4.4 Train and test split of research dataset

Once the research dataset was processed, tokenized and labelled, the following step was to split the dataset as is common in machine learning training. The dataset is split so that a model receives a subset of the data for training and another subset to validate its accuracy and effectiveness.

Using `train_test_split` method in the sklearn library the data was split with 0.33 of the data kept as a test and validation split, and 0.67 of the data kept for training the model. A pseudo random number of 42 was applied to the `random_state`, which is the attribute that controls the shuffling of the data before the split, allowing for a random or pseudo random distribution of the entries between train and test splits.

3.5 Data Distribution

Once the PCA dimensionality reduction was completed on the tf-idf vectors, the data was plotted to a scatter plot and bar plot. The scatter plot colours the data points by the y or target column of the dataset, hence anomalies or normal data points are represented in their distribution of the tf-idf. On the scatter plot the shades of blue represent the anomalies, whereas normal data points represent the red dots on the scatter plot. These colours were not selected for any specific reason, the defaults in the plotting library were merely applied.

The bar plot illustrates the distribution of the variance of the 20 features selected for the PCA. This bar plot is derived from the tf-idf vocabulary taken from 15 million words ingested. The vocabulary of 600 words was an arbitrary number chosen to limit the size of the data, but while tokenizing sufficient vocabulary for accuracy of the model.

3.5.1 Token frequency distribution bar plot

A frequency distribution bar plot was applied to the tf-idf processed text data, without splitting the data into the training and test subset. The enrichment of the dataset occurred on the target data, the y value, instead of the tokenized data of the x axis.

From the bar plot we can identify that tokens like *puma* and *apache2* are identified frequently in the research dataset. These tokens or sub-strings are from the parent process of the research dataset, indicating a web server or application that was traced for the system call logs and initiated as the main process. It is expected that these tokens and others such as *rails*, *var*, *futex*, etc. would feature frequently and would be scored lower by the tf-idf and subsequently the tree models as well.

We can also confirm that enough tokens for context are identified and that the pre-processing of tokens like numbers have not removed numbers in tokens such as *apache2*. This is important as Linux system calls can have numerical characters to indicate architecture (e.g. 64 bit) or the version of the process such as *apache2*.

The bar plot does not give us much information on anomalies, as they would be infrequent, but the frequent tokens give insight into the use of the tf-idf tokenization and vectorization effectiveness of the logs, as show in Figure 3.8, which is a representation of the top 50 tokens frequency. The figure shows us that common Linux system calls and their arguments are frequently tokenized and indicate that the tokenization is working.

3.5.2 Variance of PCA features to select number of components

PCA is an effective tool in decreasing the dimensionality of a dataset, but considerations are needed in balancing performance and the accuracy of the dataset. A PCA requires selecting the number of components that allow reducing dimensionality, but still retain the variance and accuracy of the dataset. In the context of the research aimed at practitioners that are not data scientists, the standard methods discussed by Rea and Rea (2016) were considered. The author describes many of these methods as rule of thumb, which include scree plots, Kaiser's rule and cumulative variance explained.

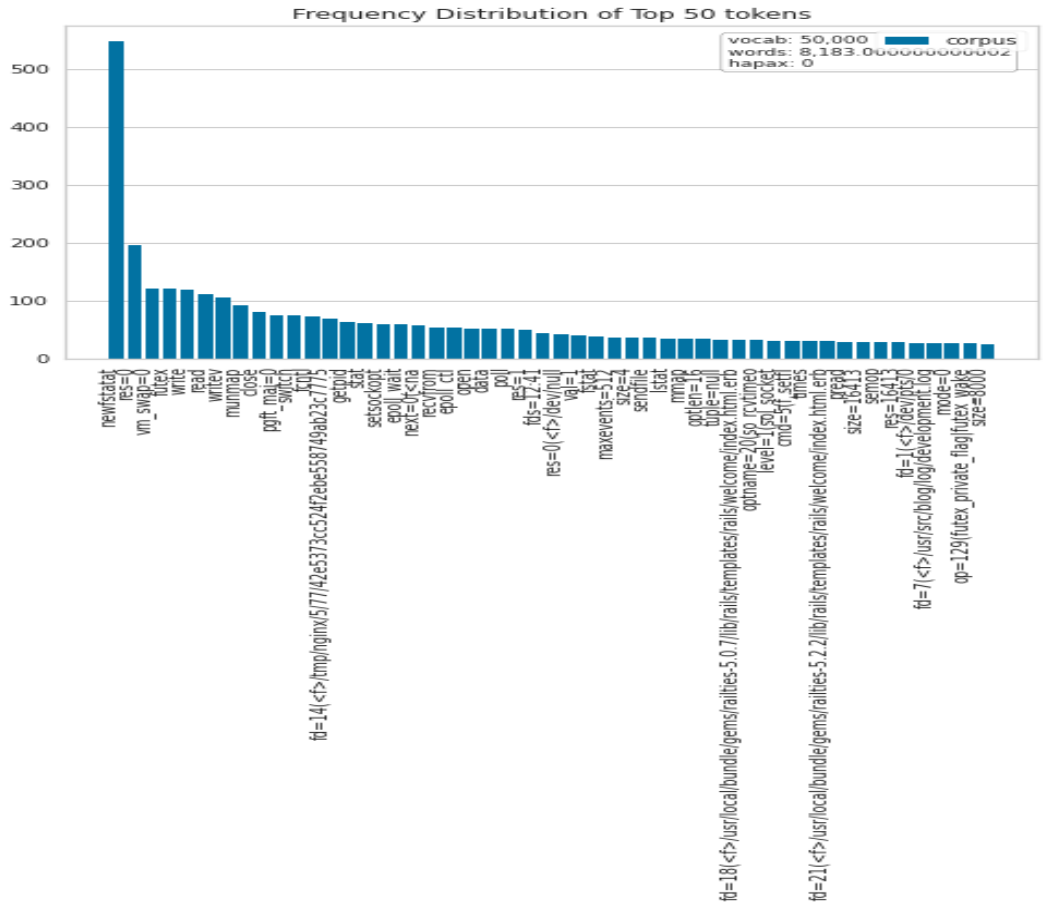

Figure 3.8: Frequency distribution of top 50 tokens from tf-idf

Within the context of the research and making the experiment accessible, the cumulative variance method described by Rea and Rea (2016) was applied to the data when fitting to a PCA. The data derived from the fitting and transforming of the Standard Scalar, was applied to the transform methods of PCA with default settings and as many components as the dataset could possibly have. With this instance of PCA, the feature variance and cumulative explained variance of the dataset could be graphed. As seen in Figure 3.9 the cumulative explained variance was graphed against the number of derived features or components in the PCA. Horizontal lines at 95%, 80% and 75% were plotted to identify the number of components that fell within these percentages.

To further display and analyse the variance of the various features, a PCA variance bar plot of the top 25 features was graphed in Figure 3.10. It was evident from the graph that the first feature, or feature 0, had the greatest percentage of variance compared with the other features, although features 1 and 2 also had a larger step down than many of the other features.

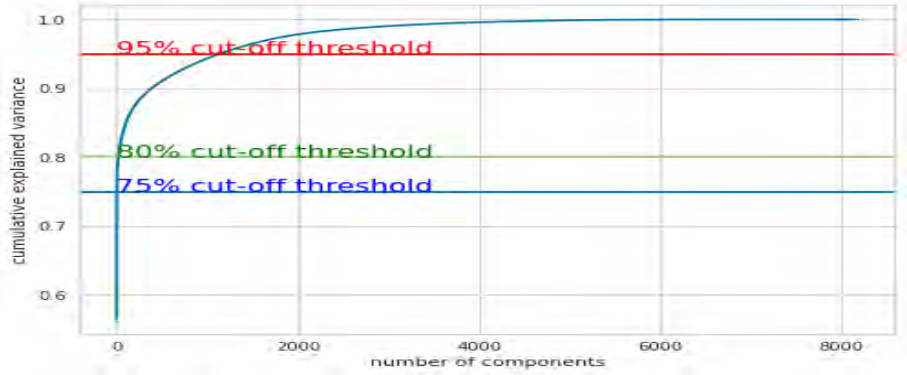

Figure 3.9: Cumulative explained variance plot of the PCA feature components

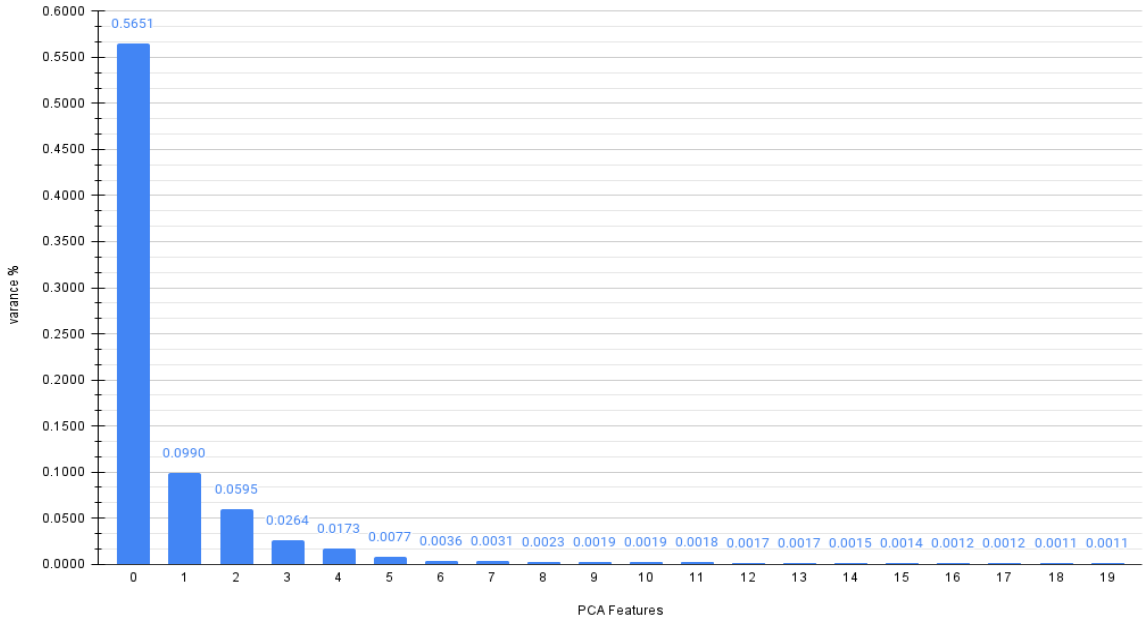

Figure 3.10: PCA bar plot features variance of first 20 features

Finally the cumulative variance plot in Figure 3.11 was graphed to find the number of features corresponding to a variance of between 70% and 80%, as a common tool proposed by Bellavia (2021). The author proposes selecting a number of components corresponding to 80% variance, as the maximum proposed explained variance from the original, which was chosen as the maxinum variance for this research. The 80% correlates to twenty components on the x axis, which is configured and applied to the PCA for the experiment. Based on Figure 3.10, a drop in variance can be identified after 20 PCA components, labeled as 19 on the graph with starting index of 0. For the purpose of finding efficient training times and still reaching a high level of accuracy in this research,

80% of 20 components was identified as the optimal percentage of variance.

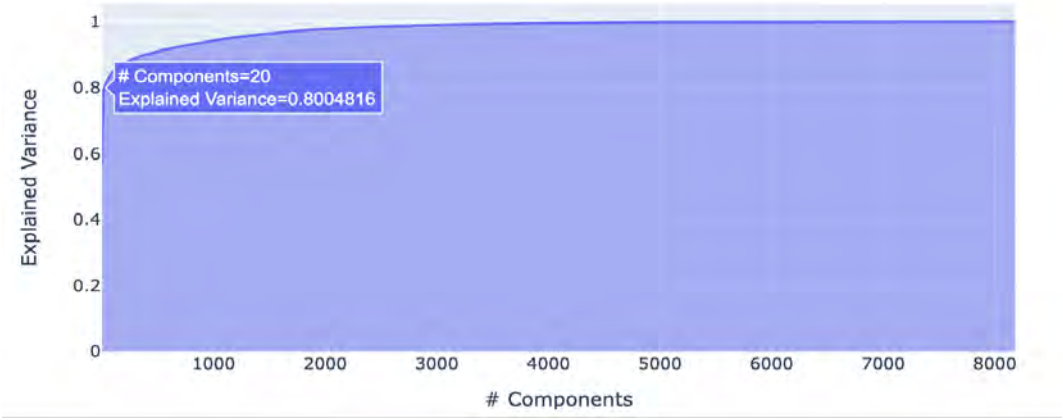

Figure 3.11: Cumulative explained variance plot of the PCA feature components

3.6 Principal Component Analysis

The unstructured text data nature of the datasets and having to capture more than 200 features to include all the Linux system calls in x86_64, means that the size of the data and its dimensionality becomes very large. In order to be able to interpret such large datasets Jolliffe and Cadima (2016) propose that it is necessary to apply methods to reduce the dimensionality of datasets drastically in an interpretable way, while most of the information is preserved. The authors describe PCA as an old and popular method of dimensionality reduction.

The PCA method is capable of discovering new variables that are linear functions of the variables in the original dataset (Jolliffe and Cadima, 2016). These variables successively maximize variance on the dataset and are uncorrelated with each other. The authors continue by describing PCA as a descriptive tool that needs no distributional assumptions, which lends itself to being an adaptive exploratory method. This method can be applied to numerical data of various types for exploration.

For this research study, PCA was used to reduce the dimensionality of the dataset, to enable producing scatter plots of the distribution of data, based on the classes of anomaly or normal, or rather 1 or -1. A tf-idf vector was fit to a PCA, which in turn produced a reduced dimension dataset for applying much quicker to the models, while still retaining the accuracy and context of the datasets within margins.

For dimensional reduction of the research dataset, PCA was applied with a limit to the top twenty components attributed. This created a sparse matrix of twenty features or

components which vastly decreased the size of the dataset to be fitted to the model, thus greatly decreasing the training time of the model. When investigating the number of features in the tf-idf dataframe that were input into the PCA, 50000 features were identified before the PCA was applied.

3.7 ML Models

To study the detection of anomalies two tree models, commonly used in anomaly classification, were selected to showcase the various aspects and performance relative to one another. With supervised models, labels are a requirement to train, test and evaluate the models, whereas unsupervised models do not require the labels, but the labels could be used to measure the accuracy and perform error analysis.

For the specific problem being solved and the nature of the anomaly detection classification problem, Random Forest and Isolation Forest models were chosen. The supervised learning model selected to test the detection of anomalies is the Random Forest model, while unsupervised learning model selected to perform the same task is the Isolation Forest model.

3.7.1 Hyper-parameter Tuning of models

Hyper-parameter tuning of both Isolation and Random Forest models was applied with the same parameters. Only the Isolation Forest needed one more parameter to be tuned.

The *n_estimators* parameter denotes the number of base estimators or trees in the forest. Both tree models, the Isolation and Random Forest trees, have a default number of estimators at a 100. To tune and test the models the estimators were tested below the default with 20 and 50, the default of 100, and above the default up to 500 parameters. As illustrated in Listing 3.9 the estimators are loaded into the model as a list, iterating over the model multiple times.

The *max_samples* parameter is the maximum number of samples drawn from the train_pca (referred to as the X provided to the classifier) to train each tree in the forest. To tune the model and iterate over multiple sized samples, a list was passed as illustrated in Listing 3.9, from one sample to the number of items in the PCA.

Another thing to note is that the Isolation forest requires a contamination factor, unlike the Random forest. The contamination number denotes the number of outliers and is used

Listing 3.8: Calculating outlier or contamination factor within the data before train and test split

```
Fraud = main_df[main_df['target']==abnormal_class]
Valid = main_df[main_df['target']==normal_class]
outlier_fraction = len(Fraud)/float(len(Valid))
print(f"Outlier Score {outlier_fraction}")
>> Outlier Score 0.12234261418186805
```

Listing 3.9: GridsearchCV and parameters for both models get_params attribute

```
n_estimators = [50,20, 100,150,200, 300, 500]
max_samples = [1,2,5,10,15,12,20,25,30,40,100, 50, 1637, 2000,
len(train_pca)]
# Tuning Random forest
tuned_rf= {'n_estimators':n_estimators,'max_samples':max_samples
          ,'verbose':[1]}
random_forest = GridSearchCV(RandomForestClassifier(
         random_state=42),param_grid=tuned_rf
         ,scoring="roc_auc",return_train_score=True)
rf_model = random_forest.fit(train_pca, train_y)
# Tuning Isolation forest
tuned_iso= {'n_estimators':n_estimators,'max_samples':max_samples
          ,'contamination':[ 0.12,],'verbose':[1]}
isolation_forest = GridSearchCV(IsolationForest(random_state=42)
             ,param_grid=tuned_iso,scoring="roc_auc"
             ,return_train_score=True,)
iso_model = isolation_forest.fit(train_pca, train_y)
```

to when fitting the model to define the threshold on the scores of the samples. In Listing 3.8 a calculation is done on the entire research dataset, before splitting it to train and test, to calculate and store the outlier score for the entire research dataset. This number was used throughout the training of the Isolation forest model and its iterations. The number was rounded down to 0.12 and stored as a variable outlier_fraction and passed as an parameter to the Isolation Forest model.

Once these parameters were defined and set, the parameters and tree models were applied to a GridsearchCV model to score the best parameters. The GridsearchCV model is scored with an auc_score and training scores returned. This step allowed to tuned the two tree models to the parameters with the highest AUC scores with the training data fit to the models.

3.8 Summary

This chapter described the steps and setup undertaken to ensure the prudent analysis of malware and security incidents. The steps taken to protect the machine and Internet from malware analysis, the data gathered as well as the analysis specifications of the machine used to conduct the training and performance evaluation of the models were documented. Care was taken to keep these environments separate and to create a realistic environment to ensure the execution of the malware in realistic conditions. In further sections the datasets of the LID-DS and VirusTotal sources were discussed, together with their strengths and relevance to the research. Later sections focused on the processing and enriching of data before vectorizing and selecting parameters for analysis. Finally the models themselves were described as well as the training and testing that was undertaken.

Chapter 4

Model Implementation and Results

In this chapter, the analysis of data gathered from various anomaly detection experiments, is discussed in detail. The experiments identified interesting trends and behaviours dependent on the specific models and parameters, which each formed a separate experiment. Two groups of experiments were conducted: running the model with all parameters as default or parameter tuning and applying the tuned parameters to the model. The two groups of experiments were applied to the two tree models (Random Forest and Isolation Forest), which for the sake of ease of understanding are discussed as four separate experiments. The resulting graphs, results and metrics were exported from *Jupyter* notebook's HTML export and embedded in the results analysis, for the visualization and description of the accuracy and scores measured during each experiment.

The first section explains the four experiments conducted for this research. The next four sections focus on the metrics and results gathered in each of the four experiments. The second last section compares the results and performance of all the experiments.

4.1 Experimental Setup

For the purpose of the research two different experiments were conducted. Experiment 1 is the anomaly detection of a supervised learning model, namely a Random Forest tree model, with the research dataset with a train and test subset split. To contrast and measure the effectiveness of a unsupervised model compared to a supervised model for anomaly detection, the same research dataset, with a train and test subset split, was applied to an Isolation Forest tree model, referred to as experiment 2. This forms the basis of the experimental work to provide baseline data.

The results from Experiments 1 and 2 lead to a further hypothesis of the effectiveness of changing the parameters of the models. This lead to changing the `max_sampling` and `n_estimators` of both models, as these were parameters they both shared. To test the impact and effectiveness on anomaly detection, a further two experiments were derived from the initial two. The experiments were derived from parameters for `max_samples`

and `n_estimators` and applied to both models, using the same data as in Experiments 1 and 2. Experiments 3 and 4, derived from Experiments 1 and 2 respectively, focused on the impact of tuning the tree parameters that are shared by both models.

All the experiments conducted are listed below and described in the subsequent sections in this order:

- Experiment 1: *Random Forest* anomaly detection performance with the research dataset.
- Experiment 2: *Isolation Forest* anomaly detection performance with the research dataset.
- Experiment 3: Test Random Forest with the most optimal n_estimators and max_samples with the research dataset.
- Experiment 4: Test Isolation Forest with the most optimal n_estimators and max_samples with the research dataset.

As the research compared two different decision tree models to the research dataset with different execution parameters, a compare_models function was implemented to reduce code and mistakes when comparing.

As illustrated in Appendix A.2, the compare_models function takes three arguments in the form of a PCA, a Y axis or target value and a list of classifiers to fit the data and measure accuracy and metrics. The function returns a heat-map as well as the classifiers after printing metrics and accuracy scores. The classifiers are a *RandomForestClassifier* and *IsolationForest* classifier from the sci-kit learn library instantiated with default or specific arguments.

Experiments 1 and 2 used the default parameter instantiated Random Forest and Isolation Forest classifiers, respectively as well as the training split PCA and training target or y, as illustrated in Appendix A.3. No further hyper-parameter tuning was applied to these classifiers.

In contrast Experiments 3 and 4 first applied the same two classifiers from the previous experiments to the sci-kit learn *GridsearchCV* class and fit the training data as shown in Appendix A.4. Unlike the previous experiments the first step was to search the models for their best performing parameters, which then would be applied to new instances of

these classifiers. Once the classifiers had the hyper-parameter tuning completed, the same compare_models function was applied to the classifiers with the dataset.

Links to the full source for the code and experiments is found in Appendix A.1. All these experiments were run in sequence and produced results to be discussed in the following section.

4.2 Model Evaluation Metrics

To evaluate and score the performance of the two models in the four experiments, the following metrics were used:

- **True Positive (TP)** Correctly predicted positive values, where the value of the actual label and predicted label are both positive.
- **True Negative (TN)** Correctly predicted negative values, where the value of the actual label and predicted label are both negative.
- **False Positives (FP)** Incorrectly predicted positive values, where the actual label is negative, but the prediction is positive.
- **False negatives (FN)** Incorrectly predicted negative values, where the actual label is positive, but the prediction is negative.
- **Precision** The ratio of the correctly predicted positive observations to the total predicted observations.
- **Recall** The ratio of the correctly predicted positive observations to all the observations in the actual class.
- **f1 score** The weighted average of precision and recall. The score takes false positives and false negatives into account.
- **Execution time** Time taken to train the model.
- **Confusion Matrix** An error matrix to evaluate model performance as seen in Table 4.1.
- **Area under the curve (AUC)** A measurement of the quality of the outlier detection by computing the area under a receiver operating characteristic (ROC) curve. The ROC curve is a plot of the true positive rate against the false positive rate.

Figure 4.1 shows an example ROC and AUC plot. This measurement or score is essential in measuring anomaly detection, as an accuracy score of 0.95 with an AUC score of 0.5, would indicate the effectiveness of the model equivalent to that of a coin toss.

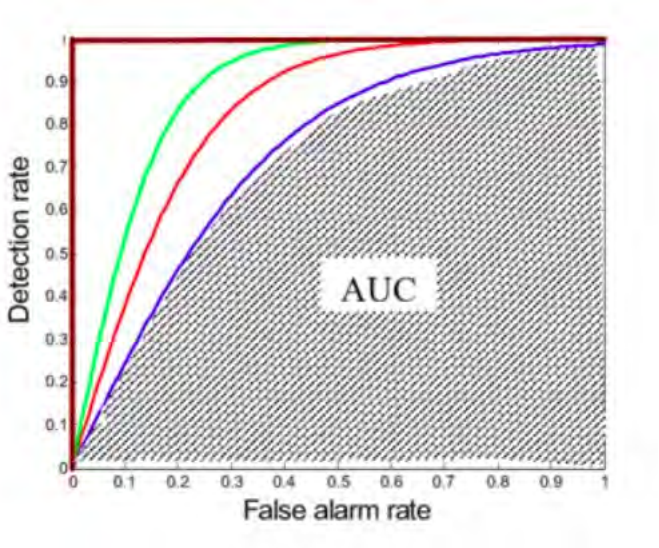

Figure 4.1: ROC curve with AUC taken from Latecki *et al.* (2007)

The following equations define the precision, recall and F1 score metrics:

$$Accuracy = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

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

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

$$F_1 Score = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

The above metrics were evaluated for both the Isolation Forest and Random Forest models. Both models were tested on the same dataset and thus form the four experimental results discussed next. Each experiment is discussed in terms of a confusion matrix, training time and AUC, as illustrated in a sample classification report and scores in Listing 4.1.

The *-1* and *1* values in the classification report in Listing 4.1 indicate an anomalous and normal classification, respectively. Both models in this research, produce binary classifications on documents as either anomalous or normal. In the context of the research,

Listing 4.1: Example metrics from experiment output

```
Time model [Isolation Forest] was applied:  1.00
Model - Isolation Forest: Errors - 10
Classification Report :
              precision    recall  f1-score   support
          -1       0.10      0.10      0.10        10
           1       0.90      0.90      0.90        90
    accuracy                           0.95       100
   macro avg       0.86      0.89      0.87       100
weighted avg       0.95      0.95      0.95       100
Scores:Model accuracy:0.90, f1 accuracy:0.90, roc_auc_score:0.90
```

a -1 or anomalous classification means that the parameters within the document vectors are abnormal in its predictions. Detecting anomalous documents is the primary focus of the experiments, as opposed to normal or benign documents. These anomalous documents are classified as -1 or anomalies.

The confusion matrix in Table 4.1 is indicative of the confusion matrix derived in the experiment from the models. The target data was marked as -1 (anomalous) and 1 (normal), thus the confusion matrix identifies the anomalies as negative values, and the normal data as positives.

Table 4.1: Confusion Matrix of actual vs. predicted -1 or 1 target values

		Predicted -1 (Negative)	Predicted 1 (Positive)
Actual	-1 (Negative)	TN	FP
Actual	1 (Positive)	FN	TP

4.3 Experiment One

The first experiment tested the effectiveness of the supervised learning model with the default values initialized for the model class. The model was instantiated as a *RandomForestClassifier* with the default parameters set by the class as given in Listing 4.2.

Listing 4.2: Experiment 1: Random Forest model get_params

```
>>> rf = RandomForestClassifier()
>>> rf.get_params()
{'bootstrap':True,'ccp_alpha':0.0,'class_weight':None,
'criterion': 'gini','max_depth':None,'max_features':'auto',
'max_leaf_nodes':None, 'max_samples':None,
'min_impurity_decrease':0.0, 'min_impurity_split':None,
'min_samples_leaf':1, 'min_samples_split':2,
'min_weight_fraction_leaf':0.0,'n_estimators':100,
'n_jobs':None,'oob_score':False,'random_state':None,'verbose':0,
'warm_start':False}
```

Listing 4.3: Experiment 1: Random Forest performance on training dataset

```
Time model [Random Forest] was applied:  0.9289071559906006
Model - Random Forest: Errors - 0
Classification Report :
              precision    recall  f1-score   support
          -1       1.00      1.00      1.00       598
           1       1.00      1.00      1.00      4884
    accuracy                           1.00      5482
   macro avg       1.00      1.00      1.00      5482
weighted avg       1.00      1.00      1.00      5482
Scores:Model accuracy:1.00, f1 accuracy:1.00,roc_auc_score: 1.00
```

4.3.1 Training evaluation

The model trained on 5482 files within 0.92 seconds as observed in Listing 4.3. The training time is very efficient, but the accuracy of the experiment is measured as 1.00. This gives this model an accuracy of 100% for identifying samples, which is evidence of overfitting the model. The AUC score also matches the accuracy score of 100%.

Figure 4.2 and Table 4.2 illustrates the ability of the model to guess an anomaly classification, overfitting the model and the ROC graph with all points having an X axis value of *0.0*. This result is not favoured, as the overfitting of a model makes the model ineffective on unseen events and identifying unexpected anomalies. The model might not be able to identify samples that were not in the training dataset for this research.

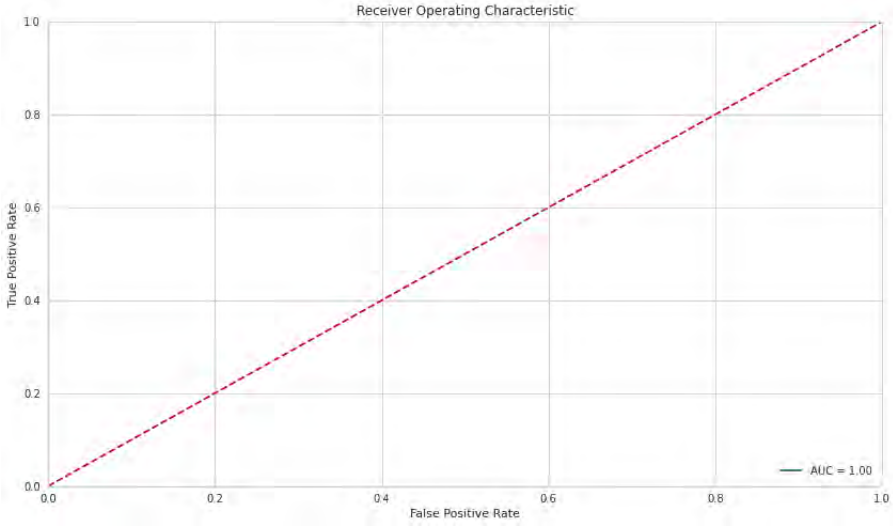

Figure 4.2: Experiment 1: ROC graph of training dataset

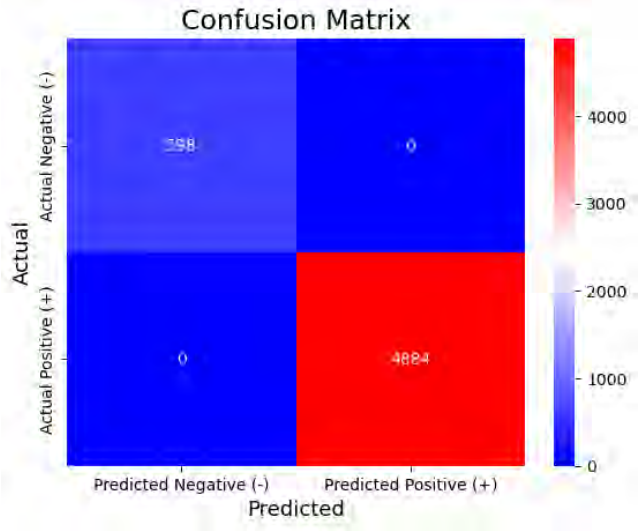

Figure 4.3: Experiment 1: Confusion matrix Random Forest with training dataset

The confusion matrix visualizes the model's training results with two zero values in two of the quadrants of Figure 4.3. This matches the model recording 0 errors in predicting the classifications. This matches the recall of the model which for both normal and anomaly classifications is 1.0.

Table 4.2: Confusion Matrix table of Experiment 1

	Total=5482	Predicted -1 (Negative)	Predicted 1 (Positive)
Actual	-1 (Negative)	TN = 598	FP = 0
Actual	1 (Positive)	FN = 0	TP = 4884

The results and scores from the Random Forest model in experiment 1, is not favourable for discovering zero day or other future anomalous executions, as the model is overfit. This is further illustrated in Figure 4.4, where actual target values are represented in blue and predicted anomalies in red. As we can see there is no difference in prediction and actual values, again indicating that the model is overfit.

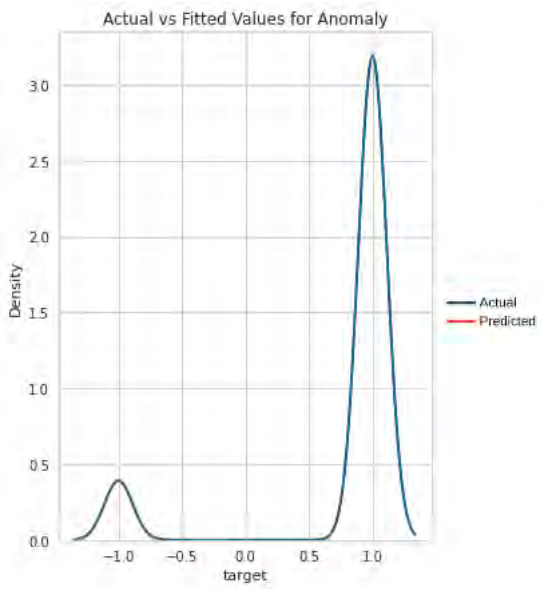

Figure 4.4: Experiment 1: Distribution plot of actual vs. predicted targets on training dataset

4.3.2 Evaluation of test dataset

To validate the model's accuracy and performance, a test dataset of 33% of the data was used to validate predictions. The distribution plot, in Figure 4.5 shows a slight increasing tendency to predict anomalies and normal values, but the accuracy and AUC scores for the test dataset still indicate a 100% result for both as seen in Listing 4.4. This further validates the overfitting result of training the Random Forest decision tree with the dataset and default values.

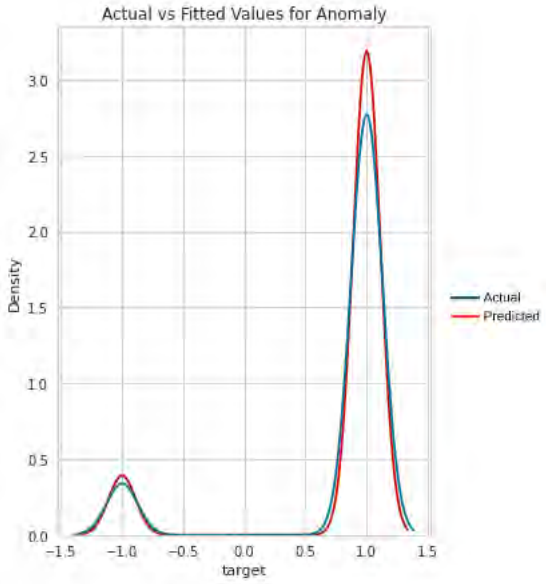

Figure 4.5: Experiment 1: Distribution plot of actual vs. predicted targets on test dataset

Listing 4.4: Experiment 1: Predictions and accuracy on test dataset

```
             Actual   Predicted
7958         -1           -1
4866          1            1
5489          1            1
2437          1            1
7942         -1           -1
Mean Absolute Error: 0.0
Mean Squared Error: 0.0
Root Mean Squared Error: 0.0
AUC score: 1.0, Accuracy: 100.0 %.
```

4.4 Experiment Two

The second experiment tested the effectiveness of the unsupervised learning model with the default values initialized for the model. The *IsolationForest* model is instantiated with the default parameters set by the class as given in Listing 4.5.

4.4.1 Training evaluation

The model trained within 0.38 seconds on 5482 files in the training dataset. As observed in Listing 4.6 the model's accuracy was measured at 0.9131 or 91.31%. The model was fit with only 476 errors with a high precision for identifying normal data, as well as a very high recall for anomalous data and an f1 accuracy score of 95.26%.

Listing 4.5: Experiment 2: Isolation Forest model `get_params`

```
>>> iso = IsolationForest()
>>> iso.get_params()
{'bootstrap': False, 'contamination': 'auto', 'max_features': 1.0,
 'max_samples': 'auto', 'n_estimators': 100, 'n_jobs': None,
 'random_state': None, 'verbose': 0, 'warm_start': False}
```

Listing 4.6: Experiment 2: Isolation Forest performance on training dataset

```
Time model [Isolation Forest] was applied:  0.3820769786834717
Model - Isolation Forest: Errors - 476
Classification Report :
              precision    recall  f1-score   support
          -1       0.69      0.36      0.48       598
           1       0.93      0.98      0.95      4884
    accuracy                           0.91      5482
   macro avg       0.81      0.67      0.72      5482
weighted avg       0.90      0.91      0.90      5482
Scores:Model accuracy: 0.9132, f1 accuracy: 0.9526,
roc_auc_score: 0.6724
```

The recall scores for the model indicate a 98% score on recalling normal data points, but the model is only able to identify 35% of anomalous data points. This is further evident with the AUC score indicating an issue with the accuracy of the model as a ROC AUC score of 67.24% indicates only being 17% more accurate than a 50/50 guess. As evident in Figure 4.6 the AUC curve has a true positive rate of 67%.

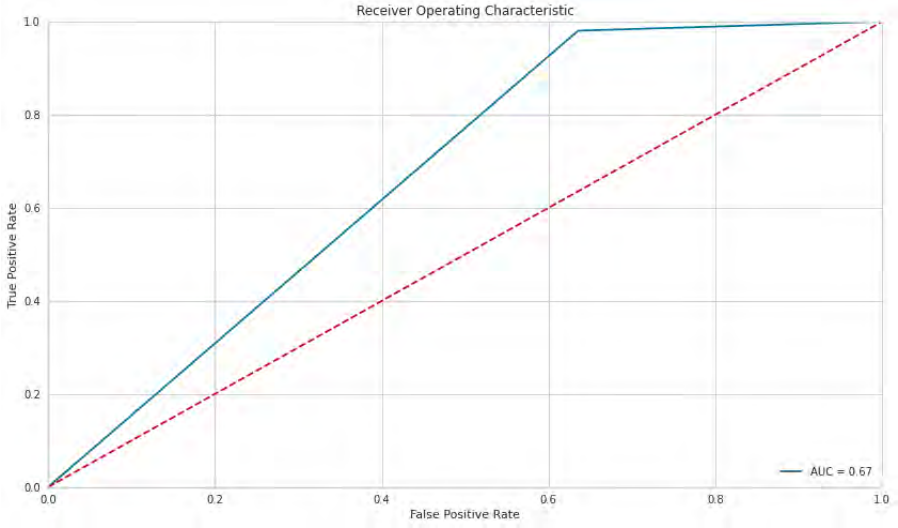

Figure 4.6: Experiment 2: ROC graph of training dataset

The confusion matrix in Figure 4.7 and Table 4.3, indicates a high false positive rate, indicating that the model classified a large number of normal data points as anomalous, yet a much smaller false negative classification. The experiment aims to accurately identify anomalous data points to identify abnormal system calls indicating a security incident, thus a large number of normal data points predicted as anomalous is not conducive to the success of the research.

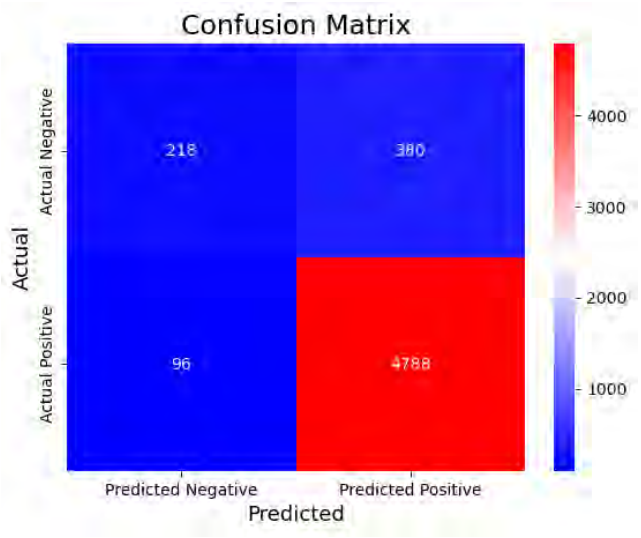

Figure 4.7: Experiment 2: Confusion matrix Isolation Forest with training dataset

Table 4.3: Confusion Matrix table of Experiment 2

	Total=5482	Predicted -1 (Negative)	Predicted 1 (Positive)
Actual	-1 (Negative)	TN = 218	FP = 380
Actual	1 (Positive)	FN = 96	TP = 4788

Referring to the distribution plot of actual vs. predicted targets in Figure 4.8, the model tends to identify more anomalous data points than the actual target values, thus misidentifying normal data points in a larger quantity. This might indicate the model being overfit for identifying normal data values, but less capable of identifying abnormal or anomalous data points.

4.4.2 Evaluation of test dataset

The test dataset consisting of 33% of the data was validated against the trained model for Experiment 2 with mixed results. Validating with the test data we observe in Listing

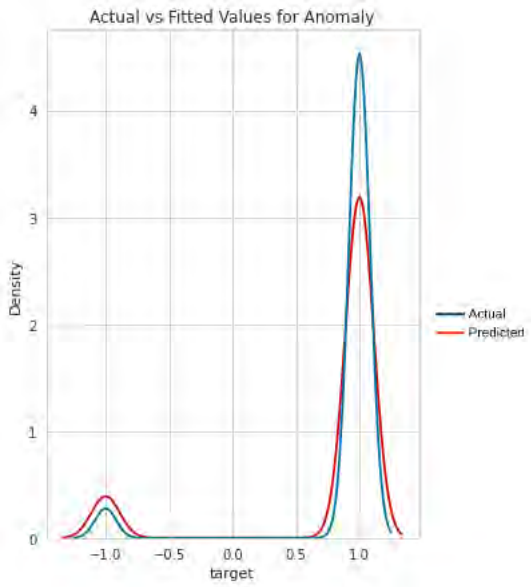

Figure 4.8: Experiment 2: Distribution plot of actual vs. predicted targets on training dataset

Listing 4.7: Experiment 2: Predictions and accuracy on test dataset

```
        Actual  Predicted
7958      -1          1
4866       1          1
5489       1          1
2437       1          1
7942      -1          1
Mean Absolute Error: 0.173659248449471
Mean Squared Error: 0.347318496898942
Root Mean Squared Error: 0.5893373370990014
AUC score: 0.6724462376636289, Accuracy: 99.83 %.
```

4.7 an increase in model accuracy of 8%, but no increase in AUC score. The 67% AUC score and 99% model accuracy is compared to the distribution plot in Figure 4.9. We observe that predicting anomalies has similar characteristics to the training dataset, but that predicting normal data points is increasingly more accurate according to the test validation.

The combination of the accuracy and efficiency score as well as the distribution plot indicates the effectiveness of the Isolation Forest's isolation of anomalies, and its ability to build decision trees unsupervised. This model is effective in its default state without hyper-parameter optimization, but additional hyper-parameter tuning is required to increase the AUC score of the algorithm.

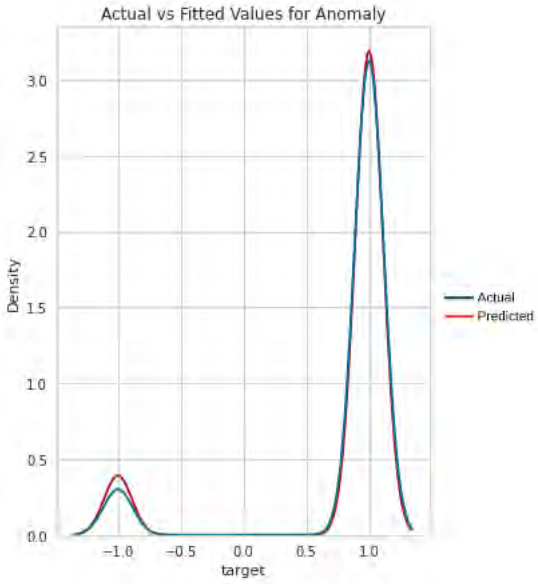

Figure 4.9: Experiment 2: Distribution plot of actual vs. predicted targets on test dataset

4.5 Experiment Three

To prevent the overfitting of the Random Forest model for use with the dataset, as Experiment 1's high score (high variance) indicates the potential for low bias, a parameter optimization was applied. The Gridsearchcv class, from scikitlearn, was applied to the training dataset. The Gridsearchcv class was fit with inputs for the `n_estimators` and `max_samples` as seen in Listing 4.8, which affect the fitting and size of the tree. These parameters were chosen as they exist in both tree models, and were fit with the RandomForestClassifier.

4.5.1 Training evaluation

The result of the Gridsearchcv produced the optimal parameters of `max_samples` 1637 and `n_estimators` of 50 for the Random Forest model. These parameters were applied to the model and the training and test data were fit again to evaluate the effectiveness of this technique.

As observed in Listing 4.9 the parameters had no effect on either the accuracy or the AUC score of the experiment. Within 0.22 seconds the model achieved an accuracy of 100% and an AUC score of 100.0%.

Listing 4.8: Experiment 3: Gridsearchcv for Random Forest with f1 scoring

```
n_estimators = [50, 100,150,200, 300]
max_samples = [1,2,5, 10, 15 ,12, 20, 25, 30, 40,
                100, 50, 1637, 2000, len(train_pca)]
tuned_rf = {'n_estimators':n_estimators,
        'max_samples':max_samples, 'n_jobs':[-1],'verbose':[1] }
random_forest = GridSearchCV(
                 RandomForestClassifier(random_state=42),
        tuned_rf, scoring='roc_auc', return_train_score=True,)
rf_model = random_forest.fit(train_pca, train_y)
print(rf_model.best_estimator_)
RandomForestClassifier(max_samples=1637, n_estimators=50,
random_state=42,verbose=1)
```

Listing 4.9: Experiment 3: Random Forest performance with parameter tuning dataset

```
Time model [Random Forest] was applied:  0.22982239723205566
Model - Random Forest: Errors - 0
Classification Report :
              precision    recall  f1-score   support
          -1       1.00      1.00      1.00       598
           1       1.00      1.00      1.00      4884
    accuracy                           1.00      5482
   macro avg       1.00      1.00      1.00      5482
weighted avg       1.00      1.00      1.00      5482
Scores:Model accuracy:1.00, f1 accuracy:1.00, roc_auc_score:1.00
```

The first observation is a great speed increase in training compared with Experiment 1's results. The GridsearchCV method identifies the maximum metric scored during fitting the data with different hyper parameters, but would require manual and random fitting or limiting the GridsearchCV method to a maximum score to prevent overfitting. The model is able to identify and classify normal and anomalous data points, but the overfit model might not be able to identify unseen data points.

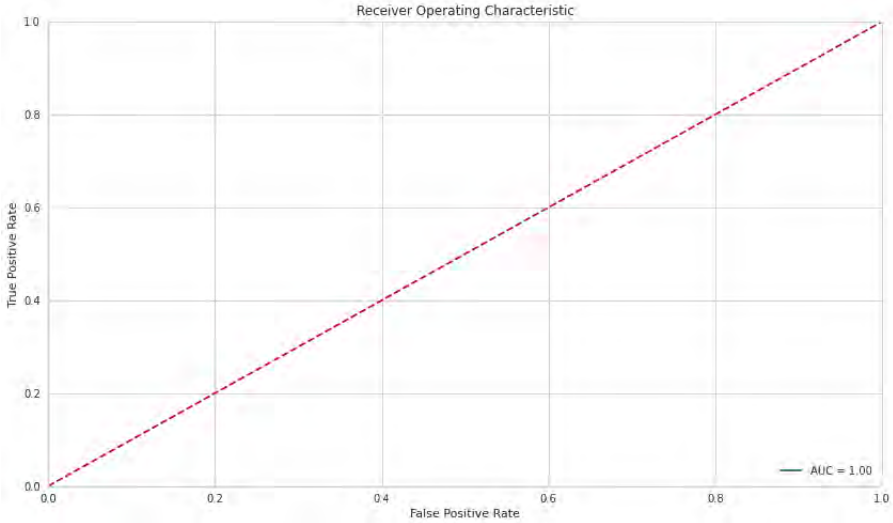

Figure 4.10: Experiment 3: ROC graph of training dataset

The ROC curve in Figure 4.10 is identical to that in Experiment 1 and the confusion matrix in Figure 4.11 and Table 4.4 shows another identical result to that in Experiment 1. Moreover Figure 4.12 is identical to the original default Random Forest model plot of actual vs. fitted values for classifying anomalies.

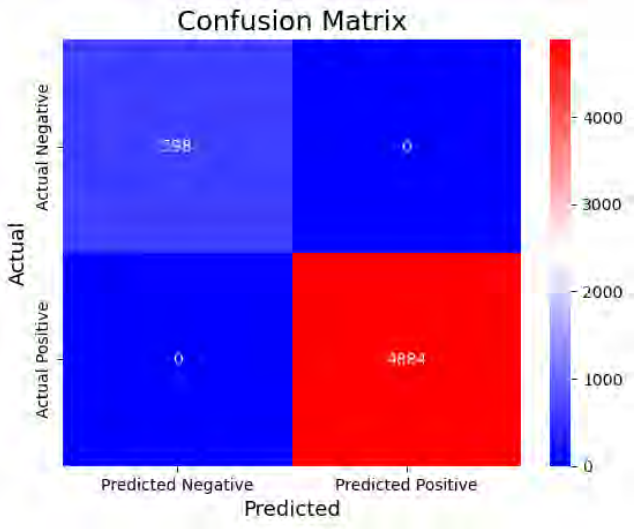

Figure 4.11: Experiment 3: Confusion matrix for Random Forest with training dataset

Table 4.4: Confusion Matrix table of Experiment 3

	Total=5482	Predicted -1 (Negative)	Predicted 1 (Positive)
Actual	-1 (Negative)	TN = 598	FP = 0
Actual	1 (Positive)	FN = 0	TP = 4884

Further manual fitting of the model could achieve a better result, but this falls out of the scope of these four experiments. As illustrated in Figure 4.12, a lack of red or actual target values indicate that the model has the same outcome for predictions and actual values. This indicates overfitting of the model, as the model would struggle with unseen data.

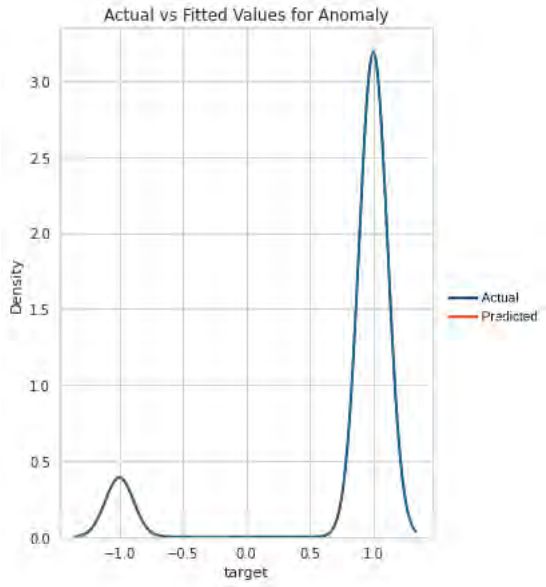

Figure 4.12: Experiment 3: Distribution plot of actual vs. predicted targets on training dataset

4.5.2 Evaluation of test dataset

Validating the overfit model in Experiment 3, the test dataset indicates a slight deviation in Figure 4.13, but according to the accuracy and AUC scores in Listing 4.10, during the predictions we observe the same scores as in the training phase. This validates the overfitting of the model, according to the two metrics. The combination of the training and test validation on the model indicates that this model is not well suited for this experiment, without further methods for hyper-parameter tuning.

Listing 4.10: Experiment 3: Predictions and accuracy on test dataset

```
      Actual  Predicted
7958      -1         -1
4866       1          1
5489       1          1
2437       1          1
7942      -1         -1
Mean Absolute Error: 0.0, Mean Squared Error: 0.0,
Root Mean Squared Error: 0.0
AUC score: 1.0, Accuracy: 100.0 %.
```

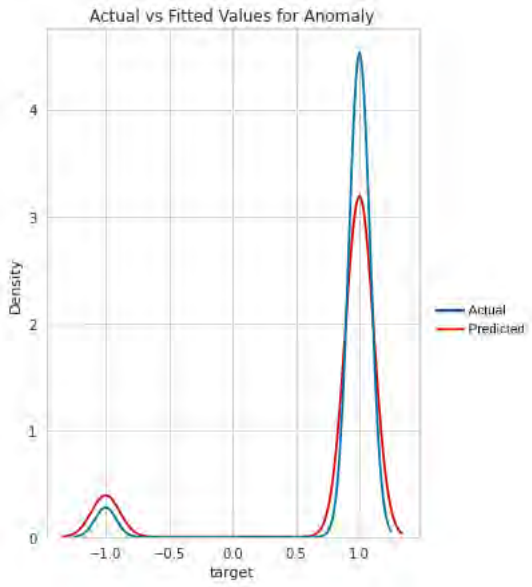

Figure 4.13: Experiment 3: Distribution plot of actual vs. predicted targets on test dataset

4.6 Experiment Four

Experiment 4 followed the same methodology as Experiment 3, except that hyper-parameter tuning was applied to the Isolation Forest model. As observed in Listing 4.11, the same parameters are applied to the Gridsearchcv class, as in Experiment 3. The `n_estimators` and `max_samples` parameters are shared between the two decision tree models, so are configured in the same way.

4.6.1 Training results

Fitting the Gridsearchcv of the training data on the Isolation Forest model resulted in the following parameters: 20 `max_samples` and 300 `n_estimators`. The `n_estimators` are

Listing 4.11: Experiment 4: Gridsearchcv for Isolation Forest with f1 scoring

```
n_estimators = [50, 100,150,200, 300]
max_samples = [1,2,5, 10, 15 ,12, 20, 25, 30, 40,
               100, 50, 1637, 2000, len(train_pca)]
tuned_iso = {'n_estimators':n_estimators,
'max_samples':max_samples,'n_jobs':[-1],'verbose':[1]}
isolation_forest = GridSearchCV(IsolationForest(random_state=42),
    tuned_iso, scoring='roc_auc', return_train_score=True)
model = isolation_forest.fit(train_pca, train_y)
print(iso_model.best_estimator_)
IsolationForest(contamination=0.12,
max_samples=20, n_estimators=300, random_state=42, verbose=1)
```

Listing 4.12: Experiment 4: Isolation Forest performance with parameter tuning and training dataset

```
Time model [Isolation Forest] was applied:  1.529893159866333
Model - Isolation Forest: Errors - 284
Classification Report :
              precision    recall  f1-score   support
          -1       0.74      0.81      0.77       598
           1       0.98      0.96      0.97      4884
    accuracy                           0.95      5482
   macro avg       0.86      0.89      0.87      5482
weighted avg       0.95      0.95      0.95      5482
Scores: Model accuracy:0.9482, f1 accuracy:0.9707,
roc_auc_score: 0.8887
```

identical to the Random Forest model in Experiment 3. These parameters were applied and the Isolation Forest model was executed again as shown in Listing 4.12.

The immediate results of the parameters applied in 1.52 seconds was an accuracy score of 94%, yielding an increase of 3%. The AUC score increased by 21.63% compared with the default parameter model in Experiment 2. The number of errors made by the model, both false positives and false negatives, are 284 out of 5482 and a f1 score of 97% was recorded.

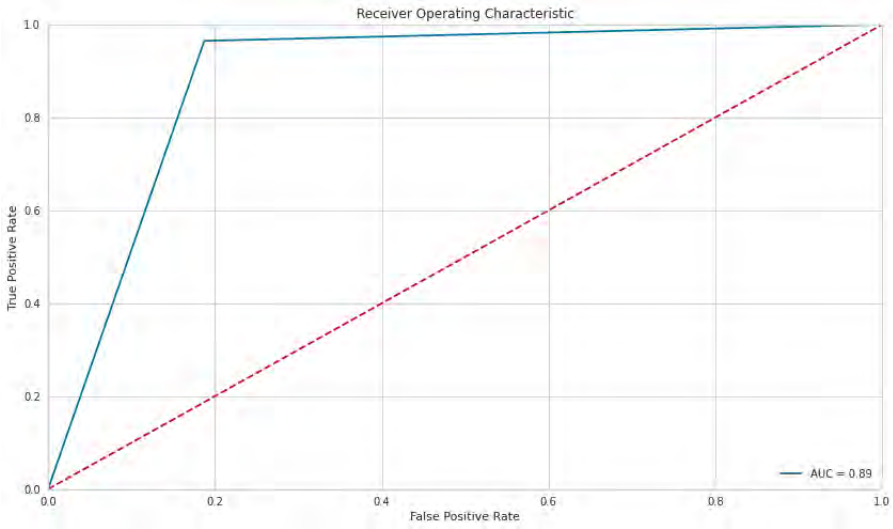

Figure 4.14: Experiment 4: ROC graph of training dataset

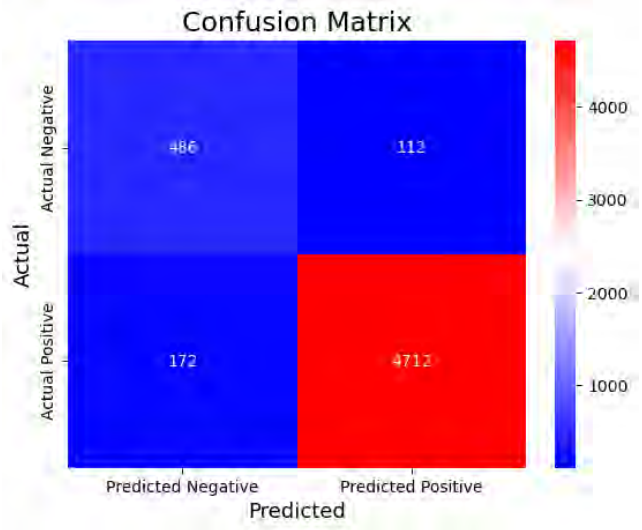

Figure 4.15: Experiment 4: Confusion matrix for Isolation Forest with training dataset

The ROC curve in the graph in Figure 4.14 varies greatly from that in Experiment 2, with the true positive rate starting below 0.2. The confusion matrix in Figure 4.15 and Table 4.5 shows a higher false negative rate than false positive rate, but observing the Figure in 4.16 we see that the actual vs predicted target values are very similar.

Table 4.5: Confusion Matrix table of Experiment 4

	Total=5482	Predicted -1 (Negative)	Predicted 1 (Positive)
Actual	-1 (Negative)	TN = 486	FP =112
Actual	1 (Positive)	FN = 172	TP = 4712

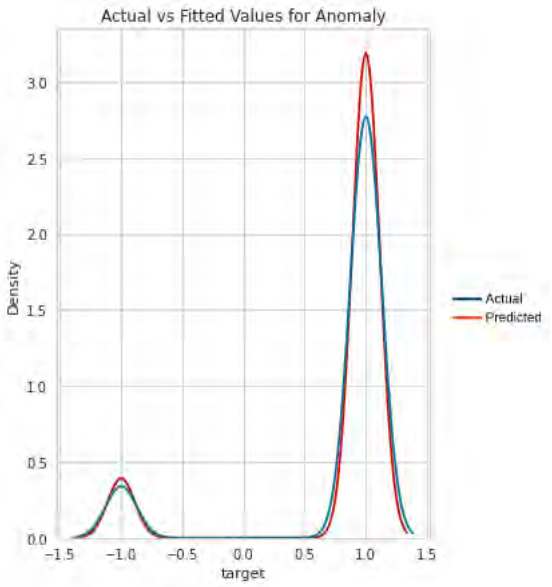

Figure 4.16: Experiment 4: Distribution plot of actual vs. predicted targets on training dataset

4.6.2 Evaluation of test dataset

Finally to validate the last experiment with the 33% test dataset we applied the model prediction method on the dataframe, with good results. The model accuracy as observed in Listing 4.13 reaches 99.79% combined with an AUC score of 94.01%. In this validation the accuracy score increased by 5% and the AUC score by 6%. To further confirm this observation, Figure 4.17 indicates a high correlation between actual and predicted target values. Thus the model is accurately identifying both normal and anomalous data points, without overfitting.

Listing 4.13: Experiment 4: Predictions and accuracy on test dataset

```
      Actual  Predicted
7958      -1         -1
4866       1          1
5489       1          1
2437       1          1
7942      -1         -1
Mean Absolute Error: 0.2134257570229843
Mean Squared Error: 0.4268515140459686
Root Mean Squared Error: 0.6533387437202608
AUC score: 0.9401105651105651, Accuracy: 99.79 %.
```

These train and test results for Experiment 4 are the best of all the experiments, excluding those results where the accuracy and AUC score reaching a 100% yielded on overfit result.

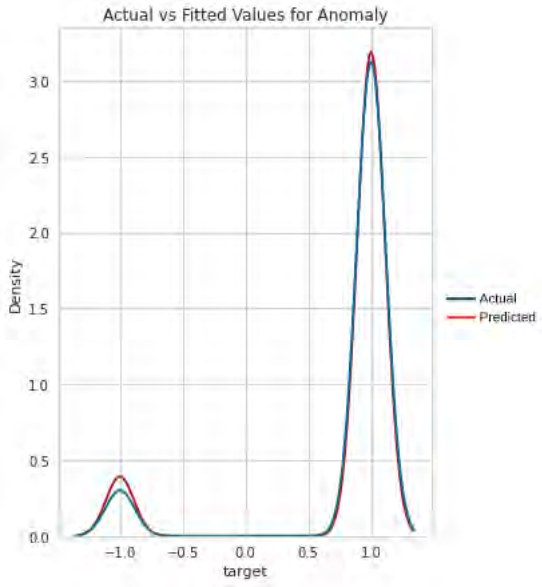

Figure 4.17: Experiment 4: Distribution plot of actual vs. predicted targets on test dataset

4.7 Comparison of Experimental Results

In this section we compare the results from the experimental model training and analysis of the Isolation Forest and Random Forest models on the train and test datasets. We found that the methods employed allowed for large datasets to be converted into numerical vectors and be fit and trained on trees within acceptable time frames. Analysis on the distribution of the dataset within the classes of 1 (normal) and -1 (abnormal) displayed

specific patterns and clusters for the data. All the experiments used the same dataset, split into a 67% training and 33% testing sample sets. The four models were compared to find the most effective model, as well as the impact of hyperparameter tuning on the models.

Further sections discussed model evaluation and performance in terms of True Positives, True Negatives, False Positives and False Negatives as well as measuring precision, recall, f1 and AUC scores. Further a confusion matrix with the other evaluation metrics showed that an Isolation Forest model identified both anomalies and normal data with high accuracy, while the Random Forest model was prone to overfitting or lower accuracy in some cases.

The Random Forest model used in Experiments 1 and 3, had a tendency in both experiments to be overfit and thus inadequate for detecting future unseen anomalies or normal samples. In Experiment 2 the Isolation Forest achieved a high level of accuracy, but the AUC score of 67% indicated that the model was not accurate enough for the classification problem.

The results show that an Isolation Forest model is accurate in detecting and classifying anomalies from normal documents, in reference to tf-idf representations of system call trace logs. More specifically Experiment 4 provides the highest accuracy, without overfitting, to identify previously seen and unseen anomalies within the system call log vectors. Experiment 4 has the highest training time of 1.52 seconds, but this is an acceptable result given the processing of 5482 log files.

The experiments conclusively proved that Experiment 4, the hyperparameter tuned Isolation Forest was the most accurate and effective model in detecting anomalies, without overfitting. This model was the unsupervised model and required no target values, although the target values were used to measure the effectiveness of predicted targets vs. actual target values. Experiment 4 validates the hypothesis that a tf-idf vector representation of system call log files can be used to train and accurately predict anomalies using an Isolation Forest.

Finally the experimental results indicated that the various tools could be applied with unsupervised models and techniques, allowing for the potential of wider adoption of these techniques outside of data science professionals and manually having to tag data.

4.8 Summary

This chapter described the experimental model implementation as well as the experimental results. The four experiments are described as variations of the Random Forest and Isolation Forest machine learning models. The experimental results are compared using the measurements of accuracy and AUC score to measure the true positive rate against the false positive rate, which is an indication of the effectiveness of the model on the data. The results indicate that the unsupervised Isolation Forest model is the most suitable for detecting security anomalies within the dataset. The experimental results additionally concluded that the Isolation forest model achieves a suitable accuracy for detecting anomalies in the dataset once hyper-parameter optimizations were applied.

Chapter 5

Conclusion

The primary goal of this research was to identify whether NLP with supervised or unsupervised machine learning could detect anomalies such as malware in system call trace logs. This primary goal was researched and explored with literature reviews, data gathering and experimentation. This chapter discusses the outcomes of this research through the discussion of the research hypothesis and research questions, stated previously. Thereafter a discussion is presented of the various limitations of the research, experiments and data gathered. Finally suggestions for future work in this area complete this chapter.

5.1 Contributions of Research

The hypothesis for this research was stated as follows: *Natural language processing techniques can be applied to logs such as syscall logs to classify and identify anomalies in previously undetected incidents or datasets.* To prove the hypothesis the seven research questions initially set are revisited.

Firstly the question around NLP being able to process and extract parameters from human readable logs such as system call traces has been experimentally proven. The use of techniques such as text pre-processing, *tf-idf*, PCA and visualization tools allow for extracting meaningful tokens from unstructured logs for statistical modelling. Natural language processing with methods such as *Bag-of-system* calls, provides researchers and security professionals with powerful tools to interrogate and detect anomalies in logs.

Secondly, the reduction of noise in the system and ease of use, is dependent on the training of the models and the type of model. An unsupervised model can create more noise in the system, but requires minimal inspection of the data source and can potentially identify previously unknown malware or vulnerabilities. Both models are able to identify anomalies in system call trace logs, but the unsupervised Isolation Forest model is best suited to detect anomalies.

The next question on *Can Natural Language processing detect malware from system call traces*, is answered through the results obtained from applying the trained classifiers on

the *VirusTotal* malware samples. The classifiers were able to identify abnormal log lines with high accuracy. As this is a limited experiment it would not account for all different families of malware and is focused on x86_64 GNU Linux ELF binaries, but the classifiers were able to identify malware system call traces mixed in with other documents of normal and abnormal traces.

On the fourth question of whether supervised or unsupervised models produce more accurate results, the answer depends on the metrics used. Using the classification report tools, accuracy score, AUC score and f1 score, the unsupervised *Isolation Forest* is best suited to the problem statement and dataset. The supervised *Random Forest classifier*, is prone to over-fitting and thus would not be able to identify unseen or zero day events. The *Isolation Forest classifier* model is well suited as its unsupervised nature allows for easy training and identifying potential patterns a data analyst or scientist is unaware of, which could identify unknown or zero day events.

On the fifth question of whether unsupervised learning models are able to detect previously unseen incidents, the answer is yes. The Isolation Forest model was able to detect anomalies in files from the *VirusTotal* dataset and *LID-DS* dataset, which it had not been trained on. The test validation split of 33% confirmed that the model was not under- or over-fit, thus suited to the hypothesis and problem. The detection varied based on the file and type of malware. Each malware would do abnormal or more dangerous system calls in different sequences, leading to different tf-idf scores. The *PCA* method applied was able to identify the most important features within the documents, ignoring normal system call events that occurred frequently within malicious execution. Thus the models were trained efficiently and focused on the most important indicators of abnormal or normal behaviour.

The second last question is focused on whether either the *Isolation Forest*, *Random Forest* or both models should be used to detect anomalies. The answer to this question is the Isolation Forest classifier algorithm. Additional hyper-parameter tuning would be required for the Random Forest model, which falls out of the scope of the thesis. Considering the performance in training time and accuracy being favourable in both, the Isolation Forest is fast and accurate with the added benefit of being an unsupervised decision tree. The Isolation Forest algorithm is less prone to overfitting compared to the Random Forest algorithm, and does not require tagging all of the data.

Finally the question is asked whether NLP and anomaly detection methods are accessible by information security professionals, outside of the realm of trained machine learning

engineer or data scientists? With the application of the tf-idf vectorizer, PCA and Isolation Forest the research indicates that non data scientists would be able to apply the tools on logs to detect and investigate logs. These tools do not replace human intervention as training and intervention would be necessary to continuously use the tools, but could add additional capabilities to teams. The unsupervised nature of the tools allows information security professionals to build tools and filter log noise in large distributed systems. Also the training time and vectorizing is low while, allowing for the burst usage of cloud resources for training, and end point devices to do inference/prediction.

This research has highlighted the strengths and possibilities of machine learning combined with NLP tools to apply human speech algorithms and techniques to computer generated human-readable logs. Techniques like tf-idf, dimension reduction, pre-processing and PCA allow for large unstructured data to be vectorized and mapped to models, for efficient and accurate anomaly detection.

5.2 Limitations

The limitations of this research can be considered as three main issues namely: CPU based libraries, availability of quality system call logs in unstructured text format and realistic malware execution without Internet access.

The first limitation is simple in the fact that the sklearn and *yellowbrick* libraries are not optimized to run on graphics cards, which are able to do many more parallel computations. The libraries used are simplified for usage with existing computing, and can use every processor available on the host, but to achieve adequate speed for the training and analysis of the models expensive cloud resources were rented, namely, an Amazon Web Services virtual machine with 64 vCPU cores and 256.0 GiB of RAM. Most of the time spent processing the data was bottle necked by Python's single threaded nature on text-preprocessing, though *pandas* and *numpy* are fairly efficient. But as the data sources grow, so the processing time and needed resources would have to increase. The size of the selected machine is not critical to use this model, but was used to speed up the experimentation for this research.

The second limitation is the dataset. There are two potential limitations with the dataset. One is that it was formulated and tested for this research. The dataset should be validated by other researchers as well, or be shared open source for further validation. Furthermore the number of system call sysdig log traces, more than 8000, might not be a sufficiently

large dataset. Further data should be gathered and applied to the corpus, dataset and model.

The third limitation is centered around the work by Rossow *et al.* (2012) for creating safe and realistic malware experiments. Executing malware with a simulated connection to the Internet and in a virtual machine, can affect the execution of the malware, if errors occur or the malware has mitigations to identify the environment. Some of the malware traces could only have captured some of the logic of the malware, as some of the rootkits and trojans would download further payloads for execution, and would execute these payloads.

Finally the difficulty around datasets in raw systrace or sysdig format are lacking for academic or shared open source access. The Grimmer *et al.* (2019) dataset provides a baseline and common framework for gathering traces, but more academics and institutions would need to share both simulated and traced real world anomalies. System call tracing and storage of raw unstructured human readable logs is not a common practice, as the computation, complexity and storage costs are high. Other datasets contain a low dimensionality and lack context of the system call arguments. Further capturing of traces and running live samples are needed to be published to further this research.

5.3 Future Work

Through some of the challenges of the research some limitations were identified, but also future work and improvements. Areas that could strengthen the research are discussed in the following subsections.

5.3.1 Manual and more advanced pre-processing of text

The tf-idf vectorizer has built in text pre-processing and tokenization, which was applied to the input data for generating the dataset. This was inline with the hypothesis to allow non-machine learning information security practitioners to use and understand these tools. The context of the input data and dimensionality and vocabulary can affect the outcome of the model, thus pre-processing and removing some of the data could hypothetically improve the model.

An example of this improvement is using encoders to change numbers to text representations such as 1 to one. This would effectively turn IP addresses into word representations for the model to map and fit. As IP addresses or domain names could indicate data ex

filtration or an injection attack, this would provide context for the model to identify and classify this kind of anomaly.

Many other symbols, punctuation or numerical values could be encoded or tokenized to preserve the context of potential anomalous behaviour.

5.3.2 Microsoft Windows and ARM based system call logs

For this research, focus was placed on Linux system calls for x86_64 only. Future research could be done on training models on the ARM, MIPS and other architectures as well as on the Microsoft Windows, macOS and BSD system call tables. Further research could also go in two directions: translation or fitting multiple architectures to the same tf-idf.

Firstly a translation layer for Microsoft Windows, ARM, x86 or other logs to a common format of exchange between systems would be required to translate different system call traces and their system calls to a sysdig or more generic format. Once a common language or format is established, it could be used to train a system agnostic model and measure the performance of a generic model.

Research on Microsoft Windows malware would be crucial to enhance techniques presented in this thesis, as Microsoft Windows malware is the most prevalent type of malware. The larger number of potential samples and diversity in the malware families could enable training a powerful classifier.

The second direction for future research is fitting a higher dimensionality vocabulary of ARM, Microsoft Windows, Linux, BSD and other operating systems and multiple CPU architectures to the tf-idf and another Isolation Forest model. With a large enough vocabulary of system calls and arguments, a classifier could be trained with a PCA that optimizes for the variance in the data on all possible iterations of a system call. With enough examples and enough data, an unsupervised learning model could hypothetically detect any type of anomaly on any type of system. The inference resources needed for this classifier would be low, but training on multiple architectures and operating systems would require a lot of training resources.

References

Abed, A. S., Clancy, C., and Levy, D. S. Intrusion detection system for applications using Linux containers. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9331, pages 123–135. 2015. doi:10.1007/978-3-319-24858-5_8.

Aleroud, A. and Karabatis, G. A contextual anomaly detection approach to discover zero-day attacks. *Proceedings of the 2012 ASE International Conference on Cyber Security, CyberSecurity 2012*, (May):40–45, 2013. doi:10.1109/CyberSecurity.2012.12.

Anderson, J. A comparison of Unix sandboxing techniques: The case for sandboxing. Technical Report Sep/Oct, FreeBSD Journal, 2017.

Anello, E. Anomaly detection with isolation forest. `https://betterprogramming.pub/anomaly-detection-with-isolation-forest-e41f1f55cc6`, 4 2021. (Accessed on 11/03/2021).

Anton, S. D., Sinha, S., and Dieter Schotten, H. Anomaly-based intrusion detection in industrial data with SVM and random forests. pages 1–6. 2019. doi:10.23919/SOFTCOM.2019.8903672.

Bagherzadeh, M., Kahani, N., Bezemer, C.-P., Hassan, A. E., Dingel, J., and Cordy, J. R. Analyzing a decade of Linux system calls. *Empirical Softw. Engg.*, 23(3):1519–1551, jun 2018. doi:10.1007/s10664-017-9551-z.

Bellavia, A. Statistical Methods for Environmental Mixtures. Self-published, 2021. URL `https://bookdown.org/andreabellavia/mixtures/`

Bengfort, B., Bilbro, R., and Ojeda, T. Applied Text Analysis with Python: Enabling Language-Aware Data Products with Machine Learning. O'Reilly Media, Inc., 1st edition, 2018.

Besomi, J. Getting started - Texthero from zero to hero. 2021. (Accessed on 2021/08/15). URL `https://texthero.org/docs/getting-started`

Bhatt, M. and Ahmed, I. Leveraging relocations in ELF-binaries for Linux kernel version identification. *Digital Investigation*, 26:S12–S20, 2018.

Boelen, M. The 101 of ELF files on Linux: Understanding and analysis. 2019. (Accessed on 2021-06-20).
URL `https://linux-audit.com/elf-binaries-on-linux-understanding-and-analysis/`

Boyle, T. Hyperparameter tuning in Python | towards data science. 2019. (Accessed on 11/02/2021).
URL `https://towardsdatascience.com/hyperparameter-tuning-c5619e7e6624`

Chandola, V., Banerjee, A., and Kumar, V. Survey of Anomaly Detection. *ACM Computing Survey (CSUR)*, 41(3):1–72, 2009.

Cozzi, E., Graziano, M., Fratantonio, Y., and Balzarotti, D. Understanding Linux malware. Technical report, IEEE Security & Privacy, 2018.

Degioanni, L. Announcing Sysdig: a system exploration tool. 2014a. (Accessed on 2021/07/01).
URL `https://sysdig.com/blog/announcing-sysdig/`

Degioanni, L. Interpreting Sysdig Output. 2014b. (Accessed on 2021/07/01).
URL `https://sysdig.com/blog/interpreting-sysdig-output/`

Denning, D. E. An intrusion-detection model. In *1986 IEEE Symposium on Security and Privacy*, pages 118–131. 1986. doi:10.1109/SP.1986.10010.

Developers.virustotal.com. Virustotal API v3 overview. 2021. (Accessed on 2021/08/01).
URL `https://developers.virustotal.com/v3.0/reference#analyses-object`

Ding, W. M., Ghansah, B., and Wu, Y. Y. Research on the Virtualization Technology in Cloud Computing Environment. *International Journal of Engineering Research in Africa*, 21(December 2015):191–196, 2015. doi:10.4028/www.scientific.net/JERA.21.191.

Elgeldawi, E., Sayed, A., Galal, A. R., and Zaki, A. M. Informatics hyperparameter tuning for machine learning algorithms used for arabic sentiment analysis. *Informatics*, pages 1–21, 2021.

Fruhlinger, J. The CIA triad: Definition, components and examples | CSO Online. 2020. (Accessed on 2021/02/05).
URL `https://www.csoonline.com/article/3519908/the-cia-triad-definition-components-and-examples.html`

Ganesan, K. All you need to know about text preprocessing for NLP and machine learning. 2019. (Accessed on 2021/08/15).
URL `https://www.kdnuggets.com/2019/04/text-preprocessing-nlp-machine-learning.html`

Garje, R., Bharati, S., Kar, P., and Khatavkar, V. Anomaly detection using context-based intrusion detection system. *International Journal of Computer Science and Information Technologies*, pages 3817–3822, 2014.

Goldstein, M. and Uchida, S. A Comparative Evaluation of Unsupervised Anomaly Detection Algorithms for Multivariate Data. *PLoS ONE*, pages 78658–78700, 2016. doi:10.1371/journal.pone.0152173.

Googlesource.com. Linux System call table. 2021. (Accessed on 29/09/2021).
URL `https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit`

Grimmer, M. and Max, M. Leipzig intrusion detection data set. 2019. (Accessed on 2021/09/29).
URL `https://www.exploids.de/lid-ds`

Grimmer, M., Röhling, M. M., Kreusel, D., and Ganz, S. A Modern and Sophisticated Host Based Intrusion Detection Data Set. *Proceedings 16. Deutscher IT-Sicherheitskongress*, page 10, 2019.

Grimmer, M., Röhling, M. M., Kricke, M., Franczyk, B., and Rahm, E. Intrusion Detection on System Call Graphs. *Proceedings - 25. DFN-Konferenz "Sicherheit in vernetzten Systemen"*, pages 1–18, 2018.

Herley, C. and Oorschot, P. C. SoK: Science, Security and the Elusive Goal of Security as a Scientific Pursuit. *Proceedings - IEEE Symposium on Security and Privacy*, pages 99–120, 2017. doi:10.1109/SP.2017.38.

Hoopes, J. Chapter 13 - Protection in untrusted environments. In **Hoopes, J.**, editor, *Virtualization for Security*, pages 301–328. Syngress, Boston, 2009. doi:https://doi.org/10.1016/B978-1-59749-305-5.00013-X.

Ioannidou, P. Anomaly Detection in Computer Networks. Ph.D. thesis, KTH Royal Institute Of Technology, 2021.

Janczuk, T. Sandboxing Code in the Era of Containers. 2015. (Accessed on 01-11-2020).
URL `https://medium.com/aws-activate-startup-blog/sandboxing-code-in-the-era-of-containers-294edb3a674`

Jolliffe, I. T. and Cadima, J. Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 374(2065):1–16, 2016. doi:10.1098/rsta.2015.0202.

Jurafsky, D. and Martin, J. H. Speech and Language Processing (2nd Edition). Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009.

Kang, D. K., Fuller, D., and Honavar, V. Learning classifiers for misuse detection using a bag of system calls representation. *Lecture Notes in Computer Science*, 3495:511–516, 2005. doi:10.1007/11427995_51.

Kibble, R. Introduction to natural language processing: Undergraduate study in Computing and related programmes. *Roeper Review*, 1(2):26, 2013.

Kovacs, E. Dyre banking trojan counts processor cores to detect sandboxes. 2015. (Accessed on 2021-08-01).
URL `https://www.securityweek.com/dyre-banking-trojan-counts-processor-cores-detect-sandboxes`

Latecki, L. J., Lazarevic, A., and Pokrajac, D. Outlier detection with kernel density functions. In **Perner, P.**, editor, *Machine Learning and Data Mining in Pattern Recognition*, pages 61–75. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-73499-4.

Li, T., Convertino, V. T. G., Wang, C. W., Most, C. H., Zajonc, C. T., and Tsai, C. Y.-H. Hypertuner: Visual analytics for hyperparameter tuning by professionals. 2018.
URL `https://bayesopt.github.io/`

Lippmann, R., Haines, J. W., Fried, D. J., Korba, J., and Das, K. Analysis and results of the 1999 DARPA off-line intrusion detection evaluation. In **Debar, H., Mé, L., and Wu, S. F.**, editors, *Recent Advances in Intrusion Detection*, pages 162–182. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

Liu, D., Caceres, M., Robichaux, T., Forte, D. V., Seagren, E. S., Ganger, D. L., Smith, B., Jayawickrama, W., Stokes, C., and Kanclirz, J. Chapter 3 - an introduction to cryptography. In *Next Generation SSH2 Implementation*, pages 41–64. Syngress, Burlington, 2009.

Liu, F. T., Ting, K. M., and Zhou, Z. H. Isolation forest. *Proceedings - IEEE International Conference on Data Mining, ICDM*, (October 2008):413–422, 2008. doi: 10.1109/ICDM.2008.17.

Liu, Z., Japkowicz, N., Wang, R., Cai, Y., Tang, D., and Cai, X. A statistical pattern based feature extraction method on system call traces for anomaly detection.

Information and Software Technology, 126:106348, 2020. doi:https://doi.org/10.1016/j.infsof.2020.106348.

Lourdusamy, R. and Abraham, S. A survey on text pre-processing techniques and tools. *International Journal of Computer Sciences and Engineering*, 06(03):148–157, 2018. doi:10.26438/ijcse/v6si3.148157.

Lundgren, B. and Möller, N. Defining information security. *Science and Engineering Ethics*, 25(2):419–441, 2019. doi:10.1007/s11948-017-9992-1.

Manning, C. D., Raghavan, P., and Schütze, H. Introduction to Information Retrieval. Cambridge University Press, Cambridge, UK, 2008.

McAfee. Whitepaper - The Growing Threat. Technical report, McAfee, 2006.

Mutz, D., Valeur, F., Vigna, G., and Kruegel, C. Anomalous system call detection. *ACM Transactions on Information and System Security*, 9(1):61–93, 2006. doi:10.1145/1127345.1127348.

Nassif, A. B., Talib, M. A., Nasir, Q., and Dakalbab, F. M. Machine learning for anomaly detection: A systematic review. *IEEE Access*, 9, 2021. doi:10.1109/ACCESS.2021.3083060.

Newman, L. H. Hacker lexicon: What is sinkholing? `https://www.wired.com/story/what-is-sinkholing/`, 02 2018. (Accessed on 11/04/2021).

O'Reilly, C., Gluhak, A., Imran, M., and Rajasegarar, S. Anomaly detection in wireless sensor networks in a non-stationary environment. *IEEE Communications Surveys & Tutorials*, 16:1413–1432, 2014.

Payam, R., Lei, T., Huan, L., and Tamer, Ö. M. Cross-Validation, pages 1–7. Springer New York, New York, NY, 2016.

Perkins, J. NLP for Log Analysis – Tokenization. 2018. (Accessed on 2021-08-05). URL `https://streamhacker.com/2018/11/13/nlp-log-analysis-tokenization/`

Primartha, R. and Tama, B. A. Anomaly detection using random forest: A performance revisited. In *2017 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6. 2017.

Rai, A. and Borah, S. Study of various methods for tokenization. In **Mandal, J. K., Mukhopadhyay, S., and Roy, A.**, editors, *Applications of Internet of Things*, pages 193–200. Springer Singapore, Singapore, 2021.

Rajagopalan, M., Hiltunen, M., Jim, T., and Schlichting, R. Authenticated system calls. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 358–367. 2005.

Rao, D. and McMahan, B. A Brief History of Time: From the Big Bang to Black Holes. O'Reilly Media, Inc, Sebastopol, CA 95472, 2019.

Ratazzi, E. P., Bommisetti, A., Ji, N., and Du, W. PINPOINT: efficient and effective resource isolation for mobile security and privacy. *CoRR*, abs/1901.07732, 2019.
URL `http://arxiv.org/abs/1901.07732`

Rea, A. and Rea, W. How many components should be retained from a multivariate time series pca. *arXiv*, pages 1–49, 2016.

Resende, P. A. A. and Drummond, A. C. A survey of random forest based methods for intrusion detection systems. *ACM Computing Surveys*, 51(3), 2018. doi:10.1145/3178582.

Rossow, C., Dietrich, C. J., Grier, C., Kreibich, C., Paxson, V., Pohlmann, N., Bos, H., and Van Steen, M. Prudent practices for designing malware experiments: Status quo and outlook. *Proceedings - IEEE Symposium on Security and Privacy*, pages 65–79, 2012. doi:10.1109/SP.2012.14.

Sabahi, F. Secure virtualization for cloud environment using hypervisor-based technology. *International Journal of Machine Learning and Computing*, pages 39–45, 2013. doi:10.7763/ijmlc.2012.v2.87.

Sánchez, A. C. and Aracil, J. Context-aware security framework based on traffic anomaly detection indicator. *Telecommunication Systems*, 65(2):319–330, 2017. doi:10.1007/s11235-016-0233-8.

Sarmah, A. Intrusion Detection Systems: Definition, Need and Challenges. Technical report, SANS Institute, 2021.

scikit yb.org. Feature analysis visualizers — yellowbrick v1.3.post1 documentation. `https://www.scikit-yb.org/en/latest/api/features/index.html`, 2021. (Accessed on 11/02/2021).

Shropshire, J. Natural language processing as a weapon. *Proceedings of the 13th Pre-ICIS Workshop on Information Security and Privacy*, pages 1–9, 2018.

Thejaswini, S. and Indupriya, C. Big data security issues and natural language processing. In *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, pages 1307–1312. 2019. doi:10.1109/ICOEI.2019.8862744.

Ubuntu. Releases @ wiki.ubuntu.com. 2021. (Accessed on 2021-10-14).
URL `https://wiki.ubuntu.com/Releases`

Vidas, T. and Christin, N. Evading Android runtime analysis via sandbox detection. *ASIA CCS 2014 - Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, pages 447–458, 2014. doi:10.1145/2590296.2590325.

VirusTotal. Virustotal search by hash. 2021. (Accessed on 2021-06-21).
URL `https://www.virustotal.com/gui/home/search`

Virustotal.com. Virustotal enterprise contact us. 2021. (Accessed on 2021-11-30).
URL `https://go.chronicle.security/virustotal-private-services`

Wells, J. Orthographic diacritics and multilingual computing. *Language Problems & Language Planning*, 24:249–272, 02 2000. doi:10.1075/lplp.24.3.04wel.

Xu, K., Yao, D. D., Ryder, B., and Tech, V. Poster : System anomaly detection with program analysis and machine learning assistance. *IEEE Security*, pages 1–2, 2013.

Yamaguchi, S. Botnet defense system: Concept, design, and basic strategy. *Information (Switzerland)*, 11(11):1–15, 2020. doi:10.3390/info11110516.

Ying, X. An overview of overfitting and its solutions. *Journal of Physics: Conference Series*, 1168(2):022022, feb 2019.
URL `https://dx.doi.org/10.1088/1742-6596/1168/2/022022`

Younis, A., Malaiya, Y. K., and Ray, I. Assessing vulnerability exploitability risk using software properties. *Software Quality Journal*, 24(1):159–202, 2016. doi:10.1007/s11219-015-9274-6.

Yse, D. L. Your guide to natural language processing (NLP). 2019. (Accessed on 2021-08-01).
URL `https://towardsdatascience.com/your-guide-to-natural-language-processing-nlp-48ea2511f6e1`

Yu, T. and Zhu, H. Hyper-parameter optimization: A review of algorithms and applications. *arXiv*, 3 2020.
URL `http://arxiv.org/abs/2003.05689`

Zerouali, A., Mens, T., Robles, G., and Gonzalez-Barahona, J. On the relation between outdated docker containers, severity vulnerabilities and bugs. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 491–501. 2018. doi:10.5281/zenodo.1250314.

Appendix A

Source code and dataset links

This appendix provides links to source code and datasets used in the research.

A.1 Links to resources used in the thesis

1. Github: `https://github.com/c-goosen/masters-thesis`
2. Virus Total Traces: `https://github.com/c-goosen/masters-thesis/tree/main/datasets/virustotal_traces`
3. Leipzig Intrusion Detection Data Set (2019): `https://www.exploids.de/lid-ds/`
4. PCA optimal variance Jupyter Notebook: `https://github.com/c-goosen/masters-thesis/blob/main/masters_thesis_final-experiment_determine_pca.ipynb`
5. Generate single dataset from multiple sources Jupyter Notebook: `https://github.com/c-goosen/masters-thesis/blob/main/masters_thesis_generate_datasets.ipynb`
6. Experiments Jupyter Notebook: `https://github.com/c-goosen/masters-thesis/blob/main/masters_thesis_final_experiment.ipynb`

A.2 Compare models function code

This appendix provides example code of the Jupyter Notebook applied to run Isolation and Random Forest models and print out their accuracy, timing and AUC score. These statistics and measurements were used to compare the two model's performance.

```
def compare_models(x,y, classifiers):
    training_heatmap = []
    for i, (clf_name,clf) in enumerate(classifiers.items()):
        start = time.time()
        #Fit the data and tag outliers
        if clf_name == "Random Forest":
            clf.fit(x, y)
            y_pred = clf.predict(x)
```

```
elif clf_name == "Support Vector Machine":
    clf.fit(x)
    y_pred = clf.predict(x)
else:
    clf.fit(x)
    y_pred = clf.predict(x)
end = time.time()
print(f"Time model [{clf_name}] was applied: ",end -
    start)
#Reshape the prediction values to 0 for Valid
    transactions , 1 for Fraud transactions
n_errors = (y_pred != y).sum()
# Run Classification Metrics
print("Model - {}: Errors - {}".format(clf_name,n_errors
    ))
print("Accuracy Score :")
print(accuracy_score(y,y_pred))
print("Classification Report :")
clf_report = classification_report(y,y_pred)
print(clf_report)
print('\n\n')
print('Model accuracy score: {0:0.4f}'. format(
    accuracy_score(y, y_pred)))
print('f1 accuracy score: {0:0.4f}'. format(f1_score(y,
    y_pred)))
print('roc_auc_score score: {0:0.4f}'. format(
    roc_auc_score(y, y_pred)))
fpr, tpr, thresholds = metrics.roc_curve(y, y_pred,
    pos_label=1)
roc_auc = metrics.auc(fpr, tpr)
print(roc_auc)
print("\n")
plt.title('Receiver Operating Characteristic')
plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' % roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
```

```
        plt.xlim([0, 1])
        plt.ylim([0, 1])
        plt.ylabel('True Positive Rate')
        plt.xlabel('False Positive Rate')
        plt.show()

        cm = confusion_matrix(y, y_pred)

        cm_matrix = pd.DataFrame(cm,
                        ['True Normal','True Anomaly'],['Pred
                            Normal','Pred Anomaly'])
        training_heatmap.append(cm_matrix)

        print("\n\n")
    return training_heatmap, classifiers
```

A.3 Code to run Experiments 1 and 2

This appendix provides example code of the Jupyter Notebook applied to run Experiments 1 and 2 and print out the accuracy and metrics of each experiment.

```
print("Experiment 1 & 2")

exp_1_and_2_heatmap, exp_1_and_2_classifiers = compare_models(
    train_pca, y_train, classifiers)
```

A.4 Code to run Experiments 3 and 4

This appendix provides example code of the Jupyter Notebook applied to run Experiments 3 and 4 and print out the accuracy and metrics of each experiment.

```
n_estimators = [50, 20, 100, 150, 200, 300, 500]
max_samples = [
    1, 2, 5, 10, 15, 12, 20, 25, 30, 40, 100, 50, 1637, 2000,
        len(train_pca)
]
```

```
# Fit Random Forest to GridSearchCV
tuned_rf = {
    'n_estimators': n_estimators,
    'max_samples': max_samples,
    'verbose': [1],
}
random_forest = GridSearchCV(RandomForestClassifier(random_state
    =42), param_grid=tuned_rf,
                                 scoring="roc_auc",
                                 return_train_score=True, )
rf_model = random_forest.fit(train_pca, train_y)

# Fit Isolation Forest to GridSearchCV
tuned_iso = {'n_estimators': n_estimators,
             'max_samples': max_samples,
             'contamination': [0.12, ],
             'verbose': [1],
             }
isolation_forest = GridSearchCV(IsolationForest(random_state=42)
    , param_grid=tuned_iso,
                                    scoring="roc_auc",
                                    return_train_score=True)
iso_model = isolation_forest.fit(train_pca, train_y)

# Gridsearch CV Classifier
gridsearchcv_classifier = {

    "Random Forest": RandomForestClassifier(

        n_estimators=rf_model.best_params_['n_estimators'],
        max_samples=rf_model.best_params_['max_samples'],

    ),
    "Isolation Forest": IsolationForest(
        n_estimators=iso_model.best_params_['n_estimators'],
        max_samples=iso_model.best_params_['max_samples'],
        contamination=0.12
```

```
    )
}

exp_3_and_4_heatmap_full, exp_3_and_4_classifiers_full =
    compare_models(
     train_pca, y_train, gridsearchcv_classifier
)
```