Evaluation of a Decision Tree-Based Signature Generation approach for IoT Malware Detection

Fiona Y Walsh

Submitted in partial fulfilment of the requirements of
Edinburgh Napier University
for the Degree of
Master of Science in Advanced Security and Digital Forensics

School of Computing

August 2021

MSc dissertation check list

Student Name: Fiona Walsh	Matric: 40334605

Please insert this form, loose-leaf, into each copy of your dissertation submitted for marking.

Milestones	**Date of completion**	**Target deadline**
Proposal	23 March 2021	Week 6
Initial report	28 May 2021	Week 12
Full draft of the dissertation	08 August 2021	2 weeks before final deadline

Learning outcome	**The markers will assess**	**Pages** [1]	**Hours spent**
Learning outcome 1 Conduct a literature search using an appropriate range of information sources and produce a critical review of the findings.	* Range of materials; list of references * The literature review/exposition/background information chapter	p 86-92 p 5-30	415 (for example, 200 hours)
Learning outcome 2 Demonstrate professional competence by sound project management and (a) by applying appropriate theoretical and practical computing concepts and techniques to a non-trivial problem, or (b) by undertaking an approved project of equivalent standard.	* Evidence of project management (Gantt chart, diary, etc.) * Depending on the topic: chapters on design, implementation, methods, experiments, results, etc.	p 94-106 p 31-71	300 (for example, 200 hours)
Learning outcome 3 Show a capacity for self-appraisal by analysing the strengths and weakness of the project outcomes with reference to the initial objectives, and to the work of others.	* Chapter on evaluation (assessing your outcomes against the project aims and objectives) * Discussion of your project's output compared to the work of others.	p 72-85 p 72-85	100 (for example, 100 hours)
Learning outcome 4 Provide evidence of the meeting learning outcomes 1-3 in the form of a dissertation which complies with the requirements of the School of Computing both in style and content.	* Is the dissertation well-written (academic writing style, grammatical), spell-checked, free of typos, neatly formatted. * Does the dissertation contain all relevant chapters, appendices, title and contents pages, etc. * Style and content of the dissertation.		80 (for example, 80 hours)
Learning outcome 5 Defend the work orally at a viva voce examination.	* Performance * Confirm authorship		1 hour

Have you previously uploaded your dissertation to Turnitin? Yes

Has your supervisor seen a full draft of the dissertation before submission? Yes

Has your supervisor said that you are ready to submit the dissertation? Yes

[1] Please note the page numbers where evidence of meeting the learning outcome can be found in your dissertation.

Authorship Declaration

I, *Fiona Y Walsh*, confirm that this dissertation and the work presented in it and my own achievement.

1. Where I have consulted the published work of others this is always clearly attributed;
2. Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;
3. I have acknowledged all main sources of help;
4. If my research follows on from previous work or is part of any larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;
5. I have read and understood the penalties associated with Academic Misconduct.
6. I also confirm that I have obtained **informed consent** from all people I have involved in the work in this dissertation following the School's ethical guidelines.

Signed:

Date: 18th August 2021

Matriculation No: 40334605

General Data Protection Regulation Declaration

Under the General Data Protection Regulation (GDPR) (EU) 2016/679, the University cannot disclose your grade to an unauthorised person. However, other students benefit from studying dissertations that have their grades attached.

Please sign your name under *one* of the options below to state your preference.

The University may make this dissertation, with indicative grade, available to others.

The University may make this dissertation available to others, but the grade may not be disclosed.

The University may not make this dissertation available to others.

Abstract

Despite the fact that IoT devices possess security weaknesses, they are infiltrating daily life more and more. These weaknesses mean that they are vulnerable to malware attacks. Researchers and analysts study malware using a variety of techniques in order to develop defensive strategies against such attacks (i.e., signatures for signature-based detection). An approach to malware detection which has garnered a lot of attention in the last few years is the use of malware detection using Data Mining techniques. The aim of this study was to evaluate a malware signature generation approach which utilised data mining techniques such as Feature Extraction/Selection and Classification.

This was achieved by implementing an end-to-end system which executed IoT malware and benign samples in a sandbox, collected System Calls from the samples, processed that data using feature extraction/selection methods and then fed it into a Decision Tree classifier. The output of that classification was used to auto-generate a set of signatures for IoT malware detection. Malware detection was then performed using the generated signatures.

Experimental results were captured at both the classification and detection phases. These results presented a number of findings including the optimal values for N-gram size (3), top N features (50) and decision tree depth (2) which resulted in the best classification performance and highest detection rate. In addition, a possible positive correlation ($r^2 = 41.45\%$) between classification performance and malware detection rate was identified. Finally, the minimum number (2) of signatures which can achieve a high detection rate for IoT malware was discovered.

This study showed that employing Data Mining techniques for IoT malware detection can have major benefits. The main one being that the manual burden of identifying the most relevant features, on which to base malware signatures, can be replaced with a much more efficient and methodical approach (i.e., Data Mining).

Contents

1 Introduction **1**

1.1 Background 1

1.2 Aim and Objectives 2

1.3 Thesis Structure 3

2 Literature Review **5**

2.1 Introduction 5

2.2 IoT Devices 5

2.2.1 Security on IoT Devices 7

2.2.2 Malware Attacks on IoT Devices 8

2.3 IoT Malware Detection 10

2.3.1 Signature-Based Detection 11

2.4 Malware Detection using Data Mining 12

2.4.1 Sample Collection 13

2.4.2 Dynamic Analysis and System Calls 15

2.4.3 Feature Extraction and Selection 17

2.4.4 Malware Classification 21

2.4.5 Classification Performance Measures 24

2.5 Decision Trees and their role in Signature Generation 26

2.6 Conclusions 28

3 Design **31**

3.1 Introduction 31

3.2 Sample Collection Phase 32

3.3 Malware Analysis Phase 33

3.4 Data Mining Phase 36
3.5 Malware Detection Phase 40
3.6 Experiment Design 42
3.6.1 Experimental Results Design 45
3.7 Conclusions 46

4 Implementation & Results **48**
4.1 Introduction 48
4.2 Environment Setup 48
4.2.1 Laptop Configuration 49
4.2.2 Virtual Machine Configuration 49
4.3 Tools 52
4.4 Sample Collection & Analysis 54
4.4.1 Malware Samples 54
4.4.2 Benign Samples 56
4.5 Dynamic Malware Analysis 57
4.5.1 System Call Dataset Generation 59
4.6 Experiments 60
4.7 Results & Initial Analysis 64
4.7.1 Malware Classification Results 64
4.7.2 Malware Classification Results Analysis 65
4.7.3 Malware Detection Results 67
4.7.4 Malware Detection Results Analysis 68
4.8 Conclusions 71

5 Evaluation **72**
5.1 Introduction 72
5.2 Research Question 1 72
5.3 Research Question 2 75
5.4 Research Question 3 77
5.5 Research Question 4 79
5.6 Limitations of the Study 80
5.7 Conclusions 81

6 Conclusions **82**
6.1 Summary of Work Done 82
6.2 Delivery against Objectives 82
6.3 Future work 84

References **92**

Appendices **94**

A Project Management **94**
A.1 Project Plan 95
A.2 Project Diaries 96

B Source Code **107**
B.1 csv_to_json_convertor.py 107
B.2 dataset_generator.py 109
B.3 feature_extraction_selection.py 111
B.4 malware_classification.py 113
B.5 signature_generator.py 118
B.6 malware_detection.py 120
B.7 main.sh 123

C Experimental Runs **126**

List of Tables

2.1 Feature Extraction Methods used in Malware Detection 18
2.2 Feature Selection Methods used in Malware Detection 19
2.3 Classification Algorithms used in Malware Detection 23
2.4 Measures of Classification-Based Malware Detection Performance (Singh & Hofmann, 2018; Ye, Li, Adjeroh, & Iyengar, 2017) . . . 25

4.1 Laptop Specifications 49
4.2 Details of Virtual Machines 50
4.3 Tools and Scripts 53
4.4 System Calls from YARA signatures (Run Number **119**) 70

5.1 Research Question 1 - Most Featured Parameters 74
5.2 Comparison to related work 75
5.3 Research Question 2 - Most Featured Parameters 77
5.4 Research Question 3 - Top 5 Frequent Combinations 79

C.1 Classification Results: Runs 1 - 35 127
C.2 Detection Results: Runs 1 - 35 128
C.3 Classification Results: Runs 36 - 70 129
C.4 Detection Results: Runs 36 - 70 130
C.5 Classification Results: Runs 71 - 105 131
C.6 Detection Results: Runs 71 - 105 132
C.7 Classification Results: Runs 106 - 140 133
C.8 Detection Results: Runs 106 - 140 134
C.9 Classification Results: Runs 141 - 175 135

C.10 Detection Results: Runs 141 - 175 136
C.11 Classification Results: Runs 176 - 210 137
C.12 Detection Results: Runs 176 - 210 138

List of Figures

2.1 Prediction of connected IoT Devices (Holst, 2021) 6
2.2 2020 Global IoT Malware Volume (Sonicwall, 2021) 7
2.3 Quick History of IoT Threats 2002 - 2018 (F-Secure, 2019) 9
2.4 Overview of Malware Detection by Data Mining 13
2.5 A Simple Decision Tree (Quinlan, 1986) 27

3.1 Decision Tree-Based Signature Generation Framework 31
3.2 Example of a Malware Analysis Lab Configuration (Monnappa, 2018) 35
3.3 Example of the Dynamic Analysis section of a *report.json* 36
3.4 Example of the System Call dataset generated from a *report.json* . 36
3.5 Data Mining Phase Overview 37
3.6 Example of a sample's System Calls post Feature Extraction and Selection 38
3.7 Example of a plotted Decision Tree 40
3.8 Example of a Decision Tree Rule 41
3.9 Example of an Auto-Generated YARA Signature 41
3.10 Example of the output of the Malware Detection Phase 41
3.11 Experiment Parameters Configuration File 43
3.12 Experimental Flow 44
3.13 Classification Report 45
3.14 Confusion Matrix 46
3.15 Collated Malware Detection Results 46

4.1 Domain Name Resolution Example 51

4.2 Aggressive (T4) Nmap scan 52
4.3 Example of results from *virustotal-search.py* 55
4.4 Malware family count across the dataset 55
4.5 Malware family count in final sample collection 59
4.6 Malware Classification and Detection System 60
4.7 Malware Classification Overall Results 65
4.8 Run 72 - Classification Report 66
4.9 Run 72 - Confusion Matrix 67
4.10 Malware Detection Overall Results 68
4.11 Run 119 - Detection Results 69
4.12 Run 119 - Decision Tree 69
4.13 Run 119 - Auto-generated YARA Signatures 70

5.1 Experimental Run Parameters @ $Recall = 97.18\%$ and $Accuracy = 98.52\%$ 73
5.2 Experimental Run Parameters @ $TPR = 97.9\%$ and $FPR = 2.83\%$ 76
5.3 Classification Performance vs Detection Performance 78
5.4 Number of Signatures vs *TPR %* / *FPR %* 80

A.1 Project Plan 95
A.2 20210127 Project Diary 96
A.3 20210209 Project Diary 97
A.4 20210228 Project Diary 98
A.5 20210324 Project Diary 99
A.6 20210504 Project Diary 100
A.7 20210511 Project Diary 101
A.8 20210528 Project Diary 102
A.9 20210608 Project Diary 103
A.10 20210622 Project Diary 104
A.11 20210701 Project Diary 105
A.12 20210727 Project Diary 106

Acknowledgements

I would like to thank my supervisor Rich MacFarlane for his invaluable guidance and support while undertaking this dissertation. In addition, I would also like to thank my employer, CGI IT UK Limited, who kindly sponsored my degree.

Chapter 1

Introduction

1.1 Background

IoT devices are on the rise with predictions of 25.44 billion connected IoT devices worldwide by 2030 (Holst, 2021). Despite this exponential growth, security on IoT devices is considered extremely weak due to the use of default or hardcoded passwords, lack of authentication, non-existent or insecure upgrade channels etc. (Makhdoom, Abolhasan, Lipman, Liu, & Ni, 2019; Zhou, Zhang, & Liu, 2019) and as such they are vulnerable to attack.

Malware attacks on IoT devices was not seen as fruitful in the past primarily due to their highly heterogeneous nature however this attitude has changed in recent years due to their prevalence, lack of security and integration into people's daily lives (Alhanahnah, Lin, Yan, Zhang, & Chen, 2018) – all of which make them a more attractive prospect. In fact, malware attacks on IoT devices have increased significantly over the last few years with no sign of slowing down. As such the need for Malware Detection, specifically lightweight approaches, is becoming more urgent.

Malware Detection can be broadly broken down into three categories: Signature-Based, Heuristic-Based and Specification-Based. Out of these three, Signature-Based is deemed more suitable for resource constrained devices as it is traditionally, less computationally intensive then Heuristic-Based (Tahir, 2018) detection however storage requirements (for a signature database) may reduce that suitabil-

ity somewhat (Arshad et al., 2020). In order to perform Signature-Based detection, signatures must be created which reflect the characteristics of known malware (as opposed to unknown malware which it cannot detect).

Researchers have been looking at ways to develop signature generation approaches for detecting IoT malware and data-driven approaches appear to be taking centre stage. These approaches are implemented using a combination of Data Mining techniques (for feature extraction/selection) and Machine Learning algorithms (for classification / detection). Recent pieces of work utilising this approach for signature generation are (Abbas & Srikanthan, 2017; Alhanahnah et al., 2018; Belaoued et al., 2019; Soe, Feng, Santosa, Hartanto, & Sakurai, 2019) however there is also a large body of research utilising Data-Mining and Machine Learning methods for IoT malware detection in general (Darabian, Dehghantanha, Hashemi, Homayoun, & Choo, 2020; Dimjašević, Atzeni, Rakamarić, & Ugrina, 2016; Huang, Hou, Zheng, & Feng, 2018; Khammas, 2020; Shobana & Poonkuzhali, 2020; Singh & Hofmann, 2018).

Research in this area is in its infancy and the studies so far have been somewhat limited (i.e., small number of IoT malware samples included in the study (Abbas & Srikanthan, 2017)), partial experimentation (i.e., generated signatures were not tested (Soe et al., 2019)), sub-optimal analysis technique (i.e., static analysis can be easily circumvented by obfuscation techniques) used to elicit features for signature generation (Alhanahnah et al., 2018)). This study aims to address some of limitations.

1.2 Aim and Objectives

The aim of this project is to evaluate the accuracy of using a Behaviour-Based (System Calls) Signature Generation approach using Data Mining Techniques in the detection of IoT Malware.

The approach will use data mining techniques (i.e., feature extraction / selection methods and a Decision Tree classification algorithm) to elicit the most relevant system calls to a) generate a model which will classify executables as malware or benign and b) to generate a minimal (yet optimal) subset of YARA signatures (from said model) which will be used for IoT Malware detection.

The following objectives will be addressed in this work:

- To conduct a literature review on current research in behaviour-based Signature Generation approaches for IoT Malware Detection. Included in this review will be an analysis of the malware that attack IoT devices and the approaches that have been adopted to detect them.
- To design, based on existing literature, a method which uses Data Mining techniques to elicit relevant behavioural features from IoT Malware and Benign samples which can be used to detect malware.
- To implement the devised method in a systematic manner which allows for an optimal set of signatures to be generated for use in IoT Malware Detection.
- To evaluate the accuracy of the generated signatures in detecting IoT Malware in a controlled, repeatable test environment.

1.3 Thesis Structure

This thesis is structured as follows:

- Chapter 2 - Literature Review - This chapter provides a review of the current research in behaviour-based Signature Generation approaches for IoT Malware Detection.
- Chapter 3 - Design - This chapter discusses the methods used in this study and gives an overview of implementation design and experimental design.
- Chapter 4 - Implementation & Results - This chapter discusses the environment setup, the tools used and the various stages of implementation of the project. It also presents a selection of the results from both the classification and detection phases.
- Chapter 5 - Evaluation - This chapter evaluates the results generated in the implementation phase and addresses the research questions raised in the literature review. It also discusses the limitations of this study.

- Chapter 6 - Conclusions - This chapter summarises the work done, maps the delivery against the original objectives and provides suggestions for future work.

Chapter 2

Literature Review

2.1 Introduction

This chapter, broken into two parts, reviews the current state of the research area in which this work aims to sit. The first part starts by introducing IoT (Internet of Things) devices, discusses their security weaknesses and summarises why traditional heavy-weight security mechanisms are not sufficient. This discussion is then followed by a summary of the malware attacks that IoT devices are susceptible to and the different ways in which they can be detected.

The second part focuses on Malware Detection using Data Mining techniques and discusses how this approach has been implemented by other researchers. It examines what data has been used in Malware Detection, how it has been selected and the way in which it is used in the detection. Finally, the chapter concludes with a summarisation of the existing research and how it forms the basis for the work carried out in this dissertation.

2.2 IoT Devices

IoT devices are enabling the evolution of a smart ecosystem which is growing year on year with no signs of slowing down. In fact, connected devices are predicted to triple (from 2019) by 2030 as shown in Figure 2.1. *Wikipedia* defines IoT as:

The Internet of things (IoT) describes the network of physical objects

> *- a.k.a. "things" — that are embedded with sensors, software, and other technologies for the purpose of connecting and exchanging data with other devices and systems over the Internet.*

The convenience brought by these devices, however, could come at a great cost as malware attacks on IoT devices saw huge growth in 2020 as shown in Figure 2.2. The 2021 Cyber Threat report from SonicWall stated that IoT malware attacks had risen to 56.9 million in 2020, a 66% increase on 2019. Compromising IoT devices can be, in some cases, quite easy due to factors such as a) poor security practices (e.g., hardcoded or default passwords both by the user and the device manufacturers), b) lack of authentication and security policies on the device and c) non-existent or insecure firmware upgrade channels (Makhdoom et al., 2019; Zhou et al., 2019).

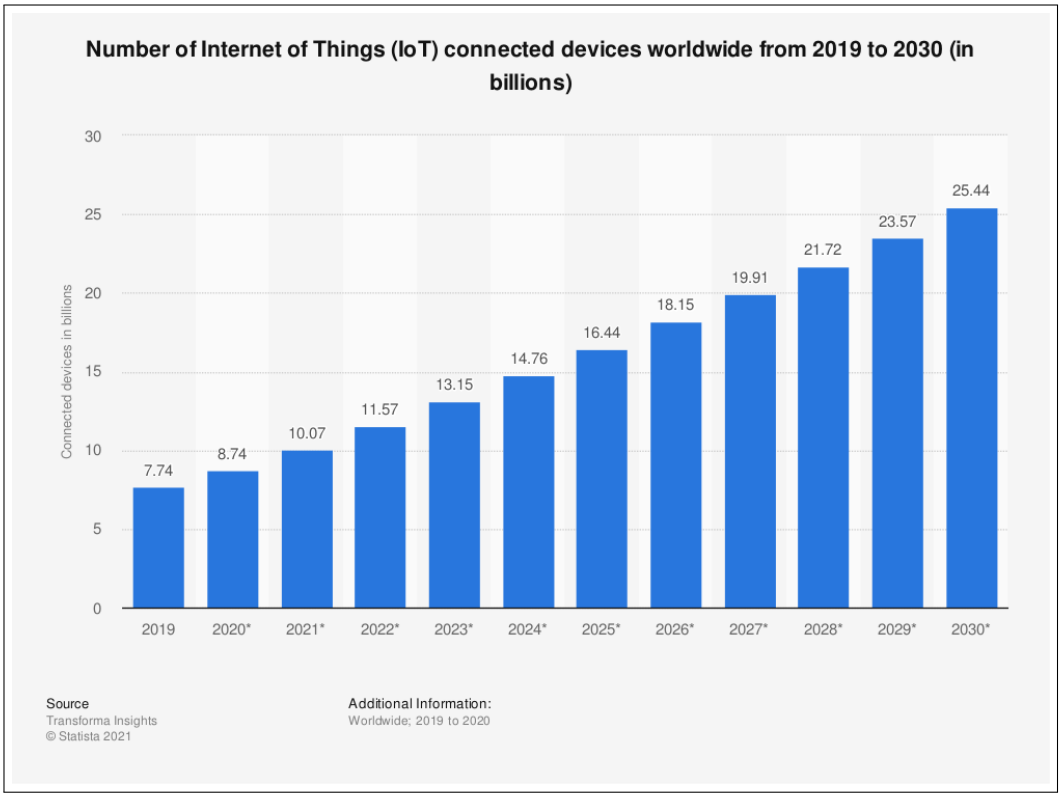


Figure 2.1: Prediction of connected IoT Devices (Holst, 2021)

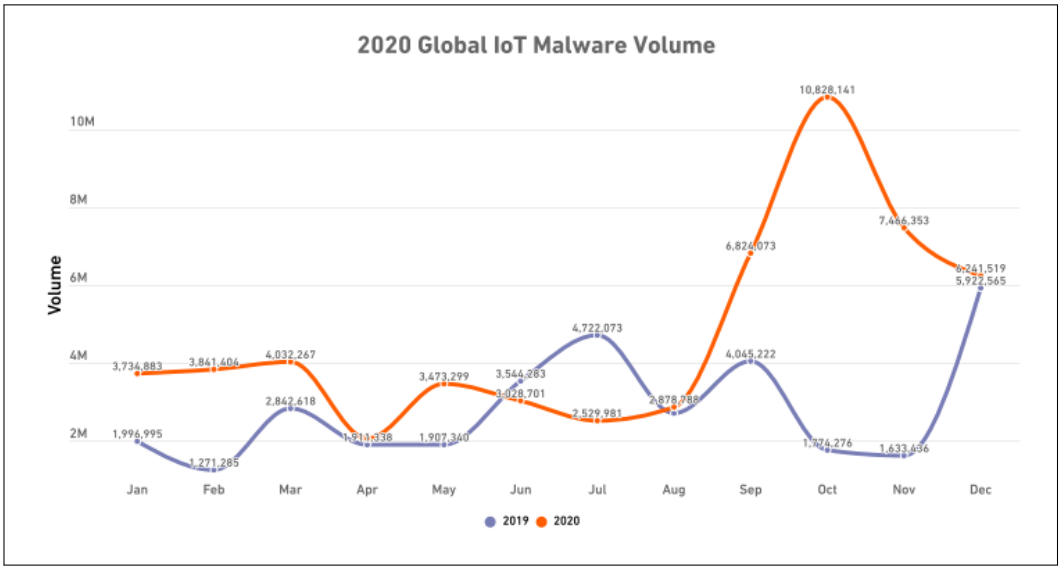


Figure 2.2: 2020 Global IoT Malware Volume (Sonicwall, 2021)

2.2.1 Security on IoT Devices

Security on IoT devices is considered very weak due to a number of reasons. Primary reasons are 1) lack of consideration given to security in the design of the devices, 2) the constrained nature of the devices means that traditional security options might not be suitable, 3) end users do not consider security (OWASP (2018) lists weak, guessable or hardcoded passwords as the number one vulnerability) and 4) unpatched vulnerabilities due to no upgrade channel and/or manufacturer patches. Ye et al. (2017) comment on the popularity of IoT devices and on the market pressure to produce vast quantities of such devices which means that manufacturers are forgoing security in a bid to meet the demand. Researchers in Zhou et al. (2019) tried to assess the security risks of IoT devices by categorising them as eight so-called ”IoT features” (i.e., *Interdependence, Diversity, Constrained, Myriad, Unattended, Intimacy, Mobile* and *Ubiquitous*). For each category, they discussed the threat, the challenge and the opportunity (i.e., ways in which the threat may be addressed by overcoming the challenge).

A number of different solutions have been suggested in order to overcome these security weaknesses. For example, one solution proposes running verified bytecode in an isolated framework to move the burden of enforcing security

from the hardware (which potentially has security vulnerabilities) to the software (Radovici, Rusu, & Serban, 2018). Device hardening, more commonly used in networked servers, was proposed by Echeverría, Cevallos, Ortiz-Garces, and Andrade (2021) and S. K. Choi, Yang, and Kwak (2018) to minimise the attack surfaces of IoT devices. Suggested hardening actions included steps such as changing standard port numbers to a different unknown range, updating the device's firmware, changing default passwords, secure boot settings and disabling unused file systems. Finally, a vulnerability mitigation framework was proposed in Hadar, Siboni, and Elovici (2017) to prevent the exploitation of an IoT device. Policies for each known vulnerability (based on CVE records) that the IoT device may possess are represented as signatures of traffic or executables to block, thus protecting the device from attack.

2.2.2 Malware Attacks on IoT Devices

Despite solutions being proffered to protect IoT devices, they are still susceptible to an ever-increasing litany of malware attacks. Figure 2.3 shows a history of malware threats on IoT devices from 2002 to 2018. Two clear observations which can be made (from the diagram) are 1) there has been a consistent multiplication of threats as each year has passed, going from 1 in 2002 to 3 in 2014 and then 19 in 2018 and 2) *Backdoor* type threats (e.g., Tsunami, Gafgyt, Mirai). are one of the most common types of attacks that IoT devices experience (F-Secure, 2019). In the case of a Backdoor attack, an IoT device is an ideal target as it has little to no standard security measures (as previously discussed) and a malicious actor can easily circumvent the ones it does have to gain high level user access all the while leaving no trace (MalwareBytes, 2021).

In order to capture the types of malware that were actively attacking IoT devices, researchers in Pa et al. (2016) created a honeypot (called IoTPoT) which aimed to emulate Telnet services of various IoT devices. IoTPot ran for 81 days in 2015 and among the observations made by the researchers, one was that the most common intrusion (to the device) was gained through dictionary attacks (i.e., weak guessable passwords due to poor security practices on open Telnet ports). It seems security lessons have not been learnt over the past few years as *Kaspersky*

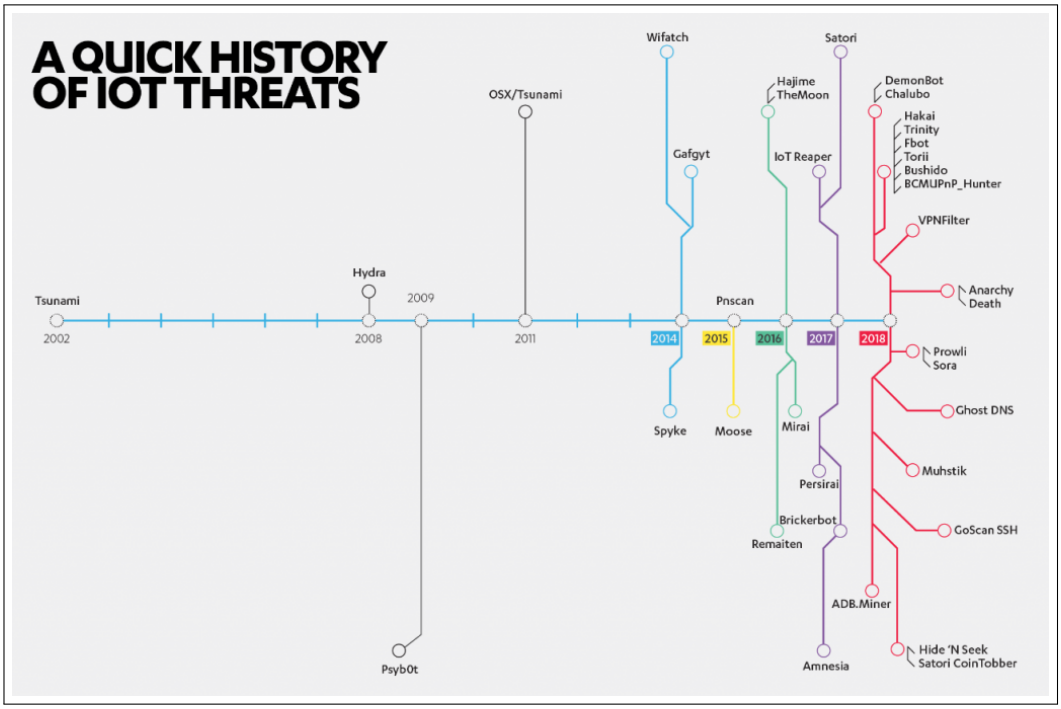


Figure 2.3: Quick History of IoT Threats 2002 - 2018 (F-Secure, 2019)

released IoT threat statistics in Q1 2021 which confirmed that most devices were still being attacked using the Telnet protocol (69.48% for telnet, 30.52% for SSH) (Kaspersky, 2021). These statistics also showed that *Mirai* is still number one in the charts (50.50% share) when it comes to the malware most downloaded to IoT devices following a successful attack.

Mirai falls into the category of brute-force attacks on IoT devices but there is also another infection route, namely exploiting vulnerabilities. The authors in Wang et al. (2017) used these two infections routes (brute-force and exploiting vulnerabilities) to analyse multiple different IoT malware over the years. BASHLITE is a malware which infects IoT devices by exploiting a vulnerability in the bash shell (*Shellshock* software bug). This bug allows a user to execute arbitrary commands to gain unauthorised access (Wikipedia, 2021a). Both BASHLITE and *Mirai* botnets have wreaked havoc using DDoS (Distributed Denial of Service) attacks. The most prominent being the attack by *Mirai* on *Dyn*, a DNS provider, in 2016 which brought down sites including PayPal, Netflix, Sony PlayStation and even Twitter to name a few (Thielman & Johnston, 2016). Combining the rise of

malware targeting IoT devices and the lack in traction with regards to resolving their security vulnerabilities, it seems that the threat of weaponising IoT devices (as one means of attack) is ever-present. As such, the only viable way to defend against these malware attacks is to detect them before they have a chance to unfold.

2.3 IoT Malware Detection

Malware detection occurs in many different forms with signature-based and anomaly-based being the most common approaches. Some methods can only detect known malware (signature-based) whereas others can detect unknown malware but they also generate high volumes of false positives (anomaly-based) (Mudgerikar, Sharma, & Bertino, 2019). Researchers in Mudgerikar et al. (2019) developed an anomaly-based Intrusion Detection System (IDS) that profiles the behaviour of IoT devices based on system level information (e.g., System Calls and running process parameters). Their system uses three different levels of detection (i.e., PWM (Process White-list), PBM (Process Behaviour Parameters) and SBM (System Call Behaviour Parameters)) however they found, in general, that as their detection accuracy increased so did the computational and storage costs on the device. The signature-based solution proposed in Dutra et al. (2019) attempts to overcome the limited processing power (of IoT devices) by performing detection using lightweight agents deployed to the IoT devices. The agent controller, hosted on a resourced machine, performs threat intelligence based on information sent from the agents thus taking on the burden of the computational overhead.

In Wan et al. (2020), researchers used Machine Learning to detect malware based on static information (Byte Sequences) extracted from ELF binaries. They were able to achieve a detection accuracy of 99.96% across multiple different CPU architectures (e.g., ARM, MIPS, PowerPC, SPARC), typically found in IoT devices. *Control Flow Graphs (CFG)* combined with Deep Learning was used in Alasmary et al. (2019) to detect IoT malware. As well as building a detection system, the authors compared CFGs of Android and IoT malware and they observed that the CFGs for IoT malware had a lesser number of nodes and edges (main evaluation metric of graph size) despite a similar file size indicating the use

of evasive techniques (by the malware authors) to prevent static analysis.

The research above shows that there is no one perfect malware detection approach and many researchers have looked to hybrid approaches (e.g., combining signature-based and anomaly-based (Aslan & Samet, 2020; Damodaran, Troia, Visaggio, Austin, & Stamp, 2017)). However, while hybrid approaches mean that weaknesses in one approach are compensated for by the addition of a complementary approach, this tactic typically does not result in a lightweight solution (necessary for IoT malware detection). Signature-Based detection is routinely criticised for its lack of ability to detect unknown malware. However, research to date generally agrees that it is a lightweight approach due to its lack of computational overhead (Arshad et al., 2020; Aslan & Samet, 2020; Damodaran et al., 2017; Ioulianou, Vasilakis, Moscholios, & Logothetis, 2018; Soe et al., 2019) thus making it the most suitable option for IoT malware detection.

2.3.1 Signature-Based Detection

Signature-based detection is widely used by anti-virus providers for malware detection (Souri & Hosseini, 2018). A signature is typically a unique sequence of bytes or strings which identify a specific malware and because they have been developed off the back of analysing specific malware, they suffer from very few false positives (Bazrafshan, Hashemi, Fard, & Hamzeh, 2013). This statement however alludes to the main weakness of the approach (i.e., a signature can only be generated for a malware once it has been seen), as such, unknown malware or zero-day attacks cannot be detected. Producing signatures for known malware (for use in detection systems) tend to incur a time delay as the malware has to be analysed first, this means that a ”known” malware may go undetected for some time after it was first discovered (Ye et al., 2017).

As the number of malware have grown, so has the number of signatures required to detect it. This growth means a larger signature database is required by a detection system and the knock-on effect is a decrease in system performance (more signatures to scan through for a match) (Hughes & Qu, 2014). It also means that in its traditional format, it becomes an impracticable option for resource constrained devices. As such, researchers in Abbas and Srikanthan (2017) proposed

a low-complexity, lightweight approach which involved the use of generic signatures to detect groups of malware. The advantage of this approach is that the subset of signatures required to detect the malware is kept small (i.e., only 7 signatures were required to detect 70 malware samples). The negative with this approach is that individual types of malware cannot be identified but this is possibly the compromise that needs to be made for IoT devices.

In Alhanahnah et al. (2018), the authors used high-level structural, statistical and string feature vectors to generate signatures offline (to reduce the computation overhead) for lightweight IoT malware detection. In order to generate these signatures, researchers performed static analysis (extracted printable strings) on a large (5,150) set of IoT malware samples. Signatures based on strings, however, can be susceptible to obfuscation techniques and/or encryption which malware authors use to circumvent detection. Creating signatures based on features extrapolated through dynamic analysis (i.e., behaviour-based indicators) that are not easy to manipulate (by the malware author) can be used to combat this (Tahir, 2018). While not technically signature-based detection, the researchers in Park, Powers, Prashker, Liu, and Yener (2020) did use features extracted through dynamic analysis to demonstrate that a high detection rate (i.e., 93.18%) could be achieved for obfuscated IoT malware. Malware detection and/or signature generation through static or dynamic analysis is a data intensive task. As such, researchers have looked to Machine Learning, specifically, Data Mining, to further the work in this area.

2.4 Malware Detection using Data Mining

Malware Detection using Data Mining involves a number of phases as shown in Figure 2.4. The process which incorporates these phases has become a standard followed by many researchers however the details (e.g., the algorithms used in each phase) has varied in the research to date (Belaoued et al., 2019; Phu, Dang, Quoc, Dai, & Binh, 2019; Shobana & Poonkuzhali, 2020; Wan et al., 2020; Ye et al., 2017). For example, some researchers have used static analysis to collect features and clustering or classification to detect malware (Alhanahnah et al., 2018; Huang et al., 2018; Wan et al., 2020; Yuxin, Xuebing, Di, Li, & Zhanchao,

2011). Others have used dynamic analysis to capture behavioural data (e.g., API calls) from a set of samples and then used a classifier to determine if the sample is malware or benign (Aiswarya Mohan, Chandran, Gressel, Arjun, & Pavithran, 2020; Asmitha & Vinod, 2014; Dinh et al., 2019; Firdausi, Lim, Erwin, & Nugroho, 2010; Norouzi, Souri, & Samad Zamini, 2016; Phu et al., 2019; Shobana & Poonkuzhali, 2020; Tian, Islam, Batten, & Versteeg, 2010).

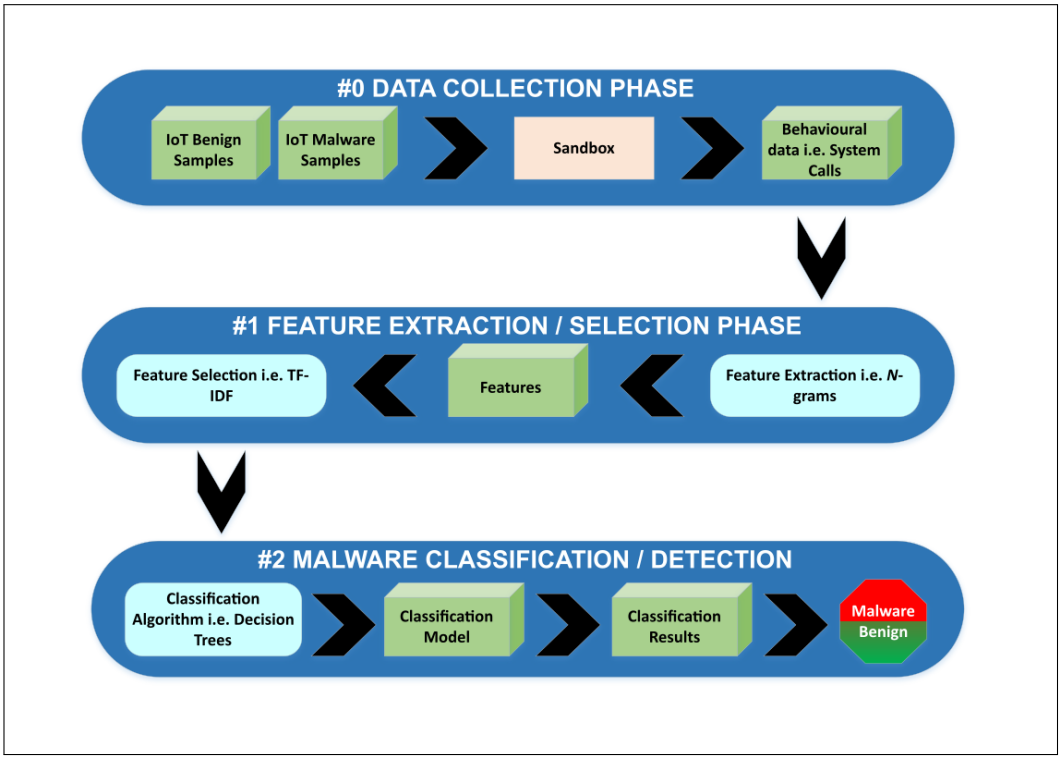


Figure 2.4: Overview of Malware Detection by Data Mining

2.4.1 Sample Collection

In order to perform malware detection using data mining techniques, a dataset is required. In recent years, there have been very few suitable datasets publicly available for IoT malware detection research and the ones that were available tended to be narrow in scope and/or limited in size. As such, researchers have either used less suitable datasets or have had to generate their own datasets for their studies. Recent examples of insufficient datasets are Abbas and Srikanthan (2017)

who only included a sample set of 70 malware executables (no benign examples) meaning their reported results have limited significance. Another study used the UNSW-NB15 dataset (network traffic captures) for IoT malware detection despite the traffic not originating or targeting IoT devices (Soe, Feng, Santosa, Hartanto, & Sakurai, 2020). Researchers in Takase, Kobayashi, Kato, and Ohmura (2020), presumably unable to find a suitable pre-existing dataset, created their own by collecting benign files from Raspbian OS and setting up a honeypot to capture an equivalent set of malware samples. Another approach used in studies, has been to extract benign files from IoT firmware images and to download malicious files from websites such as VirusTotal in order to build a dataset for evaluation (Wan et al., 2020).

Despite the scarcity of IoT malware datasets, some studies have used a pre-existing dataset, namely IoTPoT, that has been fit for purpose either on its own or in combination with other data sources (J. Choi et al., 2019; Mudgerikar et al., 2019; Shobana & Poonkuzhali, 2020). Phu et al. (2019) also used the IoTPoT dataset (made up of ELF executables) in their study but removed all samples which were not MIPS ELF files as their work focused exclusively on classifying malware targeting MIPS-based IoT devices. The original IoTPoT dataset[1] contains malware samples which were captured during 2016 using a bespoke low-interactive (simply responds to Telnet requests) IoT malware honeypot (IoTPoT) (Pa et al., 2016). In 2020, the same researchers set up an adaptive honeypot framework which emulated IoT devices (X-POT) and included a higher level of interactivity (i.e., emulating vulnerable services by using honeypot responses based on real responses collected from internet-wide scans) (Kato, Tanabe, Yoshioka, & Matsumoto, 2021). 1,560 malware samples were collected over a number of months and the top malware families found were *Mirai* and *gafgyt*.

Once a dataset has been generated, the malware samples in that dataset must be analysed in order to produce relevant raw data for feature extraction/selection and ultimately classification or clustering.

[1]Newer datasets are now available - `https://sec.ynu.codes/iot/available_datasets`

2.4.2 Dynamic Analysis and System Calls

Dynamic Analysis Static and dynamic analysis are the two main types of malware analysis that can be performed to determine what a malware does. The first method, Static Analysis, involves analysing a malware sample without running it and using tools such as decompilers, dissemblers, source code analysers etc. to inspect the code. Dynamic Analysis is the opposite in that the sample is executed in a safe environment (e.g., sandbox) in order to observe its functional behaviour (Tahir, 2018). There are pros and cons to both approaches, for example, *Static Analysis* is less time-consuming and safer to perform, it can also explore all execution paths however it cannot analyse packed malware. Alternatively, *Dynamic Analysis* displays the malware's behaviour and can analyse packed malware as the malware must unpack at some point during execution. The drawback is that it is time-consuming and riskier to run (Belaoued et al., 2019). As the work in this thesis focuses on the behaviour of malware and consequently a dynamic analysis approach has been adopted, the discussion going forward will focus on that type of analysis.

System Calls Dynamic analysis produces numerous outputs (e.g., instruction traces, registry changes, memory contents) (Damodaran et al., 2017), all of them potentially useful for malware detection however the most frequently used feature in literature to date has been System Calls (Abbas & Srikanthan, 2017; Aiswarya Mohan et al., 2020; Asmitha & Vinod, 2014; Canzanese, Mancoridis, & Kam, 2015; Dimjašević et al., 2016; Firdausi et al., 2010; Liu, Xue, Xu, Zhong, & Chen, 2019; Phu et al., 2019; Shobana & Poonkuzhali, 2020; Singh & Hofmann, 2018). *System Calls* are requests from a computer program to the kernel of the operating system (on which it is running) for a particular service (e.g., file manipulation, process control). There are six main categories of System Calls according to Wikipedia (2021d): 1) *Process Control*, 2) *File Management*, 3) *Device Management*, 4) *Information Maintenance*, 5) *Communication* and 6) *Protection*. A recent survey showed that Linux is the top operating system (OS) for IoT devices with 43% usage (FreeRTOS has 35% and Windows has 31%) (Eclispe IoT, 2020), as such all future mentions of system calls will refer to Linux system calls

exclusively.

A profile of a malware's behaviour can be built up by observing the system calls it uses. For example, lots of file management calls may indicate that Ransomware is encrypting files or alternatively, a series of unexpected protection calls (e.g., chmod, chown) may indicate that a piece of malware is attempting to elevate privileges for a malicious actor. In Firdausi et al. (2010), the authors submitted a collection of malware and benign samples to an online dynamic analysis service which then executed them in a controlled environment and produced a report which included system calls. Unfortunately, no details are supplied on what system calls were used for malware detection, so no inferences can be made in terms of the effectiveness of specific system calls for detecting malware. The authors did, however, report an overall accuracy rate of 96.8% for detection which does support the validity of using system calls for malware detection.

A study by Shobana and Poonkuzhali (2020) did show the most popular (based on frequency) system calls in the form of generated word clouds for malware (and benign) files. They used *N*-grams and *TF-IDF* (Term Frequency-Inverse Document Frequency) to select relevant features from the captured system calls and observed that the *read* system call had the highest frequency for malware files. This observation was also made in Phu et al. (2019) who documented that the *read* and *sendto* system calls were the most used by ELF files and represented 86% of all captured log files. A separate investigation into Android malware found that 31 specific system calls were essential to differentiating between malware and benign (android) applications and again, the *read* system call came out on top again (Singh & Hofmann, 2018). Running malware (and benign) samples to trace system calls results in large volumes of data being generated, Wikipedia (2021d) quotes over 300 different system calls on Linux. As expected, not all of the 300+ system calls are used by each executable, for example, researchers in Phu et al. (2019) observed that the average number used by ELF files was 146. This is still quite a lot of features to process and as such needs to be whittled down to the most relevant set.

2.4.3 Feature Extraction and Selection

As shown in Figure 2.4, once the data collection phase is complete, the next stage is feature extraction and selection. Relevant features that can characterise an executable must be extracted from the system calls which were captured. *Feature Extraction* is a Data Mining technique which derives or extracts features from raw data. *Feature Selection* occurs after feature extraction and is used to select the most relevant features (produced by feature extraction) to train a Machine Learning model. This selection is required because 1) it reduces the amount of features that a classifier (for example) has to process, 2) it removes redundant or irrelevant features which contribute little value to a classifier's accuracy and 3) running a classifier on less features means that it will require less time to train and ensures scalability (Asmitha & Vinod, 2014; Firdausi et al., 2010; Khammas, Hasan, Ahmed, Bassi, & Ismail, 2019). There is also the potential for the classifier to suffer from over-fitting (the model has over-learned the patterns in the training data and is not generalised enough when it comes to the test phase) if the feature set is not reduced to the most relevant (Phu et al., 2019).

Tables 2.1 and 2.2 show the most common feature extraction and selection methods used in the literature surveyed. As can be seen from the tables there are a mixture of methods used, with *N-grams* being the most popular extraction method (8 papers) and *Information Gain* (IG) being the most popular selection method (6 papers). Note however that the most common combination of feature extraction and selection methods is *N-grams* and *TF-IDF* (3 papers), these methods we will be discussed in more detail going forward.

N-Grams

N-grams are contiguous sequences of terms (e.g., characters or words) with a length *n*. For example, the abbreviated system call trace "*..., brk, mmap2, access, openat, openat, stat, ...*" modelled as 2-gram (also known as *bigram*) values would be: "*brk,mmap2*", "*mmap2,access*", "*access,openat*", "*openat,openat*", "*openat,stat*". They are traditionally a NLP (Natural Language Processing) model but have been widely adopted for use in malware detection (Boujnouni et al., 2016; Canzanese et al., 2015; Dinh et al., 2019; Huang et al., 2018; Khammas, 2020; Shobana &

Feature Extraction Method	Count	References
N-grams	8	Boujnouni et al. (2016); Canzanese et al. (2015); Dinh et al. (2019); Huang et al. (2018); Khammas (2020); Shobana and Poonkuzhali (2020); Wan et al. (2020); Yuxin et al. (2011)
Custom Approach	5	Aiswarya Mohan et al. (2020); Asmitha and Vinod (2014); Bai et al. (2013); Dimjašević et al. (2016); Tian et al. (2010)
Longest Common Subsequence (LCS)	1	Belaoued et al. (2019)
Markov Chains	1	Park et al. (2020)

Table 2.1: Feature Extraction Methods used in Malware Detection

Poonkuzhali, 2020; Wan et al., 2020; Yuxin et al., 2011).

When a large n value is used, the N-grams will contain more contextual (as opposed to general) information and this may have an impact on classification accuracy (i.e., only very specific classes are correctly identified). Alternatively, if the n value is too small, the amount of N-grams (features) will be greater and this will impact the classifier's performance (i.e., memory and CPU usage will be high if all of the features are used). Hence determining the optimal n value is crucial (Liu et al., 2019; Wan et al., 2020; Yuxin et al., 2011), as is combining extraction with a feature selection method to reduce the dimensionality of the features (Ye et al., 2017).

In a study on detecting IoT malware, researchers used a n value of 10 to extract system call N-grams (Shobana & Poonkuzhali, 2020). This value was chosen under the assumption that as IoT devices are resource constrained, the programs that run on them must take limited inputs therefore sequences of 10 system calls (in length) are sufficient to extract meaningful patterns. A n value of 3 was used in Huang et al. (2018) to extract Dalvik (Android) opcodes (similar to system calls) N-grams. The authors confirmed that this n value resulted in the best overall

Feature Selection Method	**Count**	**References**
Information Gain (IG)	6	Babaagba and Adesanya (2019); Bai et al. (2013); Boujnouni et al. (2016); Khammas et al. (2019); Singh and Hofmann (2018); Yuxin et al. (2011)
Correlation-Based Feature Selection (CFS)	4	Firdausi et al. (2010); Khammas et al. (2019); Singh and Hofmann (2018); Soe et al. (2020)
Term Frequency-Inverse Document Frequency (TF-IDF)	4	Belaoued et al. (2019); Canzanese et al. (2015); Huang et al. (2018); Shobana and Poonkuzhali (2020)
Chi-Square	3	Khammas et al. (2019); Phu et al. (2019); Singh and Hofmann (2018)
Gain Ratio (GR)	2	Khammas (2020); Khammas et al. (2019)
Principal Component Analysis (PCA)	2	Khammas (2020); Khammas et al. (2019)
Custom Approach	1	Dimjašević et al. (2016)
X–Symmetric Uncertainty(X–SU)	1	Asmitha and Vinod (2014)

Table 2.2: Feature Selection Methods used in Malware Detection

performance of the classifiers (*K*-Nearest Neighbor (KNN), Logistic Regression (LR), Support Vector Machines (SVM) and Random Forests (RF)) but did not provide any further details on why. Canzanese et al. (2015) experimented with different n values (i.e., 1 - 4) combined with varying system call trace lengths l and found that (malware) detectors which used $n = 1$ performed poorly (max. 30% True Positive Rate (TPR)) across all trace lengths ($l \in \{500, 1000, 1500\}$ system calls). $n \in \{3, 4\}$ produced the best TPR ($> 90\%$) at $l = 1000$, this was the peak and the rate plateaued after that.

Term Frequency-Inverse Document Frequency (TF-IDF)

Studies have shown the importance of using a feature selection method, for example, Firdausi et al. (2010) reported 3 out of 4 classifiers had a slight increase in accuracy (the *Naive Bayes* classifier having the biggest jump from 65.4% to 92.3%) when run in combination with feature selection on their binary datasets. While the increase in accuracy was modest (for the most part), the authors still achieved a $> 90\%$ accuracy rate even after reducing their features, through feature selection, from 5191 to 116 (reduction of 97.7%). This considerable reduction in features has a number of positives, for example, 1) it reduces the storage requirement for the features, 2) the classifiers take less time to train and 3) by processing less features, the classification will be less processor intensive (i.e., CPU usage). Similar observations (accuracy rate was maintained or improved with feature selection) were made in Singh and Hofmann (2018) and Babaagba and Adesanya (2019) who also tested with and without feature selection.

Term Frequency-Inverse Document Frequency (TF-IDF)[2] has been used frequently in conjunction with feature extraction methods to produce an optimal feature set for classification (Belaoued et al., 2019; Canzanese et al., 2015; Huang et al., 2018; Shobana & Poonkuzhali, 2020). It is a 2-part ranking function, the first part (*Term Frequency*) counts the total number of occurrences of a term in a document and then divides that by the total number of all the terms in the same document, see eq. 2.1:

[2] *TF-IDF* - `http://www.tfidf.com/`

$$TF(t) = \frac{\text{Number of times term } t \text{ appears in a document}}{\text{Total number of terms in the document}} \tag{2.1}$$

The second part (*Inverse Document Frequency*) determines how important the term from the first part is by computing the logarithm of the total number of documents divided by the number of documents where the term appears, see eq. 2.2:

$$IDF(t) = \log_e \frac{\text{Total number of documents}}{\text{Number of documents with term } t \text{ in it}} \tag{2.2}$$

The IDF value acts as a weight and overall, the higher the TF-IDF value of a term, the rarer it is. Terms do not just relate to individual words however; they could also refer to *N*-grams. Belaoued et al. (2019) used TF-IDF to identify the most common API calls, IP addresses and domain names used by malware and benign files. Their choice of TF-IDF as a feature selection method was motivated by its speed and effectiveness. In Shobana and Poonkuzhali (2020), researchers used TF-IDF to extract the most significant features from system call *N*-grams before entering them into a RNN (Recurrent Neural Network) in order to classify samples as malware or benign. Researchers in Canzanese et al. (2015) used TF-IDF as a feature scaling technique and captured TPR scores for TF alone (no weight applied) and TF-IDF transformation (weight applied). They found that TF-IDF transformation was effective at highlighting uncommon *N*-grams whereas TF alone was much less effective, this is not really surprising as all terms (e.g., *N*-grams) have equal importance until a weight is applied (IDF value).

2.4.4 Malware Classification

The performance of malware detection approaches using classification is heavily influenced by a) the feature set and b) the classification algorithm (Bazrafshan et al., 2013). Feature extraction/selection was covered in the previous section so the following section will analyse the classification methods used in related work.

Classification is a two-stage process, the first stage is where the classification model is built and the second stage is where the model is used to perform the classification. Considering the first stage (known as the training phase), this is where the extracted features for each training sample (i.e., executable file) are

converted to vectors. They are then combined with their class label (e.g., malware or benign), as classification is a supervised learning technique, and are input into a classification algorithm (e.g., Decision Tree (DT)) which builds a model. In the second phase (known as the testing phase), the features for a new, unseen set of executables are extracted (using the same method as the first stage) and these are fed (minus the class labels) into the model. The classification model uses parameters it learned from the training phase to classify the new samples as benign or malware (Ye et al., 2017). Binary classification (two class labels) is the most common type used in malware detection (Asmitha & Vinod, 2014; Darabian et al., 2020; Firdausi et al., 2010; Islam, Tian, Batten, & Versteeg, 2013; Norouzi et al., 2016; Singh & Hofmann, 2018; Tian et al., 2010) with only a handful using multi-class (multiple class labels) (Kim, Shim, Hong, Shin, & Choi, 2020; Koroniotis, Moustafa, Sitnikova, & Turnbull, 2019) and no occurrences of multi-label or imbalanced (found in the surveyed literature).

Table 2.3 shows the most frequently used classification algorithms in malware detection and in general, which algorithm performed better in their individual studies. As can be seen in the table, *J48* and *SVM* are the most commonly used classification algorithms for malware detection. It should be noted nevertheless, that while *RF* lags behind both algorithms in terms of use, it does have a better top performance rate. Note for the purpose of comparison, the classification algorithms were scored as top performing if they achieved the highest reported *Recall* percentage (if available or if not, then the highest *Accuracy* percentage).

Only one study from all of the literature reviewed considered the age of the malware and the effect that it might have on the classification outcome. Researchers in Islam et al. (2013) performed testing with older malware on its own, new malware on its own and then both combined. They found that classification was less effective on newer malware and proposed this was because newer malware is better at avoiding anti-virus techniques. Combining the old and new malware datasets improved the *Accuracy* percentage (no *Recall* percentage was recorded) which had been achieved for the new malware alone. To date, Wan et al. (2020) is the only study which reported malware detection (and family classification) results per IoT supported CPU architecture. The authors found that there was a slight improvement to the *Accuracy*, *Precision* and *Recall* scores for binary

Classification Algorithm	Classification Algorithm Type	Total Count	Top Performing Occurrences	References
J48 (C4.5 algorithm)	Decision Tree	9	1	Asmitha and Vinod (2014); Darabian et al. (2020); Firdausi et al. (2010); Islam et al. (2013); Kim et al. (2020); Norouzi et al. (2016); Singh and Hofmann (2018); Soe et al. (2019); Tian et al. (2010)
Support Vector Machine (SVM)	Deterministic	8	3	Darabian et al. (2020); Firdausi et al. (2010); Islam et al. (2013); Koroniotis et al. (2019); Norouzi et al. (2016); Singh and Hofmann (2018); Tian et al. (2010); Wan et al. (2020)
Random Forest (RF)	Ensemble Decision Tree	6	4	Asmitha and Vinod (2014); Darabian et al. (2020); Islam et al. (2013); Kim et al. (2020); Singh and Hofmann (2018); Tian et al. (2010)
k-Nearest Neighbor	Lazy Classification	5	1	Darabian et al. (2020); Firdausi et al. (2010); Kim et al. (2020); Singh and Hofmann (2018); Wan et al. (2020)
AdaBoost (J48, SVM, IB1, RF)	Meta Classifier	4	1	Asmitha and Vinod (2014); Darabian et al. (2020); Singh and Hofmann (2018); Tian et al. (2010)
Naive Bayes	Probabilistic Learning	4	0	Firdausi et al. (2010); Kim et al. (2020); Norouzi et al. (2016); Wan et al. (2020)
IB1	Lazy Classification	3	0	Islam et al. (2013); Norouzi et al. (2016); Tian et al. (2010)
Logistic Regression	Statistical	2	1	Kim et al. (2020); Norouzi et al. (2016)
BayesNet	Probabilistic Learning	1	0	Norouzi et al. (2016)

Table 2.3: Classification Algorithms used in Malware Detection

classification when the datasets were separated by CPU architecture and analysed individually. A greater improvement, in the same metrics, was seen for the evaluation involving malware family classification (i.e., *Recall*) for both classification algorithms (SVM and KNN) jumped by 6.34% and 5.11%, respectively.

2.4.5 Classification Performance Measures

Table 2.4 shows the most commonly used metrics to evaluate the performance of classification algorithms in malware detection (Ye et al., 2017). Singh and Hofmann (2018) proposed that the *Recall* measure was a better indicator of classification performance in malware detection as it measures how many of the actual positives (malware in this case) were identified as so. In comparison, *Accuracy* gives a general score for how well both malware and benign samples were classified as themselves. The authors argue that the risk of miscategorising malware (as benign) and thus letting it through, is a far greater problem then categorising a benign sample as malware.

Measure	Description
True Positive (*TP*)	Number of samples correctly classified as malware
True Negative (*TN*)	Number of samples correctly classified as benign
False Positive (*FP*)	Number of samples incorrectly classified as malware
False Negative (*FN*)	Number of samples incorrectly classified as benign
Precision	Percentage of samples classified as malware that are actually malware: $\frac{TP}{TP+FP}$
Recall	Percentage of actual malware which were classified as such: $\frac{TP}{TP+FN}$
Accuracy	Percentage of samples (malware and benign) which were correctly classified: $\frac{TP+TN}{TP+FP+TN+FN}$

Table 2.4: Measures of Classification-Based Malware Detection Performance (Singh & Hofmann, 2018; Ye et al., 2017)

Other measures such as F1-score, AUC, True Positive Rate (TPR) and False Positive Rate (FPR) have also been reported in a handful of studies (Belaoued et al., 2019; Huang et al., 2018). *TPR*[3], see eq. 2.3, has been referred to as the *detection rate* in terms of malware and intrusion detection and is considered equal to *Recall* (Belaoued et al., 2019; Liu et al., 2019). Another measure, *Area Under the Curve (AUC)*, provides an overall evaluation of the detection accuracy based on the *TPR* and *FPR* however it has only been reported in a few malware detection studies to date (Babaagba & Adesanya, 2019; Darabian et al., 2020; Huang et al., 2018; Khammas et al., 2019).

$$TPR = \frac{\text{number of correctly detected malware}}{\text{total number of malware}} \tag{2.3}$$

Class balance also can have an effect on the performance of a classification

[3]*TPR* is True Positive Rate (i.e., the percentage of actual positives which are correctly identified) (Kohavi & Provost, 2017)

algorithm, Aiswarya Mohan et al. (2020) observed that when the number of benign and malware samples in the training set were similar (191 benign and 153 malware) then the F1-score was higher (i.e., 0.91). When the authors changed the balance of the dataset (6753 benign and 153 malware), the F1-score dropped to 0.46 leading them to conclude that a balanced dataset improves the classification model. This observation is supported by the theory that classification algorithms have a tendency to predict the largest size class (i.e., in this case predict more benign outcomes (False Negatives) then malware) (Darabian et al., 2020).

2.5 Decision Trees and their role in Signature Generation

A Decision Tree algorithm splits a (training) dataset into separate areas (classes) using a top-down approach. At the top of the tree is the root node and off that, there are several intermediate nodes. All nodes incorporate a decision (e.g., is x greater than y?). Nodes are created through attribute selection (i.e., what is the best attribute to split the training samples into the proper classes (malware or benign)) and they all, ultimately, end in a result (leaf node). Figure 2.5 depicts an example decision tree. The most prominent decision tree algorithms are *ID3*, *C4.5*, *CART*, *CHAID* and *MARS* (Wikipedia, 2021c). *J48* is an open-sourced Java implementation of the *C4.5* decision tree algorithm developed by J Ross Quinlan (Wikipedia, 2021b) and has been commonly used as the decision tree classifier in studies (as per Table 2.3). *C4.5* is the successor to Quinlan's ID3 algorithm which was developed in 1986 (Quinlan, 1986). The main difference between *C4.5* and *CART* is that they use different criteria for splitting nodes. *C4.5* uses Entropy to pick features with the maximum *Information Gain* whereas *CART* uses *Gini Impurity* to split nodes. In Gini Impurity, the aim is to reduce the impurity of a node. For example, if all data points of a node point to the same class, then the node is said to be pure (and thus an end node) otherwise it is split (a decision is made).

Signature generation is frequently featured in malware detection studies (Alhanahnah et al., 2018; de Oliveira, Grégio, & Cansian, 2012; Mohammed, Aleisa,

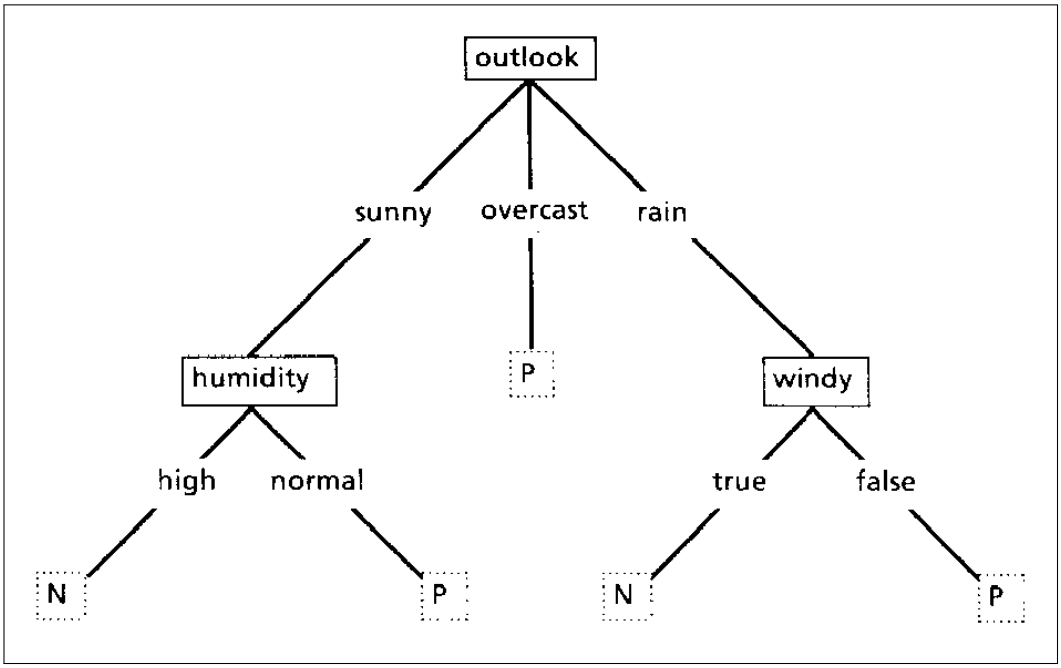


Figure 2.5: A Simple Decision Tree (Quinlan, 1986)

& Ventura, 2014; Soe et al., 2019) however only a handful of studies have used the output of the classification algorithm (*J48* Decision Tree in both cases) as a basis to generate signatures from (Mohammed et al., 2014; Soe et al., 2019). The visualisation, generated by a decision tree algorithm, lends itself to signature generation (i.e., each node corresponds to an attribute of a rule/signature and each leaf corresponds to an outcome of a signature). The path between the root node and the leaf node is equivalent to a rule/signature (Asmitha & Vinod, 2014).

In Soe et al. (2019), the authors generated a decision tree which included 24 leaves, 16 were ”attack” traffic leaves and 8 were ”normal”. The ”normal” leaves were discarded and the 16 ”attack’ leaves (and their root and intermediate nodes) were converted into signatures for a detection system. No results were published on the effectiveness of the signatures in detecting malware. In another study, researchers built a signature generator component which utilised the *C4.5* algorithm to generate signatures to detect zero-day polymorphic worms (Mohammed et al., 2014). Again, no results were published on the effectiveness of the signatures, as such it is not known how viable this approach is for zero-day attacks. Dinh et al.

(2019) argue that while Decision Tree algorithms can achieve high accuracy rates, they are a supervised learning algorithm and as such are not suitable for detecting zero-day attacks as they require a pre-labelled training dataset.

Decision Trees have shown themselves to be particularly accurate at classifying malware using behaviour-based features (e.g., System Calls). Yuxin et al. (2011) reported that the best overall detection performance (using system calls as features) was achieved by a Decision Tree classifier which exhibited higher Accuracy rate, a higher TPR, and a lower FPR (compared to three other classifiers). Firdausi et al. (2010) also reported that the *J48* classifier (in their study) achieved the best overall performance (*Recall* value of 95.9%, a *FPR* of 2.4%, a *Precision* of 97.3% and an *Accuracy* of 96.8%) when detecting malware using behaviour-based features (e.g., API Calls, System Calls). Decision Trees are also effective at classifying different types of attacks / malware, Soe et al. (2020) used the *J48* classifier to classify 9 different attacks types (Fuzzers, DoS, Backdoors, Worms etc.). Finally, several studies have cited that decision trees are simple and lightweight to implement (Aiswarya Mohan et al., 2020; Mohammed et al., 2014; Soe et al., 2019) which supports their suitability for IoT malware detection.

2.6 Conclusions

This chapter provided a background to the research that has been undertaken in the field of Malware Detection with a specific focus on its application to IoT. The first half of the review examined the prevalence of IoT devices in everyday life and discussed why their security weaknesses make them vulnerable to malware attacks. Different types of attacks were reviewed including what they target and exploit. The need for a lightweight malware detection approach due to the constrained resources on IoT devices was discussed next. While different solutions were proffered, there was a general consensus (among researchers) that Signature-Based detection can be considered a lightweight approach due to its lack of computational overhead. The second part of the review focuses in on the work that has been done on malware detection using data mining techniques. The different phases, sample collection, feature extraction/selection and classification, were examined in detail. After that, performance measures of classification algorithms

were analysed followed by a discussion on the most suitable classification algorithm for malware detection, namely Decision Trees. Finally, as this work relates to evaluating a signature generation approach for IoT Malware Detection, the role of Decision Trees for signature generation was considered.

As mentioned earlier in the chapter, employing Decision Trees to generate signatures has a number of advantages. The primary one being that signatures which are generated using a trained classification model, have already proved their effectiveness (in the classification test phase), prior to their inclusion in a detection system. Some compromises which most likely will have to be made when considering this approach for IoT malware is that 1) the number of signatures produced will need to be kept to a minimum so as to keep storage requirements low, 2) due to this reduced set of signatures, it is unlikely that anything beyond malware family detection will be possible and 3) as the detection will be signature-based, it is expected that only known malware samples will be detected.

Despite several studies showing that Decision Trees are simple to implement, are lightweight and have shown themselves to be particularly accurate at classifying malware based on behaviour-based features (e.g., System Calls), their inclusion in studies on IoT malware detection and signature generation have been limited. This fact may have been due to the lack of suitable IoT datasets (as discussed in section 2.4.1) but this is now changing with the likes of the *X-POT* and *IoTPoT* datasets being made publicly available. Also, research in this area appears to be in its infancy, for example, studies so far have been somewhat restricted, e.g., small number of IoT malware samples analysed or testing on non-IoT architectures (Abbas & Srikanthan, 2017), using a non-IoT dataset of network intrusion data as a part of signature generation (Sheikh, Rahman, Vikram, & AlQahtani, 2018) or using static analysis on malware binaries which means obfuscated malware cannot be analysed (Alhanahnah et al., 2018). Finally, the literature surveyed to date did not reveal a single study which incorporated all of the following aspects when evaluating a Decision Tree-Based Signature Generation approach for IoT Malware Detection:

1. Used a dataset compromising of IoT malware executables.
2. Carried out dynamic analysis using a sandbox targeted specifically at IoT CPU architectures.

3. Employed an automated experimental approach to find the optimal System Call *N*-gram size, top *N* features and decision tree depth for use in classification.
4. Tested the detection performance of the generated signatures.

The closest studies were Soe et al. (2019) and Belaoued et al. (2019). In Soe et al. (2019), the authors used network traffic instead of System Calls and the generated signatures were not tested. In Belaoued et al. (2019), there was some similarities in that they used TF-IDF to identify the most significant features and they also generated YARA signatures. However, their study was focused on Windows malware not IoT malware and they did not use any classification models to generate their signatures.

This study will combine all of the above aspects with the aim to address the following research questions:

- What combinations of *N*-gram size, number of features and decision tree depth result in the best classification performance?
- What combinations of *N*-gram size, number of features and decision tree depth result in a high detection rate for IoT malware?
- To what extent is classification performance an indicator of the detection rate for IoT malware?
- What is the minimum number of System Call-based signatures which can achieve a high detection rate for IoT malware?

Chapter 3

Design

3.1 Introduction

This chapter discusses the approach taken to address the following aim:

- To design, based on existing literature, a method which uses Data Mining techniques to elicit relevant behavioural features from IoT Malware and Benign samples which can be used to detect malware.

An overview of designed approach is shown in Figure 3.1. The subsequent sections describe each phase of the framework in detail.

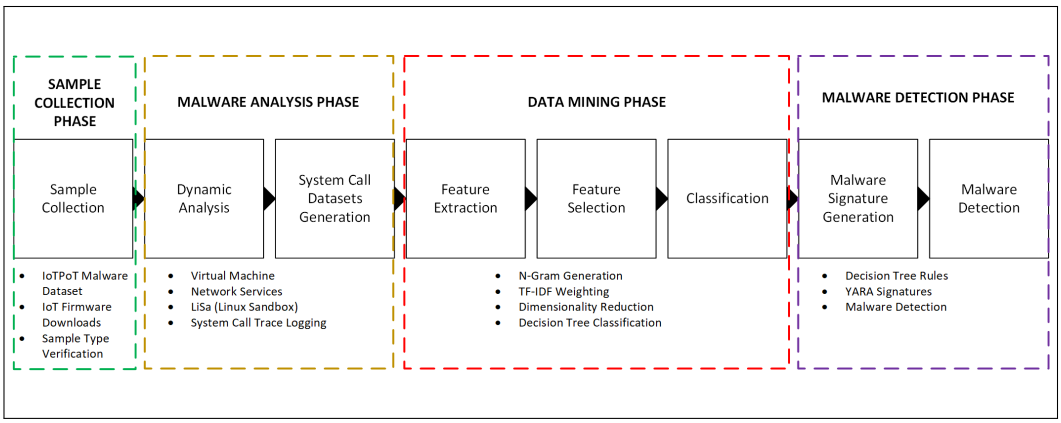


Figure 3.1: Decision Tree-Based Signature Generation Framework

3.2 Sample Collection Phase

The first step of any malware detection project is to collect a suitable sample set of both malware and benign samples. As the aim was to extract behavioural features from IoT samples, ELF executable files were selected. This was an IoT focused study, so malware and benign executables typically found on IoT devices (i.e., ELF executable files) were required. As mentioned in section 2.4.1, there have been a limited number of IoT malware executables datasets publicly available and to date, no IoT benign executables datasets.

Benign Executables Options for gathering appropriate IoT benign executables are either, obtaining executable files direct from the operating systems on IoT devices or by utilising firmware downloads (for IoT devices) as a sample source. The first option is not viable unless the researcher has access to a large set of different IoT devices and even then, extracting executables from each individual device is going to be a slow, protracted process. The second option is typically the favoured approach among researchers as it does not necessitate the need to procure any IoT hardware. Also with firmware analysis tools such as *BinWalk* (`https://github.com/ReFirmLabs/binwalk`) being freely available and straightforward to use, extracting executable files from firmware can be a relatively trivial task. The collection of firmware downloads can also be automated using downloading scripts (Wan et al., 2020). For this study, benign executables were scraped, using *BinWalk*, from downloaded IoT firmware produced by manufacturers including D-Link, Wyze, Ezviz, IDIS global, Netgear and Reolink.

Malware Executables In terms of gathering malware samples, there are three potential options. The first (and least complicated) is to acquire a suitable dataset that already exists. The second is to download malware samples direct from virus sharing websites (e.g., Virus Share - `https://virusshare.com/`) or alternatively, using malware repositories like *theZoo* - `https://github.com/ytisf/theZoo`. The final option is to set up a honeypot to capture the malware directly. This option was adopted by a team from Yokohama National University, Japan

in 2020 and the result of which was *X-POT*, a publicly available[1] dataset of malware executables (Kato et al., 2021). As this dataset includes malware executables targeting popular IoT CPU architectures, it was the most suitable option for this work.

Once the samples (for this study) were collected, their maliciousness (or not in the case of the benign samples) needed to be verified. VirusTotal (`https://www.virustotal.com/gui/`) offers a file and URL checking service to determine if they are suspicious or not. The API[2] for this service was utilised and samples were retained or discarded based on the results returned by VirusTotal. For the retained malware samples, their results from VirusTotal were run through a second tool, *AVClass2*[3] in order to determine the malware family each sample belonged to.

3.3 Malware Analysis Phase

Dynamic Analysis was used to elicit behavioural features from the IoT samples. The main difference between Dynamic and Static analysis is that in the former, malware needs to be run in order to analyse it whereas in the latter, it does not. The reasons for using one type of analysis over the other have already been discussed in section 2.4.2 but as a brief reminder, dynamic analysis is the preferred method for extracting System Calls. When running malware samples, a secure and controlled environment is recommended. This can be a physical host or a virtual machine on an air-gapped network, both have their advantages and disadvantages.

Physical Host With a physical host on an air-gapped network, the environment is more realistic however a backup image needs to be taken before any analysis activity and the host needs to be restored (back to that image) post analysis activity. This can be quite a long, drawn out process especially if a large number of malware samples need to be analysed or the same tasks need to be repeated multiple times (Sikorski & Honig, 2012b).

[1] Available by request at `https://sec.ynu.codes/iot/available_datasets`

[2] virustotal-search.py - `https://github.com/DidierStevens/DidierStevensSuite/blob/master/virustotal-search.py`

[3] AVClass2 - `https://github.com/malicialab/avclass/tree/master/avclass2`

Virtual Machine (VM) Alternatively, virtual machines offer a quick, repeatable base which can be used to analyse a sample and then reverted back to, once the analysis is complete. This approach means a clean, consistent state is used for each sample without introducing any variance. The main negative with using virtual machines is that some malware can detect they are being run in a virtual environment and may alter their behaviour in those instances to avoid detection (Sikorski & Honig, 2012b).

Network Services Regardless of whether a physical host or a virtual machine is used for malware analysis, some malware require an internet connection as part of their functionality. They may need to look for updates or send/receive commands to a Command and Control (C&C) server. As offering a connection out to the internet is not advised and can result in malware being unintentionally spread, network services should be simulated instead. *iNetSim* (`https://www.inetsim.org/`) is software which simulates common internet services (HTTP/HTTPS, DNS, FTP etc.) and is commonly used in malware analysis in lab environments (Monnappa, 2018). A typical malware lab setup is one or more Virtual Machines posing as victim machines (Windows VM in Figure 3.2) which are then networked to a single network services machine ((Linux VM in Figure 3.2) connected via a host-only connection. In this study, Virtual Machines created using VMWare Workstation were used to host both a victim machine and network services. *iNetSim* was used to simulate network services.

Malware Sandbox While dynamic malware analysis can be performed using a custom setup built by combining different tools, there is an all-in-one solution which is more commonly used namely, a sandbox (Sikorski & Honig, 2012a). Sandboxes typically offer automated analysis (static, dynamic and network traffic) and the most common ones are *Cuckoo*[4] (leading open source automated malware analysis sandbox that supports Windows, macOS, Linux and Android), *Limon*[5] (Linux sandbox for analysing malware, written in Python), *REMux*[6] (Linux toolkit

[4]Cuckoo - `https://cuckoosandbox.org/`
[5]Limon - `https://github.com/monnappa22/Limon`
[6]REMux - `https://remnux.org/`

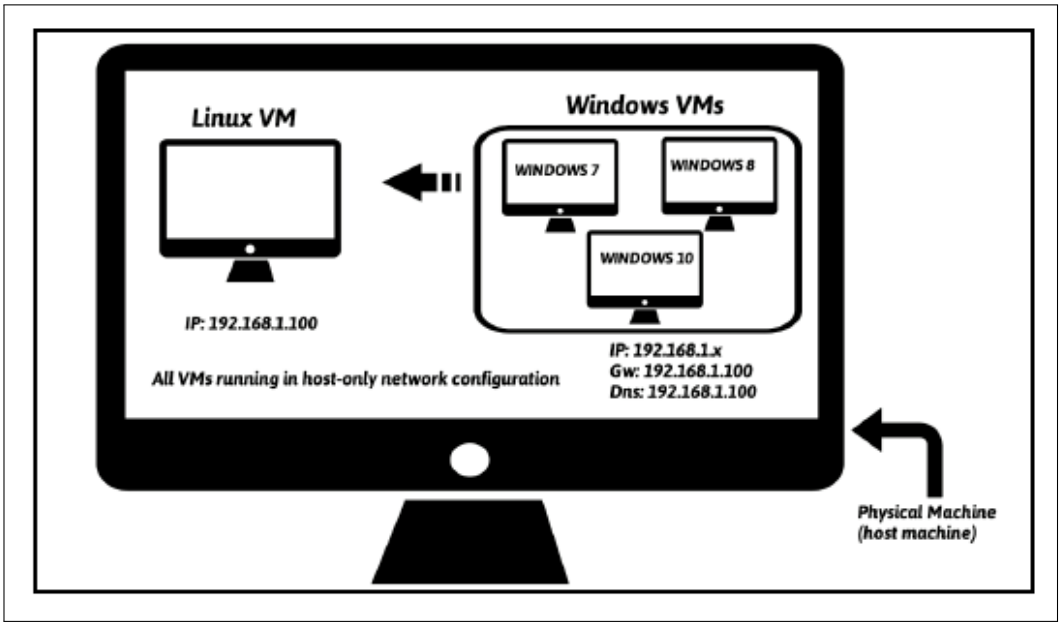


Figure 3.2: Example of a Malware Analysis Lab Configuration (Monnappa, 2018)

for reverse-engineering and analysing malicious software) and *Detux*[7] (multi-platform Linux sandbox that supports the following CPUs: x86, x8664, ARM, MIPS and MIPSEL). *LiSa* (`https://github.com/danieluhricek/LiSa`), a new addition to the group, is a multiplatform Linux sandbox which provides automated malware analysis on various CPU architectures (Uhvrivcek, 2019). Similar to *Detux*, *LiSa* supports CPU architectures (x8664, i386, ARM, MIPs and AArch64) commonly found on IoT devices but its main advantage over *Detux* is that it provides dynamic analysis. *Detux* only provides static and network analysis. Another benefit of *LiSa* is that it provides a custom implementation of *ptrace* which prevents a common anti-debugging technique. Without this custom code, some samples could detect they were being traced (for System Call information) and prevent *strace*[8] from running. For these reasons, *LiSa* was chosen as the most appropriate sandbox for running the IoT malware (and benign) executables in this study.

[7]Detux - `https://github.com/detuxsandbox/detux`
[8]strace - `https://man7.org/linux/man-pages/man1/strace.1.html`

System Call Datasets Generation LiSa generates a *report.json* file which includes the results of all of the configured analysers namely, Static, Dynamic and Network. For the purpose of this study, only the Dynamic results were used as they contained each executable's System Calls, see Figure 3.3 for an example.

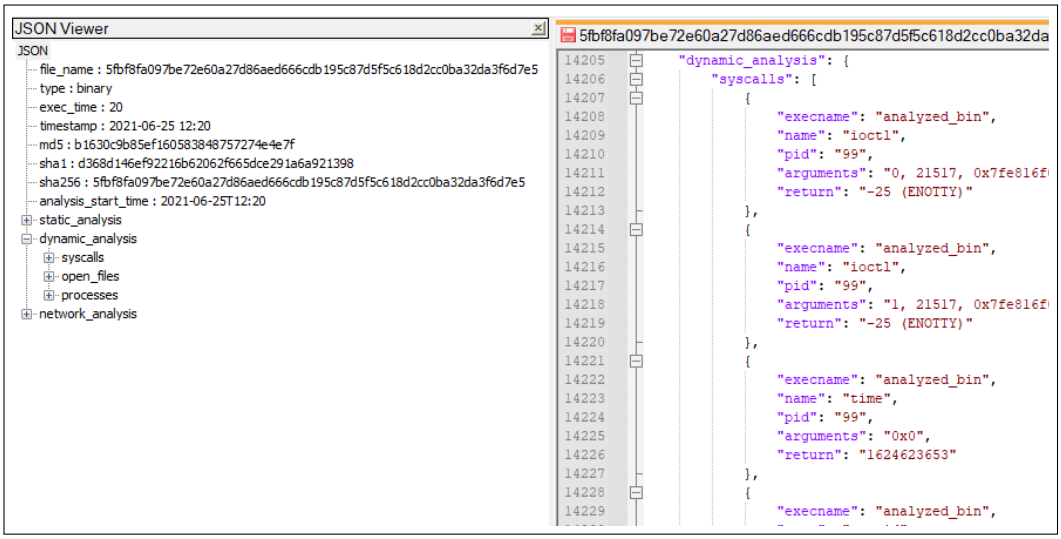


Figure 3.3: Example of the Dynamic Analysis section of a *report.json*

As per other studies, the input and output arguments of the system calls are discarded and only the system calls themselves were kept for further analysis (Abbas & Srikanthan, 2017; Belaoued et al., 2019; Shobana & Poonkuzhali, 2020). These argument less system calls were stored in .dat files (dataset files) and used in both the Data Mining and Malware Detection phases (see Figure 3.4 for an example).

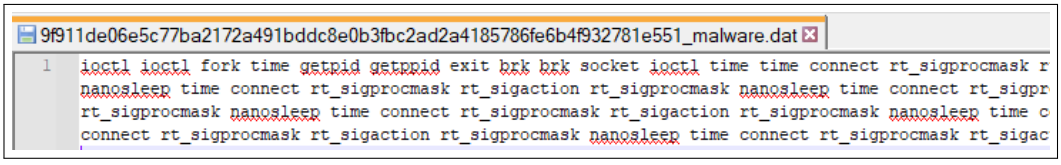


Figure 3.4: Example of the System Call dataset generated from a *report.json*

3.4 Data Mining Phase

Figure 3.5 is a reminder of the different stages in the Data Mining Phase.

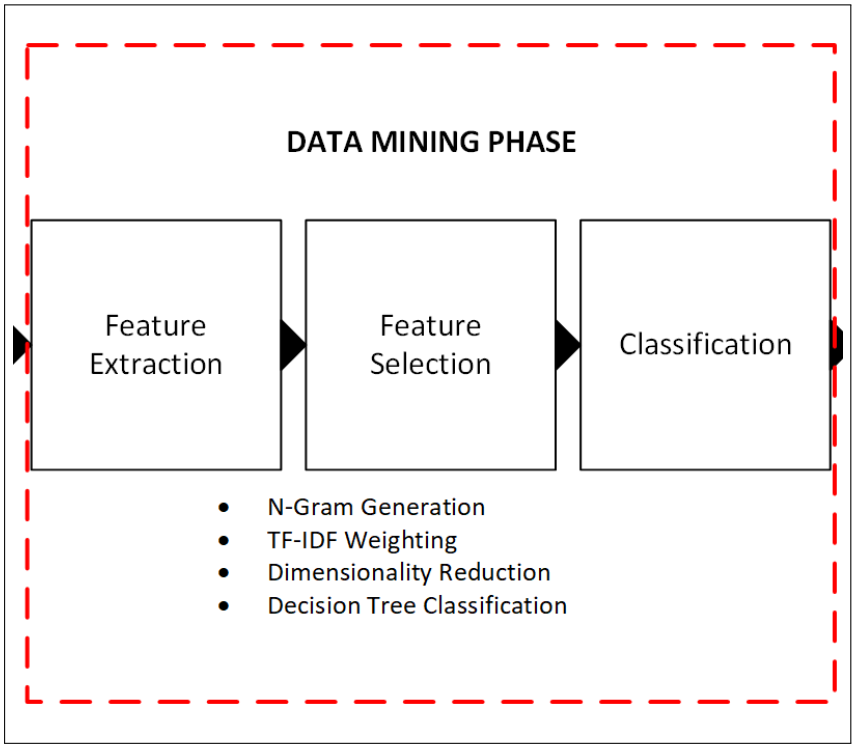


Figure 3.5: Data Mining Phase Overview

Feature Extraction and Selection Tables 2.1 and 2.2 in section 2.4.3 showed the most frequently used Feature Extraction and Selection methods used to date in the literature, namely *N*-grams and *TF-IDF*, respectively. The use of these methods is necessary because the raw data (System Call Trace Logs) produced by the Malware Analysis Phase can be quite extensive and can also include a lot of data which is redundant/adds no value. *N*-grams are used to elicit features with context, for example, a trigram (n =3) shows a brief sequence of System Calls (i.e., what came before and after a single System Call). A small n value can result in a large number of n-grams which are not necessarily of equal importance. *TF-IDF* aids dimensionality reduction of these *N*-gram features by assigning weights to each of them which is relevant to their importance across the whole dataset. Figure 3.6 shows an example of raw System Calls converted to trigrams (n=3) with their *TF-IDF* weights normalised to 0 or 1 (any value greater than 0 becomes 1).

	A	B	C	D	E	F	G
1	access openat openat	brk brk brk	brk brk getpid	brk brk time	brk getpid rt_sigaction	brk mmap2 access	brk socket fcntl
2	1	0	1	0	1	1	(
3	1	0	1	0	1	1	(
4	1	0	1	0	1	1	(
5	1	0	1	0	1	1	(
6	0	0	0	0	0	0	(
7	1	0	1	0	1	1	(
8	0	0	0	0	0	0	(
9	0	0	0	1	0	0	:
10	0	0	0	0	0	0	(
11	1	0	1	0	1	1	(
12	0	0	0	1	0	0	(

Figure 3.6: Example of a sample's System Calls post Feature Extraction and Selection

While the use of both methods has been reported frequently in Malware Detection studies, their parameters (optimal *N*-gram size or maximum number of features to select) have not. As such, their inclusion in this study meant that it was necessary to design a flexible Data Mining phase which could accommodate variable parameters. As such, each of the sub-functions (boxes in each phase in Figure 3.5) were developed as individual or combined Python modules. The *Python* programming language is a common language used in data science and as such, it comes with a number of widely-used data mining/machine learning libraries (e.g., *pandas*[9] (open source data analysis and manipulation tool), *NumPy*[10] (package for scientific computing in Python) and *scikit-learn*[11] (Machine Learning library for Python)).

Malware Classification *WEKA*[12], a tool for Decision Tree Classification (and data mining tasks in general), has been used frequently in Malware Classification (Aman, Saleem, Abbasi, & Shahzad, 2017; Babaagba & Adesanya, 2019; de Oliveira et al., 2012; Islam et al., 2013; Soe et al., 2019). However, for this study it was deemed unsuitable as it could not be easily incorporated into a data pipeline (Data Mining phase to the Malware Detection phase) without a reasonable amount of additional manual work (i.e., to generate WEKA compatible datasets). It also

[9]pandas - `https://pandas.pydata.org/`
[10]NumPy - `https://numpy.org/`
[11]scikit-learn - `https://scikit-learn.org/stable/`
[12]WEKA - `http://old-www.cms.waikato.ac.nz/~ml/weka/`

posed automation challenges, for example, with a solution developed in Python, the running (of the solution) can be automated using *Shell scripts*.

Scikit-learn provides a Decision Tree module[13] based on the *CART* algorithm which can be used for classification and regression. This was the preferred solution as it allowed for a full end-to-end automated framework and earlier modules (i.e., Feature Extraction and Selection), were already utilising *scikit-learn*. As such some practical knowledge (in this library) had already been gained prior to the classification task. Other reasons for choosing *scikit-learn* was that in addition to providing classification models, it also has a rich API for classification metrics and easily integrates with other open source packages (e.g., *Graphviz*[14]) for plotting decision trees. Figure 3.7 shows an decision tree plotted during one of the experimental runs within this study. A Python module was developed to perform classification on the feature set produced by the Feature Extraction/Selection module. This module had a number of outputs including a set of classification results (e.g., Accuracy, Recall, Precision and a Confusion Matrix graphic), a Decision Tree graphic and a set of Decision Tree rules.

[13]Decision Tree module - `https://scikit-learn.org/stable/modules/classes.html#module-sklearn.tree`

[14]Graphviz - `https://graphviz.org/`

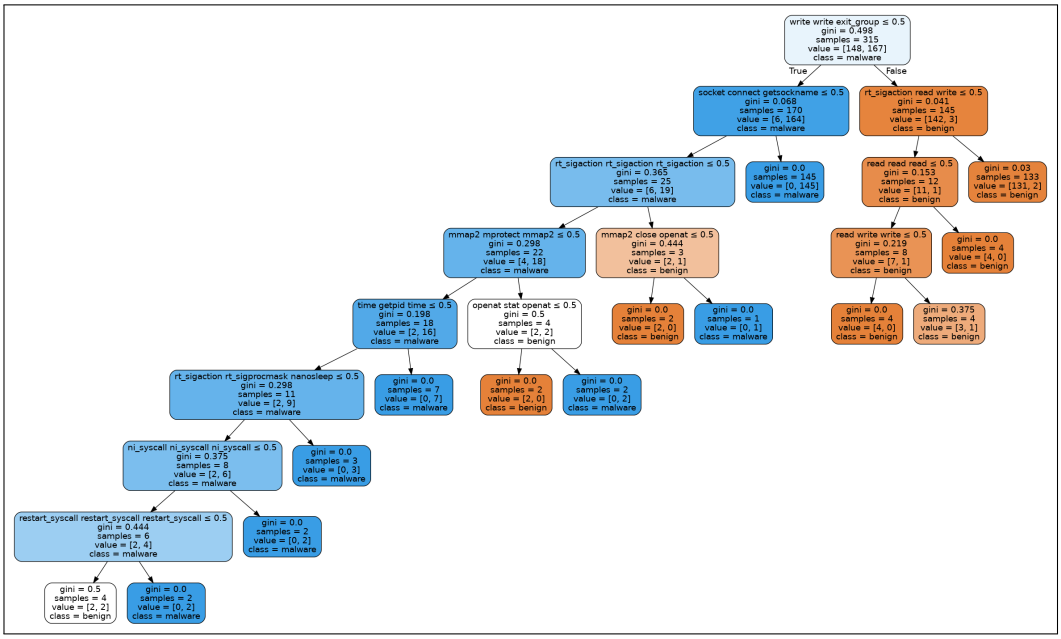


Figure 3.7: Example of a plotted Decision Tree

3.5 Malware Detection Phase

This phase comprised of two distinct functions namely, Signature Generation and Malware Detection.

Signature Generation The Decision Tree rules (from the classification phase) were a series of 'If...Else' statements that formed the basis for the automatic generation of YARA signatures, see Figures 3.8 and 3.9 for examples of both. *YARA*[15] is a powerful malware classification/detection tool which determines if a file, folder or process is malware (or not) based on a set of signatures (textual or binary patterns). Due to its widespread use by Malware researchers (Alhanahnah et al., 2018; Aman et al., 2017; Belaoued et al., 2019) and the fact that it has a Python API[16], YARA was a logical choice for inclusion in this study. A Python module which takes a file containing Decision Tree rules as an input and outputs a YARA signatures file (.yar) was developed.

[15]YARA - `https://github.com/VirusTotal/yara`
[16]YARA Python API - `https://yara.readthedocs.io/en/stable/yarapython.html`

```
if (write write write <= 0.5) and (socket connect getsockname > 0.5) then class: malware
if (write write write > 0.5) and (rt_sigaction read write > 0.5) then class: benign
if (write write write <= 0.5) and (socket connect getsockname <= 0.5) and (brk brk brk <= 0.5) and (getpid time getpid <= 0.5)
and (restart_syscall restart_syscall restart_syscall <= 0.5) and (ni_syscall ni_syscall ni_syscall <= 0.5) and (stat openat stat
 <= 0.5) and (time getpid getppid <= 0.5) then class: benign
if (write write write <= 0.5) and (socket connect getsockname <= 0.5) and (brk brk brk <= 0.5) and (getpid time getpid > 0.5)
then class: malware
if (write write write <= 0.5) and (socket connect getsockname <= 0.5) and (brk brk brk <= 0.5) and (getpid time getpid <= 0.5)
and (restart_syscall restart_syscall restart_syscall > 0.5) then class: malware
if (write write write > 0.5) and (rt_sigaction read write <= 0.5) and (read write write > 0.5) and (read read read > 0.5) then
class: benign
if (write write write > 0.5) and (rt_sigaction read write <= 0.5) and (read write write > 0.5) and (read read read <= 0.5) then
class: benign
if (write write write > 0.5) and (rt_sigaction read write <= 0.5) and (read write write <= 0.5) then class: benign
if (write write write <= 0.5) and (socket connect getsockname <= 0.5) and (brk brk brk <= 0.5) and (getpid time getpid <= 0.5)
and (restart_syscall restart_syscall restart_syscall <= 0.5) and (ni_syscall ni_syscall ni_syscall > 0.5) then class: malware
```

Figure 3.8: Example of a Decision Tree Rule

```
rule malware_3 {
    meta:
        author = "FWalsh"
        date = "30/07/2021"
    strings:
        $syscall_ngram_1 = "write write write"
        $syscall_ngram_2 = "socket connect getsockname"
        $syscall_ngram_3 = "brk brk brk"
        $syscall_ngram_4 = "getpid time getpid"
        $syscall_ngram_5 = "restart_syscall restart_syscall restart_syscall"

    condition:
        (not $syscall_ngram_1) and (not $syscall_ngram_2) and (not $syscall_ngram_3) and (not $syscall_ngram_4) and
        $syscall_ngram_5 and (filesize > 0)
}
```

Figure 3.9: Example of an Auto-Generated YARA Signature

Malware Detection The final part of the study involved using the generated YARA signatures to perform malware detection. For this task, the system call datasets generated in the Malware Analysis Phase were used an input into a Malware Detection module (developed in Python). This module used the YARA API to first compile the signatures to confirm they were syntactically correct and then secondly to classify each of system call datasets as benign or malware. Figure 3.10 shows an example of the output from the detection phase.

```
/home/datausr/01_DataAnalysis/02_SysCall_Datasets/14a29aa7ac19f4f0f9a61a088373c152a417613e632c2da14bc45be37bc6bafe_benign.dat
does not match any YARA signatures.
/home/datausr/01_DataAnalysis/02_SysCall_Datasets/41d9f77048fe832d3368652f633ad62a98c080bcb37a9ba66c882b058576edde_benign.dat
does not match any YARA signatures.
/home/datausr/01_DataAnalysis/02_SysCall_Datasets/676fc2be3dff7371225bb63c08cc1a96a89630082adad29dba3b817aa93ba332_benign.dat
does not match any YARA signatures.
/home/datausr/01_DataAnalysis/02_SysCall_Datasets/05b6a7855479ec9dec8bb2e159670cea23dbd7400ee94bb9f368488d9e169c13_benign.dat
does not match any YARA signatures.
/home/datausr/01_DataAnalysis/02_SysCall_Datasets/2cab8db3d418e08011ae033cfd4d00e22736668bacc1f7bf577b5771373e2bfb_malware.dat
 matches the following YARA rule(s): [malware_1]
/home/datausr/01_DataAnalysis/02_SysCall_Datasets/40d6af30a0507c86c6719d5fbfa2fb61c451dd9d61f2fad27b9ec09c4e23b136_benign.dat
does not match any YARA signatures.
/home/datausr/01_DataAnalysis/02_SysCall_Datasets/4a0e2670bcf5b572a91f66060da0e79bf3dcc4f3af5020d724b5a6b6c1350a17_malware.dat
 matches the following YARA rule(s): [malware_1]
/home/datausr/01_DataAnalysis/02_SysCall_Datasets/74d8afd4d4e317dd75ab66d5a01ac1172800e39a18a429d1c1997b330573f834_malware.dat
 matches the following YARA rule(s): [malware_1]
```

Figure 3.10: Example of the output of the Malware Detection Phase

3.6 Experiment Design

The experimental phase starts with Feature Extraction and concludes with Malware Detection. The previous phases, Sample Collection and Malware Analysis, were only performed once and as such, are considered prerequisite phases to the main experiments. Figure 3.12 shows the experimental flow of the activities and their input/outputs. This flow shows an end-to-end experimental run which is repeated *x* times based on the number of runs configured in a control file (see Figure 3.11). The parameters in the file were as follows:

- **run_num** The experimental run number
- **fes_ngram** The size of the *N*-gram to use during Feature Extraction, range: 1-6
- **fes_topn** The top number of features to retain (for classification) as part of Feature Selection, range: 5, 10, 20, 50 and 100
- **mc_tree_depth** The max depth of the Decision Tree to use in Malware Classification, range: 2-8

This design meant that a substantial set of parameter combinations could be tested in order to determine optimal values for *N*-gram size, top *n* features and max tree depth. Also, with regards to the tree depth parameter specifically, it has been noted that decision trees have a tendency to overfit so the use of max tree depth helped to avoid this problem by generalising the data better. The overall running of the experimental flow was triggered by a *Shell* script. Results, including details of the input parameters used, from each run were collected in a single log file for post-analysis.

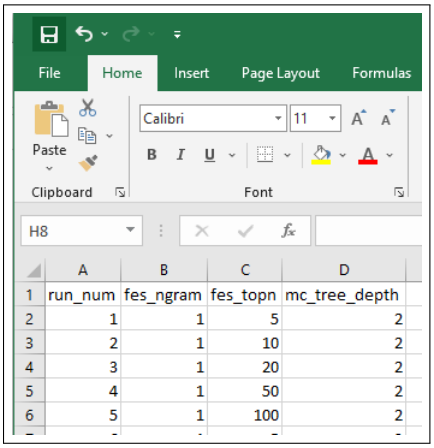


Figure 3.11: Experiment Parameters Configuration File

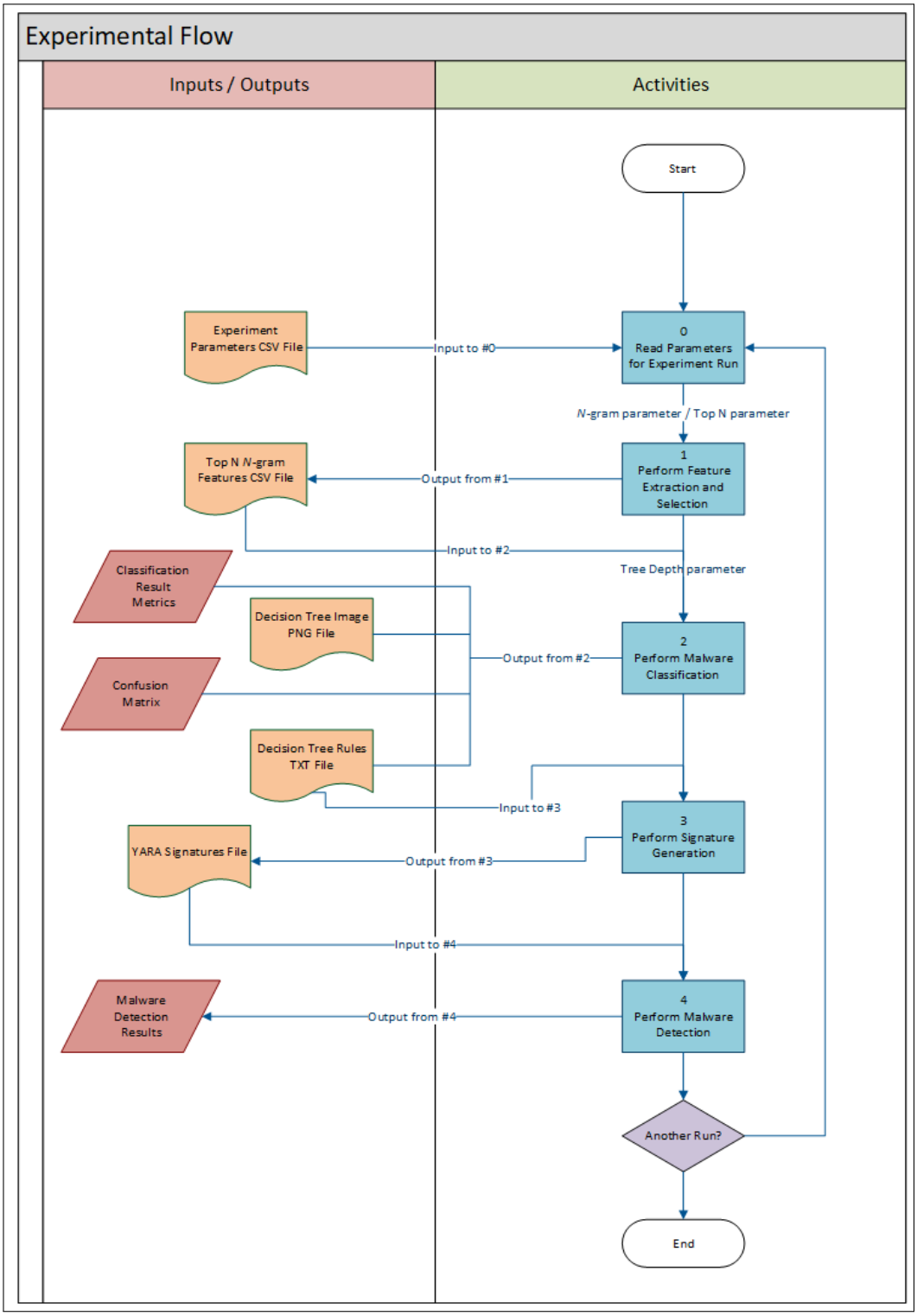


Figure 3.12: Experimental Flow

3.6.1 Experimental Results Design

The results gathered in the experiments fell into two categories, classification results and detection results. Considering classification results first, the experiments measured standard classification metrics (Precision, Recall, F1-Score, Accuracy) and recorded them in a report (see Figure 3.13). As shown below, some additional metrics were reported namely, *macro avg* and *weighted avg*. These averages were not used in the evaluation of this study as they are not typically reported in studies of this kind and do not appear to add any additional insight over and above the standard metrics. Finally a confusion matrix showing the True vs Predicted labels (classes) was also produced (see Figure 3.14).

```
8452  ################### - RESULTS - ###################
8453
8454  Classification Report:
8455
8456                precision    recall  f1-score   support
8457
8458       benign     0.9412    1.0000    0.9697        64
8459      malware     1.0000    0.9437    0.9710        71
8460
8461     accuracy                         0.9704       135
8462    macro avg     0.9706    0.9718    0.9704       135
8463  weighted avg     0.9721    0.9704    0.9704       135
8464
8465  ##################################################
```

Figure 3.13: Classification Report

The second type of results recorded in the experiments were signature-based detection results. Previously, Figure 3.10 showed the outcome of a sample being classified (using the generated signatures) using YARA. Each of these outcomes were collated into a set of overall results and rates (e.g., True Positive Count, True Positive (TPR), False Positive Count, False Positive (FPR)) were calculated (see Figure 3.15).

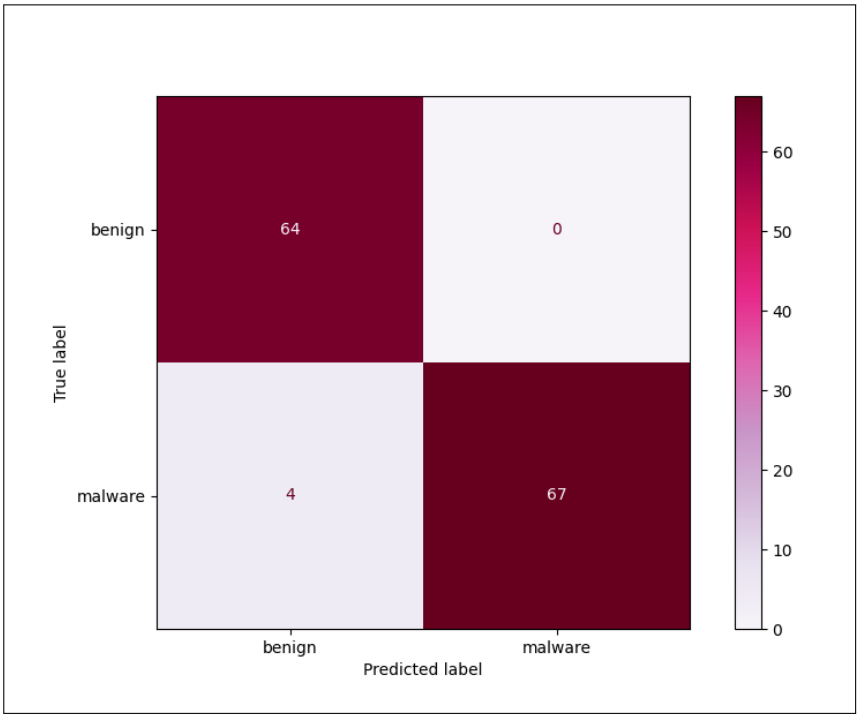


Figure 3.14: Confusion Matrix

```
8496  ################### - RESULTS - ####################
8497  Actual Malware Count: 238 | Actual Benign Count: 212
8498
8499  TP (True Positive) Count: 229, TPR: 96.22%
8500  TN (True Negative) Count: 212, TNR: 100.00%
8501  FP (False Positive) Count: 0, FPR: 0.00%
8502  FN (False Negative) Count: 9, FNR: 3.78%
8503  ##########################################################
```

Figure 3.15: Collated Malware Detection Results

3.7 Conclusions

As stated at the beginning of this chapter, the aim was to:

> *design a method which uses Data Mining techniques to elicit relevant behavioural features from IoT Malware and Benign samples which can be used to detect malware*

This was discussed, first, in terms of implementation design, then experimental

design and lastly, evaluation design. The implementation design needed two key factors built-in namely, parameter exploration and repeatability of experimental runs. Exploration of optimal values for feature extraction/selection and classification was necessary because there is currently no general consensus (in the literature) on what those values should be. Secondly, repeatability meant that a large number of experimental runs could be executed in order to explore and ultimately determine those optimal values for this study. Related studies such as Abbas and Srikanthan (2017); Belaoued et al. (2019); Shobana and Poonkuzhali (2020) were referenced for tool suggestions and where alternatives options were sought, an explanation of the reasons why were given. Next, an overview of the experimental flow design was presented and the type of experiment parameters used were discussed. Finally, the chapter concluded with a discussion of the evaluation design which included a breakdown of the results and outputs which were produced by the experiments.

Chapter 4

Implementation & Results

4.1 Introduction

This chapter discusses the environment setup, both in terms of hardware used and the virtual machine configuration, for this study. It lists all of the tools used and provides specific details on where they can be sourced. After that, it discusses what samples were used in the study, how they were collected, analysed and how the final set was determined. A detailed description of the end-to-end process is provided including the steps followed to perform dynamic analysis on each of the samples and how the System Call dataset was generated. The experimental run is illustrated using an overview diagram and each stage is discussed in detail with explanations of the code that was developed, where relevant. This chapter concludes with a presentation of the results and some initial analysis for each of the evaluation phases namely, classification and detection.

4.2 Environment Setup

The environment used in this study consisted of a single laptop and multiple virtual machines. With the exception of some data analysis in Microsoft Excel on the laptop, all other practical work was carried out in the virtual machines. This setup was chosen as virtual machines provide a contained environment for safe execution of malware samples and as these particular ones were running Linux

OS, the power of *grep* could be leveraged for file analysis.

4.2.1 Laptop Configuration

The virtual machines used in this study were all hosted on a laptop with the specifications listed in Table 4.1. During the execution of the malware samples (inside a virtual machine), the laptop was set to Airplane mode with its wireless adapter also disabled as a precaution.

Specification Type	**Details**
Processor	11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
Memory	16.0 GB
Operating system	Windows 10 Pro
Storage	1 TB

Table 4.1: Laptop Specifications

4.2.2 Virtual Machine Configuration

As per the design discussed in section 3.3, two virtual machines (VM) were set up with host-only connections in *VMware Workstation 16 Pro*. This type of connection creates an isolated virtual network between the VM and the physical host via a virtual network adapter that appears on the host machine (laptop). For security reasons, the virtual network adapter was removed by unticking **Connect a host virtual adapter to this network** in the *Virtual Network Editor* in VMware Workstation 16 Pro which meant there was no longer a connection between the VM and the host.

In 'host-only' mode, while VMs have no internet connection, they can communicate with each other. As such, one of the VMs was setup to simulate NetworkServices using *iNetSim*. Details of the virtual machines which were implemented are listed in Table 4.2.

Machine Name	**Machine Function**	**Operating System**	**IP Address**	**Default Gateway**
VictimHost	Dynamic analysis of samples	Pop!_OS 20.10	10.0.0.11	10.0.0.60
NetworkServices	Default Gateway / DNS Server	Pop!_OS 20.10	10.0.0.60	n/a
DevMachine	Script Development and Experimental Runs	Pop!_OS 20.10	n/a	n/a

Table 4.2: Details of Virtual Machines

Some malware require an internet connection during their execution, as such all network traffic (if present) was routed from the VictimHost (where malware samples were executed) to the NetworkServices VM. This routing was made possible by setting the IP address of the NetworkServices VM (10.0.0.60) as the Default Gateway on the VictimHost VM. Additionally, the */etc/resolv.conf* configuration file on the VictimHost was updated with *nameserver 10.0.0.60* to ensure that all domain name requests resolved to the NetworkServices VM. Figure 4.1 shows a ping from the VictimHost to *google.com* resolving to the NetworkServices VM, confirming that is acting as a DNS server

```
(base) datausr@alpha:~$ ip a show ens33
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 00:0c:29:fd:96:89 brd ff:ff:ff:ff:ff:ff
    altname enp2s1
    inet 10.0.0.11/24 brd 10.0.0.255 scope global ens33
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fefd:9689/64 scope link
       valid_lft forever preferred_lft forever
(base) datausr@alpha:~$ ping google.com
PING google.com (10.0.0.60) 56(84) bytes of data.
64 bytes from www.inetsim.org (10.0.0.60): icmp_seq=1 ttl=64 time=0.401 ms
64 bytes from www.inetsim.org (10.0.0.60): icmp_seq=2 ttl=64 time=0.764 ms
64 bytes from www.inetsim.org (10.0.0.60): icmp_seq=3 ttl=64 time=0.756 ms
64 bytes from www.inetsim.org (10.0.0.60): icmp_seq=4 ttl=64 time=0.840 ms
64 bytes from www.inetsim.org (10.0.0.60): icmp_seq=5 ttl=64 time=0.883 ms
64 bytes from www.inetsim.org (10.0.0.60): icmp_seq=6 ttl=64 time=0.683 ms
^C
--- google.com ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5012ms
rtt min/avg/max/mdev = 0.401/0.721/0.883/0.156 ms
(base) datausr@alpha:~$
```

Figure 4.1: Domain Name Resolution Example

Figure 4.2 shows the full list of NetworkServices, made available by iNetSim on the NetworkServices VM, which are accessible by the VictimHost VM. These services were discovered using an aggressive *Nmap* scan. This type of scan assumes that the machine performing the scan is on a reasonably fast (and reliable) network and so speeds it up.

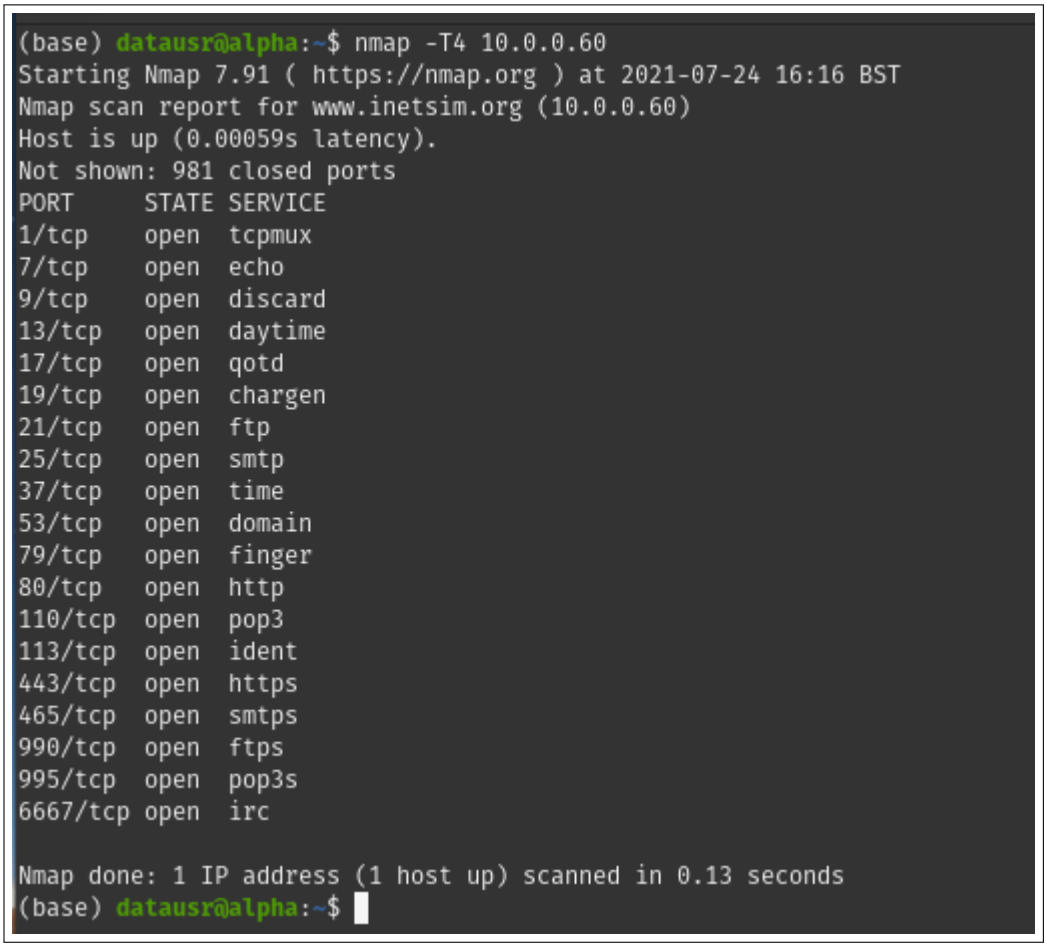


Figure 4.2: Aggressive (T4) Nmap scan

4.3 Tools

A combination of tools and scripts were used throughout the study, their details are listed in the order they were used in Table 4.3.

Tool Name	Tool Type	Function	Source
virustotal-search.py	Python Script	Performs VirusTotal Searches	See Section 3.3 for URL
csv_to_json_convertor.py	Python Script	CSV to JSON Convertor	See Appendix B.1
avclass2_labeler.py	Python Script	Malware Sample Labelling Tool	See Section 3.2 for URL
BinWalk	GitHub Project	Firmware Analysis Tool	See Section 3.2 for URL
LiSa	GitHub Project	Linux Multi-Platform Sandbox	See Section 3.3 for URL
dataset_generator.py	Python Script	Dataset File (.dat) Generator	See Appendix B.2
feature_extraction_selection.py	Python Script	Performs Feature Extraction and Selection	See Appendix B.3
malware_classification.py	Python Script	Performs Malware Classification	See Appendix B.4
signature_generator.py	Python Script	YARA Signature Generator	See Appendix B.5
malware_detection.py	Python Script	Performs Malware Detection	See Appendix B.6
YARA	GitHub Project	Malware Detection Tool	See Section 3.5 for URL
main.sh	Shell Script	Experimental Run Automation	See Appendix B.7

Table 4.3: Tools and Scripts

4.4 Sample Collection & Analysis

There were two types of samples used in this study namely, Malware and Benign. All samples had a file type of ELF executable, reasons for selecting this file type have already been discussed in section 3.2 so they will not repeated here.

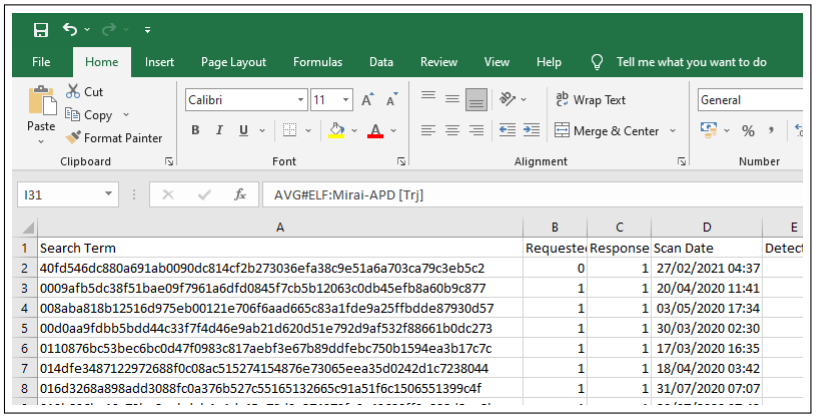
4.4.1 Malware Samples

As mentioned in section 3.2, a collection of malware executables were sourced from an existing dataset called *X-POT*. A team at Yokohama National University provided access to their SFTP server and the dataset was downloaded from there. On initial inspection of the dataset, there were two main sample sets, one from September 2020 and one from December 2017. The newer samples were used in this work. The initial total of this set was 1,730 samples but this number was reduced to 1,228 once files with a file type other than ELF executable were excluded. Each file had already been named using their SHA256 hash so any duplicate files that were found were also removed, this brought the overall total down to 1,168 malware samples.

VirusTotal The next step was to gather a list of filenames (SHA256 hashes) and submit these to VirusTotal to check their maliciousness. *virustotal-search.py* was used to submit each hash to VirusTotal (see Code Listing 1 for an example) and a CSV file containing VirusTotal's responses was generated (see Figure 4.3 for an example). Out of the 1,168 hashes that were submitted: 883 were flagged as malicious, 284 were not found and 1 was not detected by any AntiVirus vendor.

```
#!/bin/bash
~/00_Scripts/virustotal-search.py malware_hashes -e md5,sha1,sha256
```

Listing 1: VirusTotal Search Command



	A	B	C	D	E
1	Search Term	Requeste	Response	Scan Date	Detec
2	40fd546dc880a691ab0090dc814cf2b273036efa38c9e51a6a703ca79c3eb5c2	0	1	27/02/2021 04:37	
3	0009afb5dc38f51bae09f7961a6dfd0845f7cb5b12063c0db45efb8a60b9c877	1	1	20/04/2020 11:41	
4	008aba818b12516d975eb00121e706f6aad665c83a1fde9a25ffbdde87930d57	1	1	03/05/2020 17:34	
5	00d0aa9fdbb5bdd44c33f7f4d46e9ab21d620d51e792d9af532f88661b0dc273	1	1	30/03/2020 02:30	
6	0110876bc53bec6bc0d47f0983c817aebf3e67b89ddfebc750b1594ea3b17c7c	1	1	17/03/2020 16:35	
7	014dfe34871229726 88f0c08ac515274154876e73065eea35d0242d1c7238044	1	1	18/04/2020 03:42	
8	016d3268a898add3088fc0a376b527c55165132665c91a51f6c1506551399c4f	1	1	31/07/2020 07:07	

Figure 4.3: Example of results from *virustotal-search.py*

AVClass2 *AVClass2* only accepts JSON files and they must adhere to a specific format. As such, the 883 CSV rows which were confirmed as malicious were converted to JSON files using a script developed as part of this study, namely *csv_to_json_convertor.py*. AVClass2 generated family tags for 669 samples (rows) based on the AV labels that VirusTotal had returned, it was unable to generate a result for 214 samples. Figure 4.4 shows the breakdown of family names across the dataset, as can be seen from the chart, *mirai* and *gafgyt* are the most common type. Both of these are DDoS type malware.

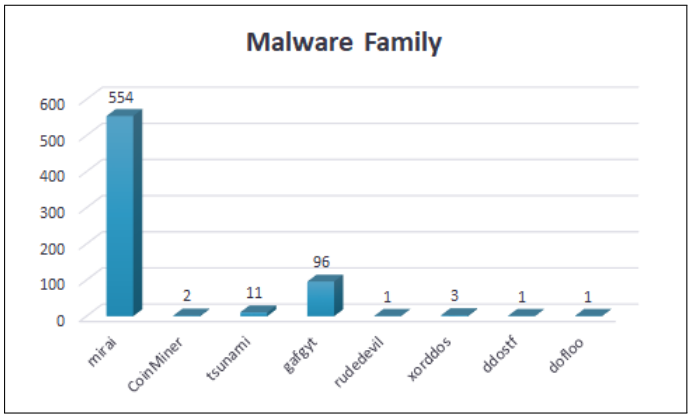


Figure 4.4: Malware family count across the dataset

4.4.2 Benign Samples

After an exhaustive internet search, it appears that datasets of benign executables are not something that is publicly available. So for this study, a dataset of benign executables was created by a) downloading firmware from IoT product vendors (D-Link, Wyze, Ezviz, IDIS global, Netgear and Reolink), b) using *BinWalk* to extract their file systems and c) using a combination of *grep* commands to search the file systems for ELF executables. Code Listing 2 shows the BinWalk commands used to extract the file system from the firmware image. Running the command with the *–signature* flag triggered a search for file signatures inside a firmware image, the results of which confirmed the file system type in use. To extract this file system, another BinWalk command was run, this time with a *–extract* flag, which extracted the root file system to a folder. Each firmware download was examined using BinWalk and their extracted file systems were searched for ELF executables.

```
#!/bin/bash
binwalk --signature --term
 ↪  ~/01_DataCollection/FirmwareSamples/WyzeCamV3/demo_wcv3_4.36.1.4.bin
binwalk --extract
 ↪  ~/01_DataCollection/FirmwareSamples/WyzeCamV3/demo_wcv3_4.36.1.4.bin
```

Listing 2: BinWalk Commands

Once a set of benign executables were gathered, a similar process to the analysis of the malware samples was followed. For example, each benign file was renamed with its SHA256 hash and any duplicates found were discarded. As per the malware samples, hashes were submitted to VirusTotal with the expectation this time that no results would be found. In total, 835 Benign samples were collected.

4.5 Dynamic Malware Analysis

Now that a preliminary sample set had been assembled, the next step was to run each of them (benign and malware) through the *LiSa* sandbox in order to capture System Call trace logs for each one. As previously stated in section 3.3, LiSa is a multiplatform Linux sandbox which provides automated malware analysis on various IoT CPU architectures. For each executable file passed to it, LiSa will determine the target CPU architecture of the executable and then boot up a QEMU[1] virtual machine using one of its CPU targeted images. It then copies the ELF file to the root directory in the image, starts all of the configured analysers (StaticAnalyzer, DynamicAnalyzer and NetworkAnalyzer) and runs the executable for a configured amount of time. The QEMU virtual machine is then stopped and a set of results, including System Call trace logs, are recorded in a file (*report.json*).

For the purposes of this study, all analysers were enabled although only the results from the DynamicAnalyzer were used. Executable files can be submitted to LiSa via its web interface or by using *curl* commands. Due to the large number of samples to execute, 1,504 in total (669 malware and 835 benign), curl commands were used. Also, in this study, the maximum execution time for each sample was configured as 120s, this was to keep the System Call trace logs at a manageable level (for further processing). The process for running each of the sample sets (benign and malware) through LiSa was as follows:

1. Start with a clean snapshot of both the VictimHost and NetworkServices Virtual Machines.
2. Start iNetSim on NetworkServices VM and perform basic connectivity checks (e.g., ping, nmap). from the VictimHost VM to the NetworkServices VM.
3. Start LiSa with 1 worker[2] on the VictimHost. This worker performs the analysis of the sample.
4. Trigger a curl command (e.g.,

[1] QEMU is a processor emulator - `https://www.qemu.org/`

[2] Note: informal testing with LiSa found the success rate (of samples) was lower when multiple workers were spun up.

```
curl -X POST http://localhost:4242/api/tasks/create/file -F
↪  file=@<sample\_file>
```

) to send a sample to the worker. This step was automated with a for loop to send all samples in a set to the LiSa worker. Each one sat in a queue until the worker was free to process it.

5. LiSa creates a unique id for each task that is submitted and once the task is complete the results can be found in a folder (with the task id as it's name) in *<LiSa_Install_dir>/LiSa/data/storage*.
6. Once all tasks were finished, the *report.json* in each task folder was renamed with the SHA256 hash of the corresponding executable. Note: if an executable failed to run in LiSa, then no *report.json* was produced for it.
7. Finally, all (renamed) *report.json* files were securely removed from the VictimHost VM to the DevMachine VM for further analysis and experimentation.

Post dynamic malware analysis, only 212 out of 835 benign samples executed successfully on LiSa. The reasons for the other 623 files failing are unknown, the only error that LiSa gave was that it failed to copy the sample to the image filesystem. Investigations into why there was an issue copying the files resulted in no answers. There was a higher rate of success among the malware samples with 366 out of 669 samples recording a successful outcome. In order to avoid an imbalanced dataset, a number of malware samples were removed from the 366 in order to bring the total samples more in line with the benign total. 3 out of the 366 related to single instances of specific malware families namely, *ddostf*, *dofloo* and *rudedevil* and as such were removed as a minimum of 2 samples are required for classification (training and test). Also, as there was an abundance of *mirai* samples, this total was halved (i.e., 251 to 126). The final sample collection totals for the experimental runs were 238 malware and 212 benign, see Figure 4.5 for a breakdown by malware family.

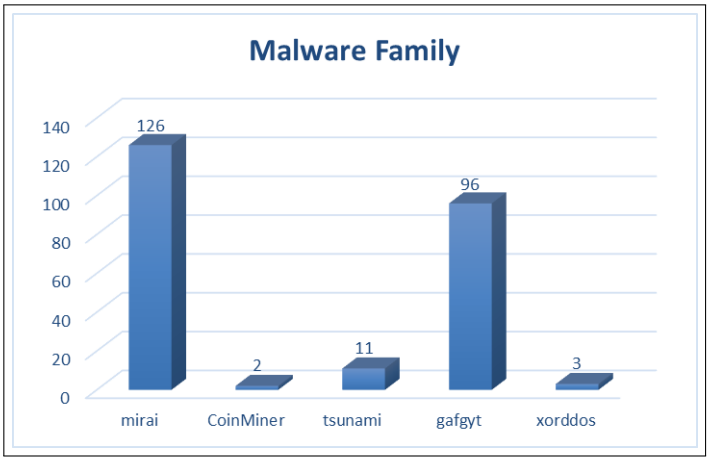


Figure 4.5: Malware family count in final sample collection

4.5.1 System Call Dataset Generation

The final stage before the experimental runs was to extract System Calls from each of the 450 (238 Malware and 212 Benign) dynamic analysis JSON results files and save them as individual dataset files (.DAT). *dataset_generator.py* was used to perform this function. Figures 3.3 and 3.4 show the input (<*sha256hash*>*.json*) and the output (<*sha256hash*>*_gafgyt.dat*) of the script. It accepted two arguments namely, sample type *-t* (i.e., malware or benign) and the number (first n) of System Calls *-s* to include in the output (see Code Listing 3). For the purposes of keeping the data samples manageable (across the 450 System Call Trace Logs), only the first 100 System Calls were extracted (from the JSON files) to each dataset.

```
#!/bin/bash
~/00_Scripts/dataset_generator.py -i
 ↪  ~/01_DataAnalysis/01_Malware_SysCall_Logs/tsunami/ -o
 ↪  ~/01_DataAnalysis/02_SysCall_Datasets/ -t malware -s 100
```

Listing 3: Dataset Generator Command

4.6 Experiments

An experimental run started with the Feature Extraction / Selection (python) module and concluded with the Malware Detection module. Figure 4.6 shows both the prerequisite and the experimental run steps. The prerequisite steps have already been covered in the preceding sections, so this section will focus on the experimental run steps only. The experiments were controlled by a shell script (see appendix B.7) and a set of experiment run parameters stored in a CSV (see Figure 3.11). The shell script was triggered by running the command in Code Listing 4 and upon start-up, it read the parameters from the passed CSV file.

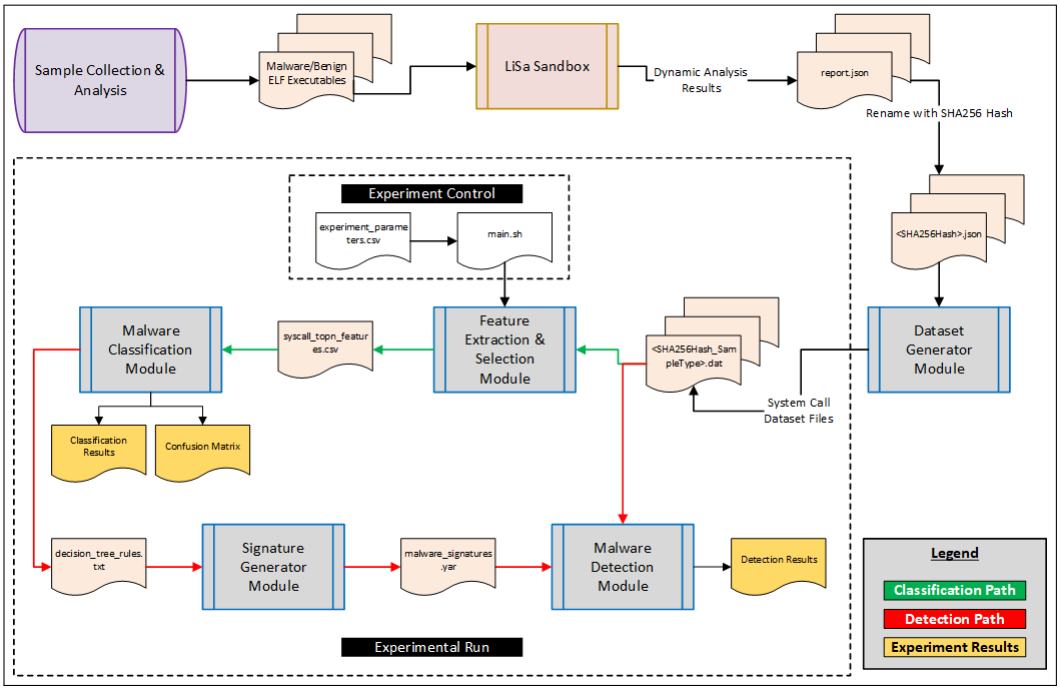


Figure 4.6: Malware Classification and Detection System

```
#!/bin/bash
~/00_Scripts/main.sh ~/02_Experiments/experiment_parameters.csv >
↪  ~/02_Experiments/experiment_results.log
```

Listing 4: Experimental Runs Command

Each row in the CSV file contained the parameters for a single experimental run. In total, the file contained 210 rows which was necessary to cover the following combination of parameters and their ranges:

- Feature Extraction / Selection *N*-gram size, Range: 1 to 6
- Feature Extraction / Selection Top *N* features, Range: 5, 10, 20, 50 and 100
- Malware Classification Tree Depth Size, Range: 2 to 8

Feature Extraction & Selection Module *feature_extraction_selection.py* was the first script to be triggered, see Code Listing 5 for an example command. Taking in the passed parameters, this script read in all of the System Call Dataset files and used the *scikit-learn* TfidfVectorizer class to perform the following:

- Created System Call *N*-grams based on the passed *n* size.
- Calculated the TF-IDF weight for each of the System Call *N*-grams.
- Discarded all System Call *N*-grams that did not make it into the top N features.
- Converted all non-zero TF-IDF weights to 1 (as any weight above zero means that a System Call *N*-gram exists and can thus be used for classification).
- Finally, output the results to a CSV file called *syscall_top<n>_features_<YYYYMMDDHHMISS>.csv*.

```
#!/bin/bash
~/00_Scripts/feature_extraction_selection.py -i
↪ ~/01_DataAnalysis/02_SysCall_Datasets/ -o
↪ ~/02_Experiments/exp_results_run_num_105/ -n 3 -t 100
```

Listing 5: *feature_extraction_selection.py* Command

Malware Classification Module *malware_classification.py* was triggered once Feature Extraction / Selection was complete, see Code Listing 6 for an example command. This module ingested the CSV output from the previous module and used the *scikit-learn* DecisionTreeClassifier class to perform the following:

- Loaded the features dataset (e.g., *syscall_top<n>_features_<YYYYMMDDHHMISS>.csv*).
- Divided the dataset into features (System Call *N*-grams) and the target variable (class label: benign or malware).
- Split the dataset into a 70% training and 30% test. Note: the *random_state* argument of the *sklearn.model_selection.train_test_split* method was set to 1 (for each run) in order to replicate the same shuffling (i.e., the same samples are categorised as training or test) across each of the 210 runs. Also, the *stratify* argument was set to an array of the target variables to ensure that there was a representative distribution of target classes in the test group.
- Performed the classification using a Decision Tree classifier with a max tree depth, as specified by the passed *-d* argument.
- Output the results from the classification as 1) a report with metrics such as precision, recall, f1-score etc. and 2) as a Confusion Matrix showing actual versus predicted class labels.
- Plotted and saved the model produced by the Decision Tree classifier to an image file.
- Finally, exported a set of 'If...Else' rules which were derived from the model to a text file called *syscall_top<n>_features_<YYYYMMDDHHMISS>_decision_tree_rules.txt*.

```
#!/bin/bash
~/00_Scripts/malware_classification.py -i
 ↪  ~/02_Experiments/exp_results_run_num_105/ -o
 ↪  ~/02_Experiments/exp_results_run_num_105/ -d 8
```

Listing 6: *malware_classification.py* Command

Signature Generator Module Once classification was complete, the *signature_generator.py* script was triggered, see Code Listing 7 for an example command.

```
#!/bin/bash
~/00_Scripts/signature_generator.py -i
 ↪  ~/02_Experiments/exp_results_run_num_105/ -o
 ↪  ~/02_Experiments/exp_results_run_num_105/
```

Listing 7: *signature_generator.py* Command

The script read *syscall_top<n>_features_<YYYYMMDDHHMISS>_ decision_tree_rules.txt* line by line and converted each line to a YARA signature (example shown in 3.9). Each *If* statement (in decision_tree_rules.txt) is evaluated (by the script) and if it has a $\leq$ condition then that specific System Call *N*-gram becomes a *NOT* string in the YARA conditions. Alternatively, if it has a $>$ condition then the System Call *N*-gram is included as a condition to look for. For example, the following decision tree rule:

```
if (write write exit_group <= 0.5) and (socket connect getsockname <=
 ↪  0.5) and (rt_sigaction rt_sigaction rt_sigaction <= 0.5) and (mmap2
 ↪  mprotect mmap2 > 0.5) and (openat stat openat > 0.5) then class:
 ↪  malware
```

became the following YARA signature:

```
(not $syscall_ngram_1) and (not $syscall_ngram_2) and (not
 ↪  $syscall_ngram_3) and $syscall_ngram_4 and $syscall_ngram_5 and
 ↪  (filesize > 0)
```

Malware Detection Module The final stage in the experimental run was Malware Detection. The *malware_detection.py* script read in the auto-generated YARA signatures (created by the previous module), compiled them and then checked for any matches in the System Call dataset files (.DAT). Results such as *TPR*, *FPR*, *TNR* and *FNR* were recorded.

```
#!/bin/bash
~/00_Scripts/malware_detection.py -i
 ↪  ~/01_DataAnalysis/02_SysCall_Datasets/ -o
 ↪  ~/02_Experiments/exp_results_run_num_105/ -s
 ↪  ~/02_Experiments/exp_results_run_num_105/
```

Listing 8: *malware_detection.py* Command

4.7 Results & Initial Analysis

As mentioned in the previous section, 210 experiments were run in total. For the purposes of readability the raw results from all of the experimental runs are contained in Appendix C. The following sections provide the top performing results and some initial analysis.

4.7.1 Malware Classification Results

Figure 4.7 shows the classification performance results from the 210 runs. Scores are between 0 and 1 on the y-axis. As can be seen from the figure, the scores across the runs were quite high (i.e., 0.9 or above with only 5 runs scoring less than that).

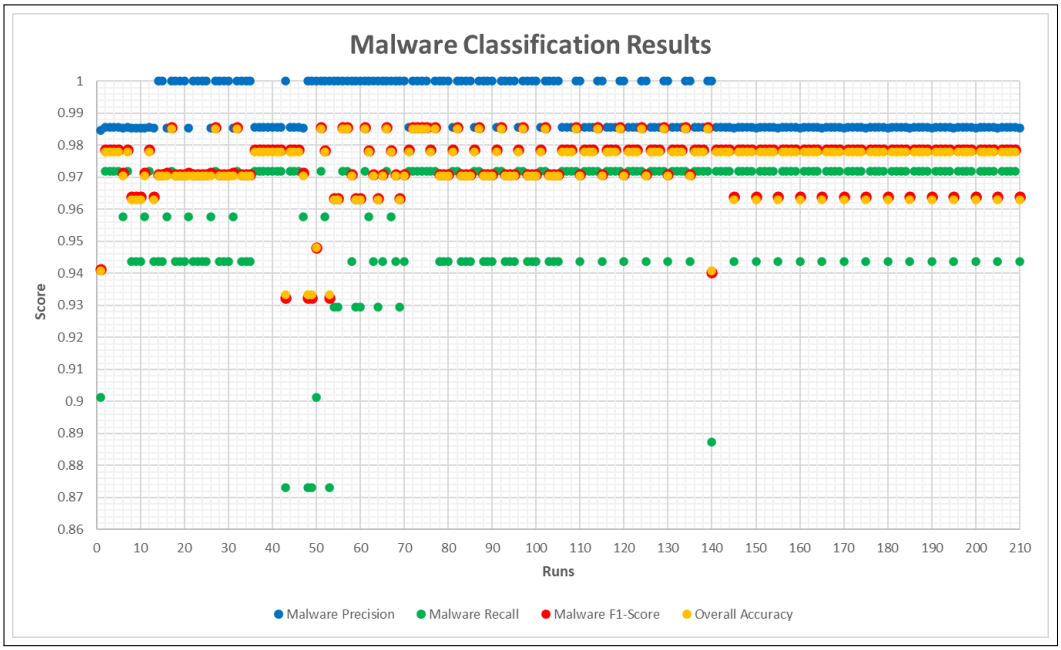


Figure 4.7: Malware Classification Overall Results

4.7.2 Malware Classification Results Analysis

As stated in section 2.4.5, *Recall* (also known as Sensitivity) is a much more relevant measure for malware detection studies. It is critical, in malware detection, that malware samples do not go undetected (i.e., the False Negative Rate is kept low). As *Recall* measures the percentage of the actual malware that was identified as such, it is the best metric for evaluating this type of classification problem. Other metrics such as *Precision*[3] and *F1-Score*[4] are not as useful for malware detection studies. This is because *Precision* measures the percentage of true positives out of all of the predicted positives meaning that False Positives (if present) are counted in that total. As such a high *Precision* score does not necessarily mean that every malware sample was correctly identified. In addition, the *F1-Score* gives equal weight to both measures (*Precision* and *Recall*) when averaging which means that it is suitable metric when seeking a balance between *Precision* and *Recall*.

[3] *Precision* is the fraction of predicted positive cases which are true positives.
[4] *F1-Score*is the harmonic average of recall and precision.

125 experimental runs achieved the highest *Recall* percentage for malware namely, 97.18%. This result confirmed that the classification model in those runs correctly labelled 97.18% of actual malware samples as malware. Considering only classification metrics, the only other difference between the 125 runs was the *Accuracy* score which was 97.78% (100 runs) and 98.52% (25 runs). Figure 4.8 shows the classification report from run number **72** which was 1 of 25 runs which achieved a *Recall* value of 97.18% and an *Accuracy* value of 98.52%.

```
5779  ################### - RESULTS - ###################
5780
5781  Classification Report:
5782
5783                precision    recall  f1-score   support
5784
5785       benign     0.9697    1.0000    0.9846        64
5786      malware     1.0000    0.9718    0.9857        71
5787
5788     accuracy                         0.9852       135
5789    macro avg     0.9848    0.9859    0.9852       135
5790  weighted avg     0.9856    0.9852    0.9852       135
5791
5792  ####################################################
```

Figure 4.8: Run 72 - Classification Report

As shown in the Confusion Matrix in Figure 4.9, there were 2 false negatives (malware labelled as benign) which is what brought the *Recall* percentage for malware down. All 64 benign test samples were correctly identified.

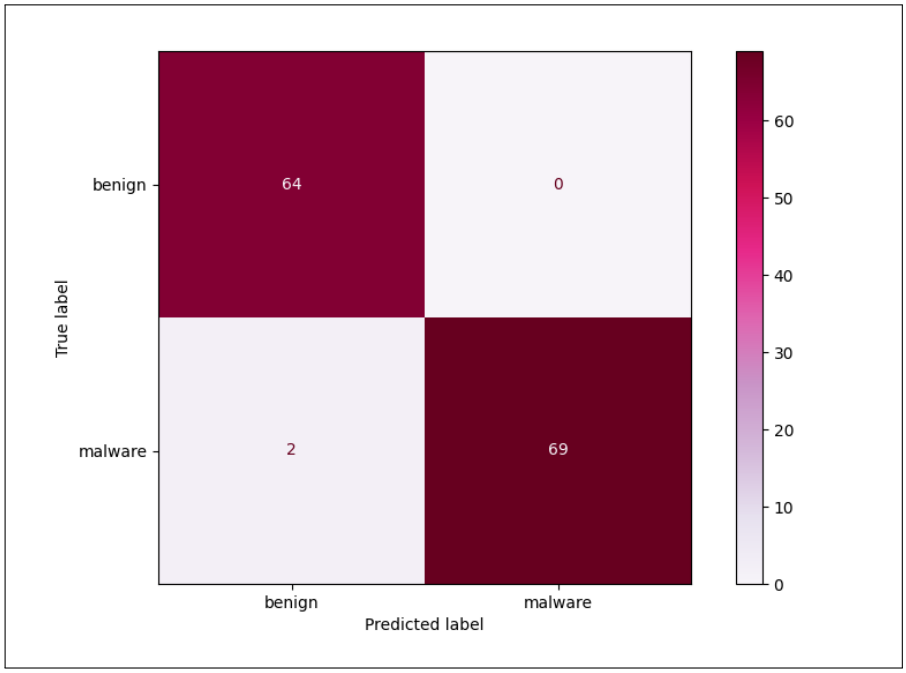


Figure 4.9: Run 72 - Confusion Matrix

4.7.3 Malware Detection Results

Figure 4.10 shows the detection rates from the 210 runs. Results are between 0% and 100% on the y-axis. As can be seen from the figure, the percentages across the runs were quite high (i.e., all but 1 run was above 90%) for *TPR* and *TNR*. Alternatively, both the *FPR* and *FNR* percentages are mostly low with the odd exception (i.e., the max *FPR* percentage was 3.77% and the max *FNR* percentage was 13.03%).

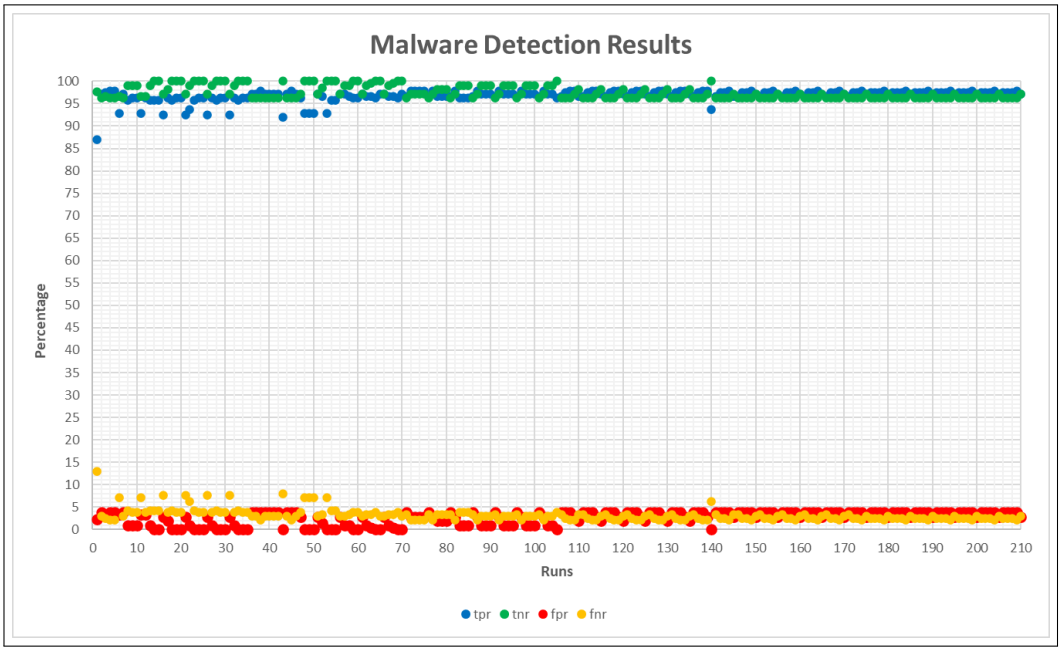


Figure 4.10: Malware Detection Overall Results

4.7.4 Malware Detection Results Analysis

When performing malware detection, the priority is to correctly identify all malware as malware (True Positives) and avoid missing any (False Negatives). Ideally, the false positive rate (FPR) will also be kept low but the cost of incorrectly flagging a benign process as malware is considered minimal compared to flagging malware as benign. Henceforth, a high *TPR* was the criteria used to determine the best detection results in the subsequent paragraph. A low *FPR* was used to reduce the results if more than one was returned. Alternatively, a low *TPR* was used to identify the worst performing run in terms of malware detection.

Figure 4.11 shows the detection results from run number **119** which was 1 of 17 runs which had the highest *TPR* percentage namely, 97.90%. This means that the generated signatures detected 233 out of 238 malware samples. Run number **119** also had the median number (4) of auto-generated signatures out of all of the 17 runs where the minimum was 2 and the maximum was 8.

```
9630  ################### - RESULTS - ######################
9631  Actual Malware Count: 238 | Actual Benign Count: 212
9632
9633  TP (True Positive) Count: 233, TPR: 97.90%
9634  TN (True Negative) Count: 206, TNR: 97.17%
9635  FP (False Positive) Count: 6, FPR: 2.83%
9636  FN (False Negative) Count: 5, FNR: 2.10%
9637  ########################################################
```

Figure 4.11: Run 119 - Detection Results

The decision tree generated in the classification phase of run number **119** is depicted in Figure 4.12. As can be seen, it contains a number of different System Call four-grams ($n = 4$) and has 4 malware leaves (end nodes where class = malware) which translate to the 4 auto-generated YARA signatures (some are shown in Figure 4.13).

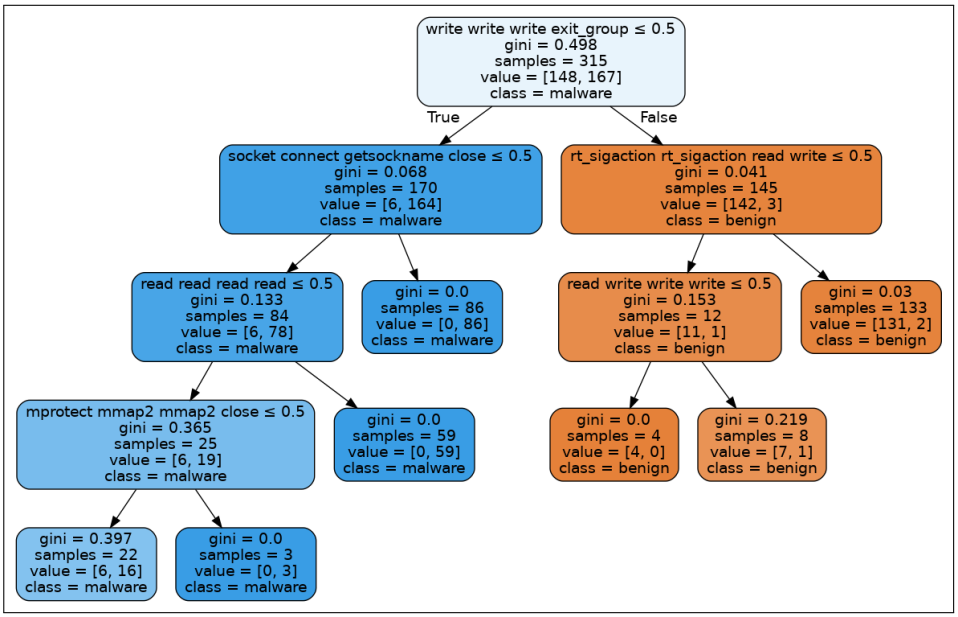


Figure 4.12: Run 119 - Decision Tree

```
rule malware_1 {
    meta:
        author = "FWalsh"
        date = "30/07/2021"
    strings:
        $syscall_ngram_1 = "write write write exit_group"
        $syscall_ngram_2 = "socket connect getsockname close"

    condition:
        (not $syscall_ngram_1) and $syscall_ngram_2 and (filesize > 0)
}

rule malware_2 {
    meta:
        author = "FWalsh"
        date = "30/07/2021"
    strings:
        $syscall_ngram_1 = "write write write exit_group"
        $syscall_ngram_2 = "socket connect getsockname close"
        $syscall_ngram_3 = "read read read read"

    condition:
        (not $syscall_ngram_1) and (not $syscall_ngram_2) and $syscall_ngram_3 and (filesize > 0)
}

```

Figure 4.13: Run 119 - Auto-generated YARA Signatures

Table 4.4 shows the System Calls that were used in the auto-generated signatures for run number **119** and their meaning.

System Call	Function
connect	initiate a connection on a socket
close	close a file descriptor
exit_group	exit all threads in a process
getsockname	get socket name
mmap2	map files or devices into memory
mprotect	set protection on a region of memory
read	read from a file descriptor
socket	create an endpoint for communication
write	write to a file descriptor

Table 4.4: System Calls from YARA signatures (Run Number **119**)

4.8 Conclusions

This chapter documented the end-to-end process for evaluating a Decision Tree-Based Signature Generation approach for IoT Malware Detection. It gave detailed descriptions of the environment setup, the tools that were used, the code that was developed and the experiments that were run.

As part of this experimental work, a virtual machine providing network services was setup. This was solely to mimic a more realistic environment for executing malware however the network traffic was not captured or analysed as part of this study. Regarding sample collection, gathering benign samples required quite a bit of manual effort for a number of reasons. Firstly, there was a lack of suitable firmware downloads available on-line. It would appear that the preference for firmware upgrades on smart/IoT devices is via an accompanying mobile application. This is most likely a security-based decision but it does mean that appropriate IoT firmware images are harder to find. Secondly, the firmware images that were available harboured a lot of the same ELF executables.

Running samples on LiSa had a mixed outcome and the success rate was 38.43% (578 out of 1,504 samples). The number of successful samples (450 after reduction) however were sufficient for this study. As mentioned in previous sections, optimal parameters for N-gram size, top N features and decision tree depth were not known hence the 210 runs to try and identify them for this study. The results from these 210 runs can also serve as possible suggestions for N-gram size, top N features and/or decision tree depth for other similar studies. Finally, the top classification and detection results did not always equate to a single experimental run so in those cases a representative run was presented. This was not a surprise, as with 210 runs it is to be expected that there will be some overlap in terms of results.

Chapter 5

Evaluation

5.1 Introduction

The aim of this study was to evaluate a Decision Tree-Based Signature Generation approach for IoT Malware Detection. This aim was broken down into a number of objectives which aspired to answer four research questions (listed in section 2.6). This chapter will evaluate whether or not these research questions were answered by reviewing the gathered evidence for each question in the sections below.

5.2 Research Question 1

> *What combinations of N-gram size, number of features and decision tree depth result in the best classification performance?*

Figure 5.1 shows the combination of parameters which were used in the 25 experimental runs that achieved the top $Recall = 97.18\%$ and $Accuracy = 98.52\%$ scores. As can be seen from the diagram, there was quite a bit of variance in the individual (i.e., N-gram size, top N features and decision tree depth) and combined parameters. Referring back to section 4.6 which lists the full range of values for each parameter, it appears that only N-gram size = 5 or 6 did not feature in the 210 experimental runs. Table 5.1 shows the individual parameters which occurred the most across all runs. Run Number **72** was the only run which combined all three.

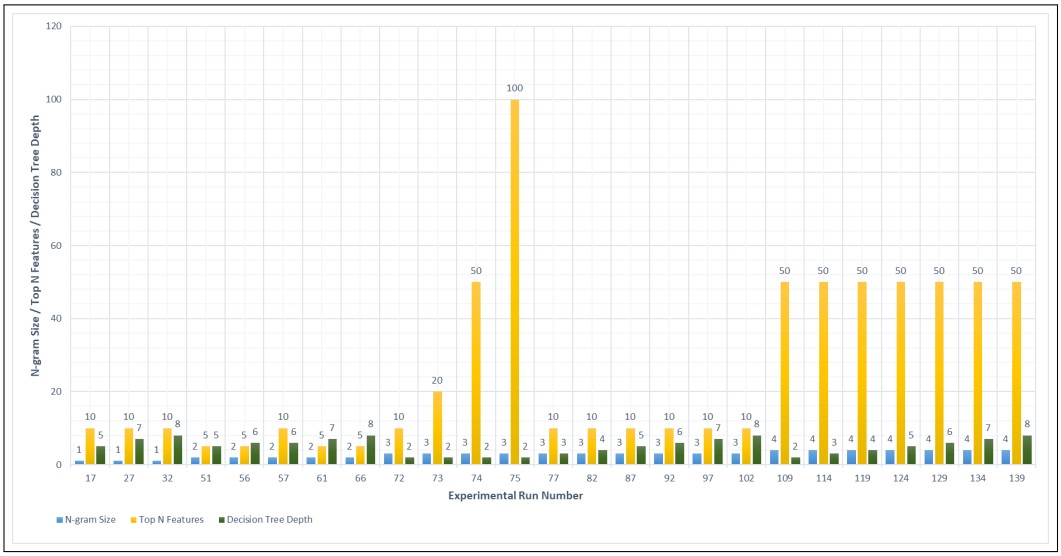


Figure 5.1: Experimental Run Parameters @ *Recall* = 97.18% and *Accuracy* = 98.52%

Considering the counts in Table 5.1, it would seem that using specific *N*-gram Size and Top *N* Features is more influential when it comes to achieving high *Recall* and *Accuracy* results. Alternatively, it would appear that tree depth size is less influential as there is no real preference for any specific length (i.e., the whole range of values appear) in the results. A possible correlation between *N*-gram size and top *N* features has emerged, in that the larger the *N*-gram size the more features that are required. For example, *N*-gram size = 1 or 2 only required 5 - 10 features to produce the top *Recall* and *Accuracy* scores whereas *N*-gram size = 4 required 50 features. Additionally, *N*-gram size = 3 appears to have been the most flexible in terms of the number of top *N* features required as it occurred in combinations with 10, 20, 50 and 100 features.

Parameter Type	Parameter Value	Count
N-gram Size	3	10
Top *N* Features	10	11
Decision Tree Depth	2	5

Table 5.1: Research Question 1 - Most Featured Parameters

Table 5.2 shows a comparison between this work and related work which also used a combination of Feature Extraction/Selection and Decision Trees for malware classification and detection. As can be seen from the table, not all approaches used *N*-grams and the majority of work did not report (n/r) decision tree depth. Singh and Hofmann (2018) was the only study to achieve a higher *Recall* score then the method proposed in this work. The authors did require 67.7% more features to achieve this higher score which is most likely due to them not using *N*-grams which would have added context (e.g., sequence) to their list of features. The results depicted in Figure 5.4 answer the stated research question in that a selection of parameter combinations which achieved the best performance of the classifier have been provided. Also, Table 5.1 highlights what is most likely the strongest combination of *N*-gram size, number of features and decision tree depth out of all 25 options which serves as an informed starting point for any follow-on studies.

Source	Feature Type	*N*-gram Size	Num of Features	Decision Tree Depth	Recall	Accuracy
Khammas et al. (2019)	Hex Codes	4	n/r	n/r	96.50%	100%
Yuxin et al. (2011)	System Calls	3	150	n/r	88.00%	87.50%
Soe et al. (2019)	System Calls	n/a	43	10	n/r	n/r
Asmitha and Vinod (2014)	System Calls	1	27	n/r	95.50%	97.60%
Firdausi et al. (2010)	System Calls	n/a	11	n/r	94.50%	94.90%
Singh and Hofmann (2018)	System Calls	n/a	31	n/r	97.22%	97.58%
Proposed Method	System Calls	3	10	2	97.18%	98.52%

Table 5.2: Comparison to related work

5.3 Research Question 2

> *What combinations of N-gram size, number of features and decision tree depth result in a high detection rate for IoT malware?*

Figure 5.2 shows the combination of parameters which were used in the 17 experimental runs that achieved the highest $TPR = 97.9\%$ and lowest $FPR = 2.83\%$ scores. In comparison to the classification results from the previous section, there is slightly less variance in both the individual and combined parameters reported in the detection results. Table 5.2 shows the individual parameters which occurred the most across all runs. Run Number **74** was the only run which combined all three.

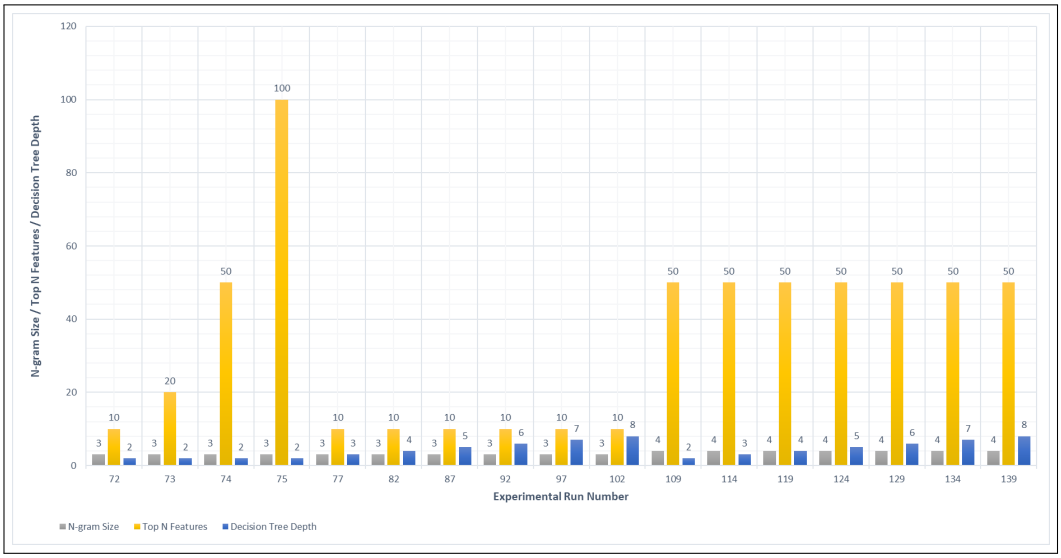


Figure 5.2: Experimental Run Parameters @ $TPR = 97.9\%$ and $FPR = 2.83\%$

Again it would seem that using specific N-gram Size and Top N Features (as shown in Table 5.3) is potentially more influential when it comes to achieving high TPR and low FPR percentages. Tree depth, as per the previous set of results, appears to be less influential. Ignoring the tree depth values, N-gram Size $= \{3,4\}$ and Top N Features $= \{10,50\}$ were the most common combinations that produced strong malware detection results. Researchers in Canzanese et al. (2015) arrived at a similar conclusion when they experimented with N-gram sizes: $1-4$. They observed that at $n = 2$ there was a significant improvement in their malware detectors from $TPR = 30\%$ to a TPR between 80% and 90%. However, to achieve a TPR % greater than 90%, they had to use $n \in \{3,4\}$. In the work in this thesis, a TPR % above 90% was achieved for $n \in \{1,2,3,4\}$. This difference between the studies, may be down to the malware type that was targeted (i.e., IoT malware in this work, Windows malware in Canzanese et al. (2015)). The authors in Alhanahnah et al. (2018) suggested that IoT malware is less complex and thus possibly easier to detect.

Parameter Type	Parameter Value	Count
N-gram Size	3	10
Top *N* Features	50	8
Decision Tree Depth	2	5

Table 5.3: Research Question 2 - Most Featured Parameters

As per the previous section, a selection of parameter combinations which achieved the highest malware detection rate have been provided in Figure 5.2. Table 5.2 documents the most frequently occurring individual parameters. Run number **74** proved they can achieve a high malware detection rate as such, it appears that the research question has been addressed.

5.4 Research Question 3

> *To what extent is classification performance an indicator of the detection rate for IoT malware?*

This question aims to determine if there is any correlation between the *Recall* percentage achieved in classification and the *TPR* score achieved in the detection phase. As described in section 4.6, the malware signatures used for detection were generated using the decision tree rules produced by the classification phase. Combine that fact with the knowledge that *Recall* and *TPR* are calculated in the same manner (i.e., $\frac{TP}{TP+FN}$) it is feasible that one (*Recall*) may be an indicator for the other (*TPR*).

Figure 5.3 shows the *TPR* % plotted against the *Recall* %. The size of each circle represents the number of experimental runs that those specific combination of *TPR* % and *Recall* % occurred together. For example, the green circle on the very bottom of the figure shows that there was 1 occurrence of a $TPR = 86.97\%$ when $Recall = 90.14\%$. Table 5.4 shows the top 5 most frequently occurring results across all the experimental runs. The linear trendline (dashed black line) points upwards, when read left to right, which indicates that there may be a positive correlation between *Recall* % and *TPR* %. Also, the r^2 value is equal to 41.45% which shows that a reasonable portion of the variation in the

dependent variable (*TPR %*) can be predicted by the independent variable (*Recall %*). This can be seen by observing the location of the circles (data points) in relation to the trendline (i.e., there are quite a few on or close to it and a similar portion that are further away). A similar study by Soe et al. (2019) also adopted a decision tree-based signature generation approach for IoT cyber-attacks however they did not perform malware detection with the generated signatures. They also did not include any classification results in their write-up. This study builds on that work by performing malware detection with the generated signatures and providing both classification and detection results. It is believed that the results discussed above address the research question.

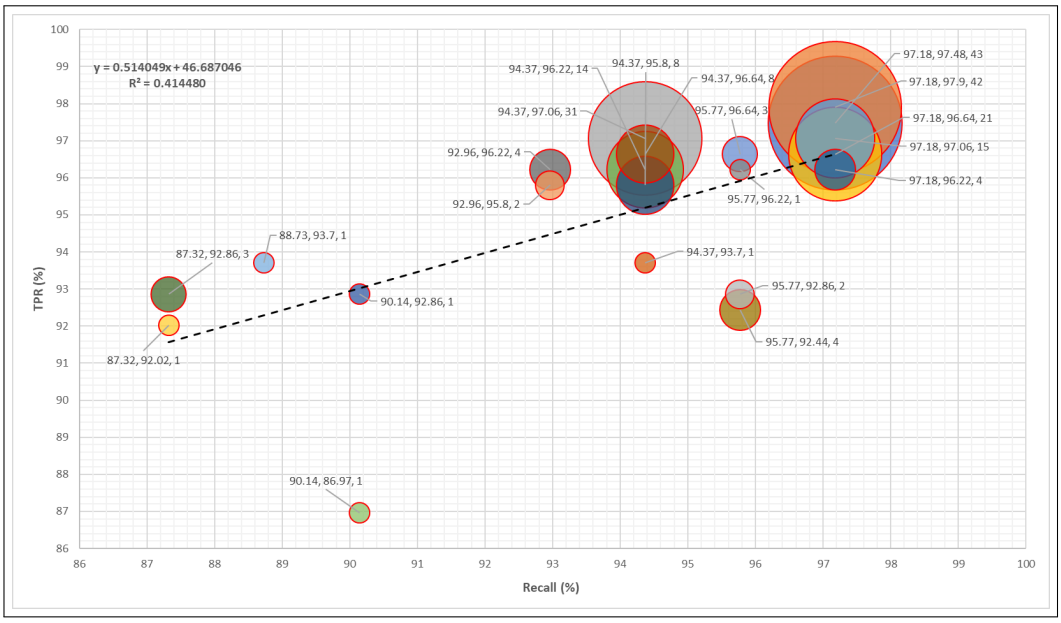


Figure 5.3: Classification Performance vs Detection Performance

Recall	*TPR*	Count
97.18%	97.48%	43
97.18%	97.9%	42
94.37%	97.06%	31
97.18%	96.64%	21
97.18%	97.06%	15

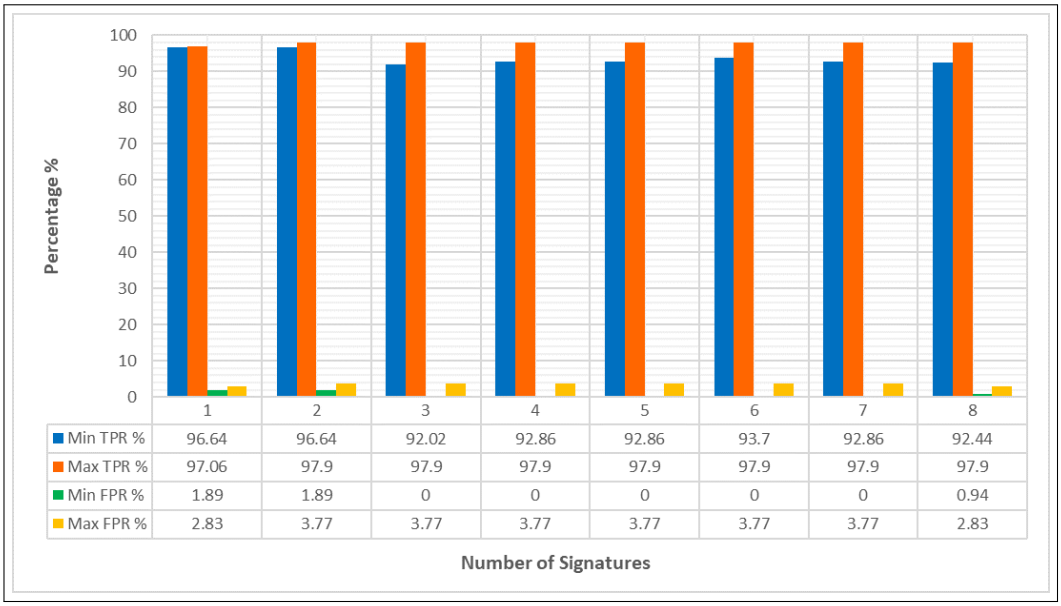
Table 5.4: Research Question 3 - Top 5 Frequent Combinations

5.5 Research Question 4

What is the minimum number of System Call-based signatures which can achieve a high detection rate for IoT malware?

Figure 5.4 shows the number of auto-generated signatures and the minimum/maximum *TPR %* and *FPR %* scores they achieved. *TNR %* and *FNR %* are not depicted in the chart as their values can be inferred from *FPR %* and *TPR %* values, respectively. As can be seen, the number of auto-generated signatures ranged from 1 to 8. A high *TPR %*, as mentioned previously, is more critical than a low *FPR %* in malware detection so considering that, it would appear that 2 signatures is the minimum number which can achieve a high malware detection rate (i.e., 97.9%). Two signatures did, however, incur a higher max *FPR %* result in some experimental runs, most likely due to the signatures being too generic and flagging benign samples incorrectly. To retain the top *TPR %* value and reduce the *FPR %* value, 8 signatures could be used instead which is still a relatively small amount of signatures considering the storage constraints of IoT devices. This number of signatures detected, at their best, 233 out of 238 malware samples and 220 out of 238 at their worst. In comparison, Abbas and Srikanthan (2017) required 3 signatures to detect 63 out of 70 malware samples which is a 90% detection rate for a considerably smaller sample set. Belaoued et al. (2019) implemented a similar type of system for auto-generation of YARA signatures. They reported a higher *TPR %* namely, 98.28%, than the maximum *TPR %* in this study however they did not reveal how many signatures it took to achieve that. The results presented in Figure 5.4 confirm the minimum number of

signatures required to achieve a high detection rate for IoT malware as such, this research question has been satisfied.



	1	2	3	4	5	6	7	8
Min TPR %	96.64	96.64	92.02	92.86	92.86	93.7	92.86	92.44
Max TPR %	97.06	97.9	97.9	97.9	97.9	97.9	97.9	97.9
Min FPR %	1.89	1.89	0	0	0	0	0	0.94
Max FPR %	2.83	3.77	3.77	3.77	3.77	3.77	3.77	2.83

Figure 5.4: Number of Signatures vs *TPR % / FPR %*

5.6 Limitations of the Study

While all of the research questions were answered, this study did have some limitations. Firstly, the number of malware and benign samples used was not as large as hoped. As described in sections 4.4.1 and 4.5, the initial malware sample count was high (1,730) but that was ultimately reduced by 86% to 238 samples. With regards to the benign samples these were reduced by 78% to 212 due to a large majority of the samples not registering a successful outcome when dynamic analysis was performed. Though the sample set was greatly reduced, the number of samples tested were on par with a number of related studies (i.e., Darabian et al. (2020) had 247 malware and 269 benign, Singh and Hofmann (2018) had 216 malware and 278 benign, Shobana and Poonkuzhali (2020) had 200 of each).

There was also a lack of variation in the type of malware samples which were included in this study. As shown in Figure 4.5, the *mirai* and *gafgyt* malware families, which are both DDoS type malware, made up 93% of the samples used in the study. This lack of variation however, was because the only publicly available IoT malware dataset

predominately contained DDoS type malware. This dataset was created from the malware captured in a honeypot so it would seem that DDoS malware is the most common threat to IoT.

5.7 Conclusions

This chapter presented the results captured in this work and evaluated their meaning. One key takeaway was that *N*-gram Size and Top *N* Features are the most effective parameters when performing classification and detection on IoT malware using System Calls. Results from related work support this observation. This work has shown that classification performance is an indicator of detection performance when a data mining approach is adopted to generate malware signatures. This data-driven approach could potentially shift the burden of manual signature generation to a more informed (based on classification outcomes) automated method. In addition, the combination of classification and detection results presented here have not been reported in many studies as discussed previously and as such, these results advance the knowledge in this area. Finally, in this study the minimum number of signatures needed to detect a reasonably large portion of IoT malware was discovered. This is an important detail as malware attacks on IoT devices are increasing (as per section 2.2) and malware detection solutions which support constrained resources are needed.

Chapter 6

Conclusions

6.1 Summary of Work Done

This study set out to evaluate a Decision Tree-Based Signature Generation approach for IoT Malware Detection. It began by collecting a suitable set of IoT malware and benign executables to experiment with. After that, system call trace logs were captured for the collected samples using the dynamic analysis module provided by the LiSa sandbox. The trace logs were then converted into datasets of System Calls and these were processed through a Feature Extraction & Selection module which used *N*-grams and *TF-IDF* to elicit a set of features to represent each sample. Next, the features were ingested into a Decision Tree classifier which performed binary classification and generated a set of decision tree rules. The penultimate step (of the approach) was for the decision tree rules to be converted to YARA signatures which were then used to perform malware detection. The whole end to end approach was evaluated at two different points, 1) the classification phase and 2) the detection phase.

6.2 Delivery against Objectives

For reference purposes, the objectives of this work are stated below:

1. To conduct a literature review on current research in behaviour-based Signature Generation approaches for IoT Malware Detection. Included in this review will be an analysis of the malware that attack IoT devices and the approaches that have been adopted to detect them.

2. To design, based on existing literature, a method which uses Data Mining techniques to elicit relevant behavioural features from IoT Malware and Benign samples which can be used to detect malware.

3. To implement the devised method in a systematic manner which allows for an optimal set of signatures to be generated for use in IoT Malware Detection.

4. To evaluate the accuracy of the generated signatures in detecting IoT Malware in a controlled, repeatable test environment.

Objective 1 This aim was to undertake an in-depth analysis of the current state of research in behaviour-based Signature Generation approaches for IoT Malware Detection. Chapter 2 presents a focused analysis of the latest research in IoT malware and detection using classification approaches. In particular, a number of studies have shown that Decision Tree classifiers are lightweight, simple to implement and produce promising results when classifying malware. They also offer a base on which to perform signature generation for malware detection systems. A lack of suitable IoT malware executable datasets was identified and the limiting factor this has had on previous studies. Finally, a number of studies closely related to this work were identified and their limitations were proposed as research gaps for this work to address. It is the author's belief that this objective has been met.

Objective 2 To address this objective a method which used data mining techniques (i.e., *N*-grams, *TF-IDF* and Decision Tree classification) was designed to perform IoT malware signature-based detection. Chapter 3 describes in detail the design undertaken and the justification for the methods used at each stage. It also presents a breakdown of the experimental design and a description of the type of results which would be produced in order to evaluate the approach. It is the author's belief that this objective has been met.

Objective 3 The design created for the previous objective was implemented using a combination of third-party tools (e.g., BinWalk, VirusTotal API, AvClass2, LiSa Sandbox, YARA) and Python modules written specifically for this study. Chapter 4 gives an overview of the end to end solution (see Figure 4.6) including the environment setup, the prerequisite tasks and the experiments performed using the implemented system. A number of challenges were identified during the implementation namely, the lack of available

IoT benign samples. The success rate of running of executables on LiSa also proved variable. However, in spite of this a reasonable sample size was included in the experimental runs and a range of parameters for *N*-gram size, top *N* features and decision tree depth were tested across 210 runs. It is the author's belief that this objective has been met.

Objective 4 The final aim was to evaluate the implemented approach in a repeatable manner. As discussed in Section 4.6, a CSV file with experimental parameters and a main run script were developed. This automated the running of multiple experiments which used a different set of parameters each time. Relevant results (e.g., *Recall*, *Accuracy*, *TPR %*, *FPR %*) were captured in order to allow the author to evaluate the detection rate of the auto-generated signatures. Chapter 5 and Appendix C provide the details of these experimental runs and the interpretation of them. Briefly, the results showed that *N*-gram Size and Top *N* Features are the most effective parameters when performing classification and detection on IoT malware. In addition, a potential positive correlation between classification and detection performance was observed. Finally, this chapter presented a combination of classification and detection results for IoT malware detection which have not been featured in many studies to date. It is the author's belief that this objective has been met.

6.3 Future work

As with all research studies there are some limitations which are obvious candidates for future work. As a follow on to this study, the items below could be investigated further:

- **Network Traffic Analysis** - Network traffic was captured as part of the malware analysis performed in this study but it was discarded as the study was focused exclusively on behavioural data (i.e., System Calls). Further insights and/or improvements could be gained by combining network traffic data with the System Calls as observed in Belaoued et al. (2019).

- **Ensemble Decision Tree Classifier** - Consider using an ensemble decision tree classifier (i.e., *Random Forest*) in the classification phase as they typically outperform individual decision tree classifiers. However, converting their output to a set of rules which can be converted to YARA signatures most likely is not a trivial step.

- **Real IoT Devices** - With some adjustment required, perform the malware detection

phase on real IoT devices so that performance measurements (e.g., storage usage, CPU usage) can be measured.

- **Unknown Malware Samples** - This study produced quite a small number of System Call-based signatures which had a high detection rate when applied to a reasonable malware sample size. However, these were all known malware samples, the signatures were not tested on any samples that did not feature in the classification phase. It would be interesting to investigate how detecting unknown samples could be incorporated into the approach.

References

Abbas, M. F. B., & Srikanthan, T. (2017). Low-Complexity Signature-Based Malware Detection for IoT Devices. In *8th international conference, atis 2017 auckland, new zealand, july 6–7, 2017 proceedings.* doi: 10.1007/978-3-662-45670-5

Aiswarya Mohan, K. P., Chandran, S., Gressel, G., Arjun, T. U., & Pavithran, V. (2020). Using Dtrace for Machine Learning Solutions in Malware Detection. *2020 11th International Conference on Computing, Communication and Networking Technologies, ICCCNT 2020*. doi: 10.1109/ICCCNT49239.2020.9225633

Alasmary, H., Khormali, A., Anwar, A., Park, J., Choi, J., Abusnaina, A., … Mohaisen, A. (2019). Analyzing and Detecting Emerging Internet of Things Malware: A Graph-Based Approach. *IEEE Internet of Things Journal*, *6*(5), 8977–8988. doi: 10.1109/JIOT.2019.2925929

Alhanahnah, M., Lin, Q., Yan, Q., Zhang, N., & Chen, Z. (2018). Efficient signature generation for classifying cross-architecture IoT malware. In *2018 ieee conference on communications and network security, cns 2018* (pp. 1–9). IEEE. doi: 10.1109/CNS.2018.8433203

Aman, N., Saleem, Y., Abbasi, F. H., & Shahzad, F. (2017). A hybrid approach for malware family classification. *Communications in Computer and Information Science*, *719*(June), 169–180. doi: 10.1007/978-981-10-5421-1_14

Arshad, J., Azad, M. A., Amad, R., Salah, K., Alazab, M., & Iqbal, R. (2020). A Review of Performance, Energy and Privacy of Intrusion Detection Systems for IoT. *Electronics (Switzerland)*, *9*(4), 1–24. doi: 10.3390/electronics9040629

Aslan, O., & Samet, R. (2020). A Comprehensive Review on Malware Detection Approaches. *IEEE Access*, *8*, 6249–6271. doi: 10.1109/ACCESS.2019.2963724

Asmitha, K. A., & Vinod, P. (2014). Linux malware detection using extended-symmetric uncertainty. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *8804*, 319–332. doi: 10.1007/978-3-319-12060-7_21

Babaagba, K. O., & Adesanya, S. O. (2019). A study on the effect of feature selection on malware analysis using machine learning. *ACM International Conference Proceeding Series*, *Part F1481*, 51–55. doi: 10.1145/3318396.3318448

Bai, J., Yang, Y., Mu, S., & Ma, Y. (2013). Malware detection through mining symbol table of linux executables. *Information Technology Journal*, *12*(2), 380–384. Retrieved from `http://library1.nida.ac.th/termpaper6/sd/2554/19755.pdf` doi: 10.3923/itj.2013.380.384

Bazrafshan, Z., Hashemi, H., Fard, S. M. H., & Hamzeh, A. (2013). A survey on heuristic malware detection techniques. *IKT 2013 - 2013 5th Conference on Information and Knowledge Technology*, 113–120. doi: 10.1109/IKT.2013.6620049

Belaoued, M., Boukellal, A., Koalal, M. A., Derhab, A., Mazouzi, S., & Khan, F. A. (2019). Combined dynamic multi-feature and rule-based behavior for accurate malware detection. *International Journal of Distributed Sensor Networks*, *15*(11). doi: 10.1177/1550147719889907

Boujnouni, M. E., Jedra, M., & Zahid, N. (2016). New malware detection framework based on N-grams and Support Vector Domain Description. *Proceedings of the 2015 11th International Conference on Information Assurance and Security, IAS 2015*, 123–128. doi: 10.1109/ISIAS.2015.7492756

Canzanese, R., Mancoridis, S., & Kam, M. (2015). System Call-Based Detection of Malicious Processes. *Proceedings - 2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015*, 119–124. doi: 10.1109/QRS.2015.26

Choi, J., Spaulding, J., Anwar, A., Nyang, D. H., Alasmary, H., & Mohaisen, A. (2019). IoT malware ecosystem in the wild: A glimpse into analysis and exposures. *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, SEC 2019*, 413–418. doi: 10.1145/3318216.3363379

Choi, S. K., Yang, C. H., & Kwak, J. (2018). System hardening and security monitoring for IoT devices to mitigate IoT security vulnerabilities and threats. *KSII Transactions on Internet and Information Systems*, *12*(2), 906–918. doi: 10.3837/tiis.2018.02.022

Damodaran, A., Troia, F. D., Visaggio, C. A., Austin, T. H., & Stamp, M. (2017). A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, *13*(1), 1–12. doi: 10.1007/s11416-015-0261-z

Darabian, H., Dehghantanha, A., Hashemi, S., Homayoun, S., & Choo, K. K. R. (2020). *An opcode-based technique for polymorphic Internet of Things malware detection* (Vol. 32) (No. 6). doi: 10.1002/cpe.5173

de Oliveira, I. L., Grégio, A. R. A., & Cansian, A. M. (2012). A Malware Detection System Inspired on the Human Immune System. In *Computational*

science and its applications – iccsa 2012 (pp. 286—-301).

Dimjašević, M., Atzeni, S., Rakamarić, Z., & Ugrina, I. (2016). Evaluation of android malware detection based on system calls. *IWSPA 2016 - Proceedings of the 2016 ACM International Workshop on Security and Privacy Analytics, co-located with CODASPY 2016*, 1–8. doi: 10.1145/2875475.2875487

Dinh, P. V., Shone, N., Dung, P. H., Shi, Q., Hung, N. V., & Nguyen Ngoc, T. (2019). Behaviour-aware malware classification: Dynamic feature selection. *Proceedings of 2019 11th International Conference on Knowledge and Systems Engineering, KSE 2019*. doi: 10.1109/KSE.2019.8919491

Dutra, B. V., Alencastro, J. F., Caldas Filho, F. L., Martins, L. M. C. E., de Sousa Jr., R. T., & Albuquerque, R. O. (2019). HIDS by signature for embedded devices in IoT networks. In *V jornadas nacionales de investigación en ciberseguridad* (pp. 53–61).

Echeverría, A., Cevallos, C., Ortiz-Garces, I., & Andrade, R. O. (2021). Cybersecurity model based on hardening for secure internet of things implementation. *Applied Sciences (Switzerland)*, *11*(7). doi: 10.3390/app11073260

Eclispe IoT. (2020). *2020 IoT Developer Survey Key Findings* (Tech. Rep.). Retrieved from `https://iot.eclipse.org/community/resources/iot-surveys/assets/iot-developer-survey-2020.pdf`

Firdausi, I., Lim, C., Erwin, A., & Nugroho, A. S. (2010). Analysis of machine learning techniques used in behavior-based malware detection. *Proceedings - 2010 2nd International Conference on Advances in Computing, Control and Telecommunication Technologies, ACT 2010*, 201–203. doi: 10.1109/ACT.2010.33

F-Secure. (2019). *IOT Threat landscape* (Tech. Rep.). Retrieved from `https://blog-assets.f-secure.com/wp-content/uploads/2019/04/01094545/IoT-Threat-Landscape.pdf`

Hadar, N., Siboni, S., & Elovici, Y. (2017). A lightweight vulnerability mitigation framework for IoT devices. *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy, co-located with CCS 2017*, 71–75. doi: 10.1145/3139937.3139944

Holst, A. (2021). *Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2030.* Retrieved 2021-05-29, from `https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/`

Huang, W., Hou, E., Zheng, L., & Feng, W. (2018). MixDroid: A multi-features and multi-classifiers bagging system for Android malware detection. *AIP Conference Proceedings*, *1967*(May). doi: 10.1063/1.5038987

Hughes, K., & Qu, Y. (2014). Performance measures of behavior-based signatures: An anti-malware solution for platforms with limited computing resource. *Proceedings - 9th International Conference on Availability, Reliability and Security, ARES 2014*, 303–309. doi: 10.1109/ARES.2014.47

Ioulianou, P., Vasilakis, V., Moscholios, I., & Logothetis, M. (2018). A Signature-based Intrusion Detection System for the Internet of Things. *Information and Communication Technology Form*.

Islam, R., Tian, R., Batten, L. M., & Versteeg, S. (2013). Classification of malware based on integrated static and dynamic features. *Journal of Network and Computer Applications*, *36*(2), 646–656. Retrieved from http://dx.doi.org/10.1016/j.jnca.2012.10.004 doi: 10.1016/j.jnca.2012.10.004

Kaspersky. (2021). *IT threat evolution Q1 2021. Non-mobile statistics.* Retrieved 2021-06-02, from https://securelist.com/it-threat-evolution-q1-2021-non-mobile-statistics/102425/

Kato, S., Tanabe, R., Yoshioka, K., & Matsumoto, T. (2021). Adaptive Observation of Emerging Cyber Attacks targeting Various IoT Devices. In *Ifip/ieee international symposium on integrated network management (im).*

Khammas, B. M. (2020). The performance of iot malware detection technique using feature selection and feature reduction in fog layer. *IOP Conference Series: Materials Science and Engineering*, *928*(2). doi: 10.1088/1757-899X/928/2/022047

Khammas, B. M., Hasan, S., Ahmed, R. A., Bassi, J. S., & Ismail, I. (2019). Accuracy Improved Malware Detection Method using Snort Sub-signatures and Machine Learning Techniques. *2018 10th Computer Science and Electronic Engineering Conference, CEEC 2018 - Proceedings*, 107–112. doi: 10.1109/CEEC.2018.8674233

Kim, J., Shim, M., Hong, S., Shin, Y., & Choi, E. (2020). Intelligent detection of iot botnets using machine learning and deep learning. *Applied Sciences (Switzerland)*, *10*(19), 1–22. doi: 10.3390/app10197009

Kohavi, R., & Provost, F. (2017). Special Issue on Applications of Machine Learning and the Knowledge Discovery Process. *Glossary of Terms journal of Machine Learning*, *30*(1998), 271–274. Retrieved from https://linkinghub.elsevier.com/retrieve/pii/B9780128123461000160

Koroniotis, N., Moustafa, N., Sitnikova, E., & Turnbull, B. (2019). Towards the development of realistic botnet dataset in the Internet of Things for network forensic analytics: Bot-IoT dataset. *Future Generation Computer Systems*, *100*, 779–796. Retrieved from https://doi.org/10.1016/j.future.2019.05.041 doi: 10.1016/j.future.2019.05.041

Liu, M., Xue, Z., Xu, X., Zhong, C., & Chen, J. (2019). Host-Based Intrusion Detection System with System Calls. *ACM Computing Surveys*, *51*(5), 1–36. doi: 10.1145/3214304

Makhdoom, I., Abolhasan, M., Lipman, J., Liu, R. P., & Ni, W. (2019). Anatomy of Threats to the Internet of Things. *IEEE Communications Surveys and Tutorials*, *21*(2), 1636–1675. doi: 10.1109/COMST.2018.2874978

MalwareBytes. (2021). *Backdoor computing attacks.* Retrieved 2021-06-01, from

https://www.malwarebytes.com/backdoor/

Mohammed, M. M. Z. E., Aleisa, E., & Ventura, N. (2014). An Automated Signature Generation Method for Zero-day Polymorphic Worms Based on C4.5 Algorithm. In *Icsea 2014* (pp. 215–220).

Monnappa, K. A. (2018). *Learning Malware Analysis : Explore the Concepts, Tools, and Techniques to Analyze and Investigate Windows Malware*. Packt Publishing Ltd.

Mudgerikar, A., Sharma, P., & Bertino, E. (2019). E-Spion: A system-level intrusion detection system for IoT devices. *AsiaCCS 2019 - Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 493–500. doi: 10.1145/3321705.3329857

Norouzi, M., Souri, A., & Samad Zamini, M. (2016). A Data Mining Classification Approach for Behavioral Malware Detection. *Journal of Computer Networks and Communications*, *2016*, 20–22. doi: 10.1155/2016/8069672

OWASP. (2018). *OWASP Internet of Things (IoT) Top 10 2018* (Tech. Rep.). Retrieved from https://owasp.org/www-pdf-archive/OWASP-IoT-Top-10-2018-final.pdf

Pa, Y. M. P., Suzuki, S., Yoshioka, K., Matsumoto, T., Kasama, T., & Rossow, C. (2016). IoTPOT: A novel honeypot for revealing current IoT threats. *Journal of Information Processing*, *24*(3), 522–533. doi: 10.2197/ipsjjip.24.522

Park, D., Powers, H., Prashker, B., Liu, L., & Yener, B. (2020). Towards obfuscated malware detection for low powered IoT devices. *arXiv*. Retrieved from http://arxiv.org/abs/2011.03476

Phu, T. N., Dang, K. H., Quoc, D. N., Dai, N. T., & Binh, N. N. (2019). A Novel Framework to Classify Malware in MIPS Architecture-Based IoT Devices. *Security and Communication Networks*, *2019*. doi: 10.1155/2019/4073940

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, *1*(1), 81–106. doi: 10.1007/bf00116251

Radovici, A., Rusu, C., & Serban, R. (2018). A Survey of IoT Security Threats and Solutions. *Proceedings - 17th RoEduNet IEEE International Conference: Networking in Education and Research, RoEduNet 2018*, 1–5. doi: 10.1109/ROEDUNET.2018.8514146

Sheikh, N. U., Rahman, H., Vikram, S., & AlQahtani, H. (2018). A lightweight signature-based IDS for IoT environment. *arXiv*.

Shobana, M., & Poonkuzhali, S. (2020). A novel approach to detect IoT malware by system calls using Deep learning techniques. *2020 International Conference on Innovative Trends in Information Technology, ICITIIT 2020*, 0–4. doi: 10.1109/ICITIIT49094.2020.9071531

Sikorski, M., & Honig, A. (2012a). Basic Dynamic Analysis. In *Practical malware analysis: The hands-on guide to dissecting malicious software*

(chap. 3).

Sikorski, M., & Honig, A. (2012b). Malware Analysis in Virtual Machines. In *Practical malware analysis: The hands-on guide to dissecting malicious software* (chap. 4). No Starch Press.

Singh, L., & Hofmann, M. (2018). Dynamic behavior analysis of android applications for malware detection. *ICCT 2017 - International Conference on Intelligent Communication and Computational Techniques*, *2018-Janua*(2013), 1–7. doi: 10.1109/INTELCCT.2017.8324010

Soe, Y. N., Feng, Y., Santosa, P. I., Hartanto, R., & Sakurai, K. (2019). *Rule Generation for Signature Based Detection Systems of Cyber Attacks in IoT Environments* (Vol. 8; Tech. Rep. No. 2). Retrieved from `http://ww.bncss.org/index.php/bncss/article/view/113/117`

Soe, Y. N., Feng, Y., Santosa, P. I., Hartanto, R., & Sakurai, K. (2020). Implementing Lightweight IoT-IDS on Raspberry Pi Using Correlation-Based Feature Selection and Its Performance Evaluation. In *Advances in intelligent systems and computing* (Vol. 926, pp. 458–469). Springer International Publishing. doi: 10.1007/978-3-030-15032-7_39

Sonicwall. (2021). *2021 SonicWall Cyber Threat Report* (Tech. Rep.).

Souri, A., & Hosseini, R. (2018). A state-of-the-art survey of malware detection approaches using data mining techniques. *Human-centric Computing and Information Sciences*, *8*(1). doi: 10.1186/s13673-018-0125-x

Tahir, R. (2018). A Study on Malware and Malware Detection Techniques. *International Journal of Education and Management Engineering*, *8*(2), 20–30. doi: 10.5815/ijeme.2018.02.03

Takase, H., Kobayashi, R., Kato, M., & Ohmura, R. (2020). A prototype implementation and evaluation of the malware detection mechanism for IoT devices using the processor information. *International Journal of Information Security*, *19*(1), 71–81. doi: 10.1007/s10207-019-00437-y

Thielman, S., & Johnston, C. (2016). *Major cyber attack disrupts internet service across Europe and US.* Retrieved 2021-06-02, from `https://www.theguardian.com/technology/2016/oct/21/ddos-attack-dyn-internet-denial-service`

Tian, R., Islam, R., Batten, L., & Versteeg, S. (2010). Differentiating malware from cleanware using behavioural analysis. *Proceedings of the 5th IEEE International Conference on Malicious and Unwanted Software, Malware 2010*, 23–30. doi: 10.1109/MALWARE.2010.5665796

Uhvrivcek, D. (2019). LiSa – Multiplatform Linux Sandbox for Analyzing IoT Malware. In *Excel@fit 2019.*

Wan, T.-L., Ban, T., Cheng, S.-M., Lee, Y.-T., Sun, B., Isawa, R., ... Inoue, D. (2020). Efficient Detection and Classification of Internet-of-Things Malware Based on Byte Sequences from Executable Files. *IEEE Open*

Journal of the Computer Society, *1*(November), 262–275. doi: 10.1109/ojcs.2020.3033974

Wang, A., Liang, R., Liu, X., Yingjun, Z., Chen, K., & Li, J. (2017). An Inside Look at IoT Malware. In *Second eai international conference, industrial iot 2017*.

Wikipedia. (2021a). *BASHLITE*. Retrieved 2021-06-02, from https://en.wikipedia.org/wiki/BASHLITE

Wikipedia. (2021b). *C4.5 Algorithm*. Retrieved 2021-06-17, from https://en.wikipedia.org/wiki/C4.5_algorithm

Wikipedia. (2021c). *Decision tree learning*. Retrieved 2021-06-17, from https://en.wikipedia.org/wiki/Decision_tree_learning

Wikipedia. (2021d). *System Call*. Retrieved 2021-06-09, from https://en.wikipedia.org/wiki/System_call

Ye, Y., Li, T., Adjeroh, D., & Iyengar, S. S. (2017). A survey on malware detection using data mining techniques. *ACM Computing Surveys*, *50*(3), 33. doi: 10.1145/3073559

Yuxin, D., Xuebing, Y., Di, Z., Li, D., & Zhanchao, A. (2011). Feature representation and selection in malicious code detection methods based on static system calls. *Computers and Security*, *30*(6-7), 514–524. Retrieved from http://dx.doi.org/10.1016/j.cose.2011.05.007 doi: 10.1016/j.cose.2011.05.007

Zhou, W., Zhang, Y., & Liu, P. (2019). The Effect of IoT new features on security and privacy: New threats, existing solutions, and challenges yet to be solved. *IEEE Internet of Things Journal*, *6*(2), 1606–1616.

Appendices

Appendix A

Project Management

A.1 Project Plan

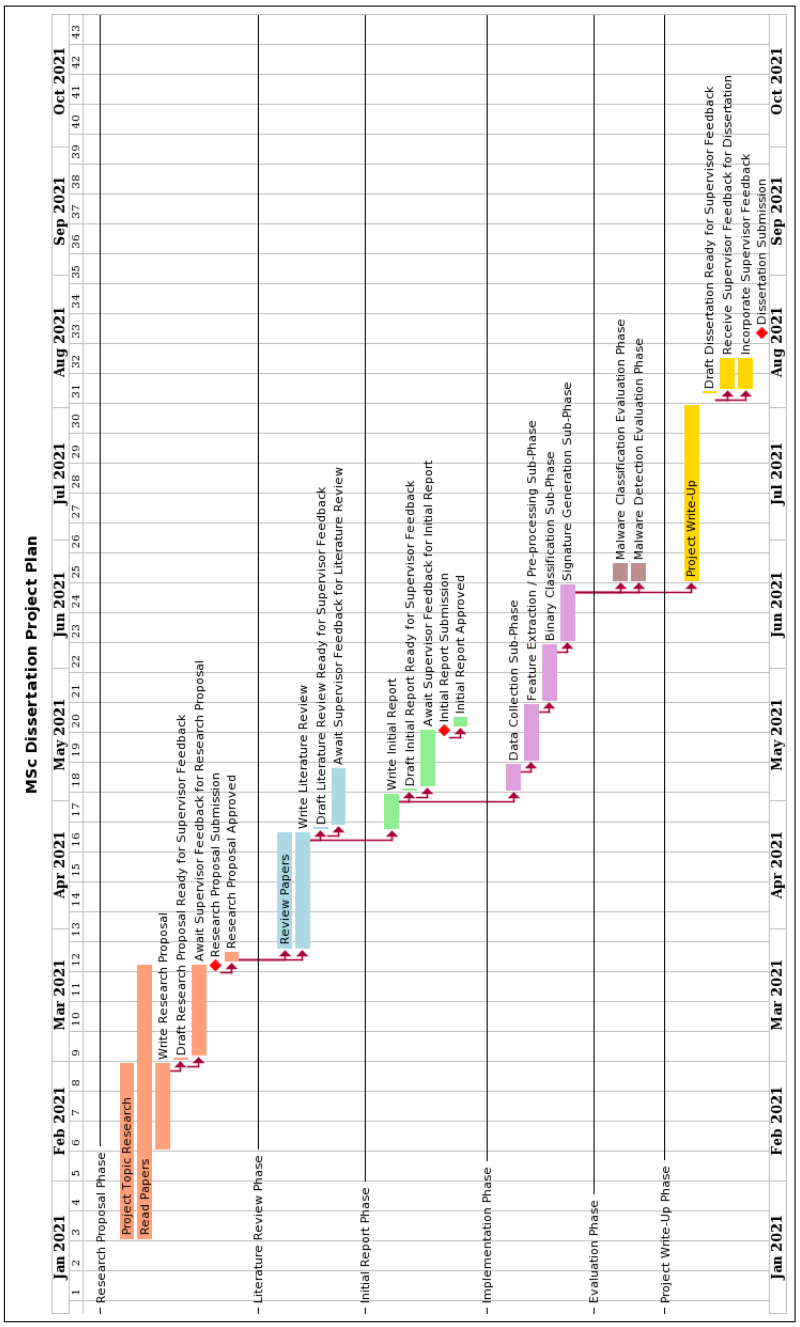


Figure A.1: Project Plan

A.2 Project Diaries

EDINBURGH NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Fiona Walsh **Supervisor: Rich MacFarlane**

Date: 27th Jan 2021 **Last diary date: First Entry**

Objectives:

- To familiarise myself with the dissertation process and module organisation
- To read about IoT Ransomware

Progress:

31.08.2020 – 13.10.2020

- Read through the papers suggested by Rich and made contact with Simon Davies (as suggested by Rich) as he is currently working in the field of Ransomware research.

27.12.2020

- Read the Module Organiser. Watched the SoC's recordings of the "The dissertation process" and "The Dissertation Lifecycle".
- Read papers on the area of IoT Ransomware.

28.12.2020

- Registered my project topic and supervisor (Rich MacFarlane) through the Project and Supervision Registration form on Moodle.

29.12.2020

- Read a number of articles on Ransomware attacks i.e., Sodinokibi, Locky, Cerber, SamSam, GandCrab, WannaCry, NotPetya and Bad Rabbit.

04.01.2021 – 11.01.2021

- Looked into specific ransomware attacks on IoT devices i.e., Thermostat hacking, Smart Bulb, Lizard Stressor, Fiat Chrysler, Flocker (Frantic Locker) etc.
- Read a selection of research papers on the topics of IoT Ransomware, IoT Forensics and Intrusion Detection Systems for IoT.

12.01.2021

- Joined the webex entitled "SoC Masters Dissertation - Getting started Workshop".

14.01.2021

- Established a potential dissertation topic/title: *Evaluation of the suitability and effectiveness of Host-Based Intrusion Detection Systems in the detection of Ransomware on IoT devices* and received initial approval from Rich to proceed with researching that topic.

19.01.2021

- Read 'Guide to Writing Your Research Proposal'.

25.01.2021

- Wrote a one pager (ResearchTopic_OnePager.docx) on my research area and potential project ideas for discussion at first supervisory meeting on 27/01/21.

Supervisor's Comments:

Figure A.2: 20210127 Project Diary

EDINBURGH NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Fiona Walsh **Supervisor: Rich MacFarlane**

Date: 9th Feb 2021 **Last diary date: 27th Jan 2021**

Objectives:

- To identify 1-2 papers on Ransomware on IoT devices and critically analyse them
- To create a draft project plan
- To familiarise myself with the marking scheme

Progress:

28.01.2021

- Read through _Cyber Security Dissertation Project.pdf and _MSc Dissertation check list - Marking scheme _ Richs extra examples.pdf.

30.01.2021

- Created a draft project plan (Project_Plan.xlsx) and uploaded it to my shared folder on dropbox (URL: https://www.dropbox.com/scl/fi/0l190zqnaww7ye8vzepk7/Project_Plan.xlsx?dl=0&rlkey=aqwbndyzdo3otjlgkd4rq1pxo)
- Identified the following papers to critically analyse:
  - Low-Complexity Signature-Based Malware Detection for IoT Devices (Abbas and Srikanthan, 2017)
  - A Novel Framework to Classify Malware in MIPS Architecture-Based IoT Devices (Phu et al., 2019)

09.02.2021

- Critically analysed both papers (mentioned above) and wrote up some notes about what I could do in ResearchTopic_FocusPapers.docx (URL: https://www.dropbox.com/scl/fi/44ggji3q45u1v9ulwnljm/ResearchTopic_OnePager.docx?dl=0&rlkey=857t4o6fafhmzw311t6msrsdu)

Supervisor's Comments:

Figure A.3: 20210209 Project Diary

EDINBURGH NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Fiona Walsh **Supervisor: Rich MacFarlane**

Date: 28th Feb 2021 **Last diary date: 9th Feb 2021**

Objectives:

- To look into papers (specifically surveys) on IoT Ransomware and find out what it does (on the device) and how it does it
- To create an initial draft of my research proposal

Progress:

11.02.2021 - 26.02.2021

- Critically analysed a selection of papers on Malware Detection and wrote up some notes what type of malware they covered, whether it was IoT related and finally how it was detected in ResearchTopic_IoTRansomware_Focus.docx (URL: https://www.dropbox.com/scl/fi/193lu5jc1gx5wd3vtxwog/FWalsh_ResearchProposal.docx?dl=0&rlkey=o7h88kyaoqfd80n2uymbzzlc4)

27.02.2021 - 28.02.2021

- I switched focus from IoT Ransomware (as it was too narrow in terms of papers and also as I had concerns, I wouldn't be able to find enough samples for a dataset for my practical work) to IoT malware in general.
- The new working title of my project is: Evaluation of Lightweight Signature-Based Malware Detection on IoT devices.
- Created an initial draft of my research proposal in FWalsh_ResearchProposal.docx (URL: https://www.dropbox.com/scl/fi/193lu5jc1gx5wd3vtxwog/FWalsh_ResearchProposal.docx?dl=0&rlkey=o7h88kyaoqfd80n2uymbzzlc4).
- Sent draft to Rich for feedback, aiming to submit my proposal to Moodle by 7th March at latest.

Supervisor's Comments:

Figure A.4: 20210228 Project Diary

EDINBURGH NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Fiona Walsh **Supervisor: Rich MacFarlane**

Date: 24th Mar 2021 **Last diary date: 28th Feb 2021**

Objectives:

- To go through my Research Proposal with Rich, incorporate feedback and get it approved by my internal examiner
- To investigate tools for the practical elements of my project

Progress:

01.03.2021 - 24.03.2021

- Research Proposal with Rich. Feedback received on 24th March, submitted to Moodle the same day.
- Research Proposal approved by Internal Examiner on 25th March.
- Investigated and setup LiSa (Multiplatform Linux Sandbox for Analyzing IoT Malware - https://github.com/danieluhricek/LiSa). Testing with benign binaries for different architectures and the sandbox is working as expected.
- Investigated and setup TheZoo (https://github.com/ytisf/theZoo) – a repository of live malwares – initial investigation show's that it only has a limited number of malwares targeting the IoT type architectures i.e. ARM (11), MIPS (0), SPARC (0), AArch64 (0) and PowerPC (0) and the latest one is from 2018 i.e. Skygofree.

Supervisor's Comments:

Figure A.5: 20210324 Project Diary

EDINBURGH NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Fiona Walsh **Supervisor: Rich MacFarlane**

Date: 4th May 2021 **Last diary date: 24th Mar 2021**

Objectives:

- To create an initial draft of my initial report.
- To critically analyse papers core papers for my literature review.
- To start exploratory practical work for my data collection phase.

Progress:

06.04.2021 - 03.05.2021

- Critically analysed core papers for my literature review.
  - See https://www.dropbox.com/scl/fi/q1bomwmuhzuskwgiro5wx/ResearchTopic_MalwareDetection-Classification_Focus.docx?dl=0&rlkey=mrtcck1ffrasuj8ui5431f2k8

17.04.2021 - 28.04.2021

- Created a draft of my initial report, ready for feedback from my supervisor.
  - See https://www.dropbox.com/scl/fi/reyx9nu90tx0k9o10s6ig/FWalsh_InitialReport.docx?dl=0&rlkey=qa17s83y41bb2h1dz7x36s1aa

19.04.2021 – 03.05.2021

- Acquired a dataset (IoTPot) from Yokohama National University of IoT Malware executables. Have done some exploratory runs in LiSa (Sandox for IoT Malware Analysis) and captured sys call traces for 171 samples.

Supervisor's Comments:

Figure A.6: 20210504 Project Diary

EDINBURGH NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Fiona Walsh **Supervisor: Rich MacFarlane**

Date: 11th May 2021 **Last diary date: 4th May 2021**

Objectives:

- To write the introduction chapter of my dissertation
- Research setting up VMs for malware analysis

Progress:

04.05.2021 - 10.05.2021

- Introduction chapter written and supervisor feedback given in today's meeting.

10.05.2021 - Present

- Reading about host-only connections and fake network services for VMs in the following books:
  - *Learning Malware Analysis - Explore the Concepts, Tools, and Techniques to Analyze and Investigate Windows Malware* By Monnappa K A
  - *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software* By Michael Sikorski and Andrew Honig

Supervisor's Comments:

Private

Figure A.7: 20210511 Project Diary

EDINBURGH NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Fiona Walsh **Supervisor: Rich MacFarlane**

Date: 28th May 2021 **Last diary date: 11th May 2021**

Objectives:

- To gather a sample dataset of benign samples and verify the malware samples already collected
- To discuss best tools for sample collection and dataset creation with Simon Davies (PhD Student)

Progress:

12.05.2021 - 27.05.2021

- Identified suitable IoT firmware downloads freely available on-line (manufacturers i.e. D-Link)
- Explored using BinWalk to extract IoT firmware's file systems and then search them for ELF executables (benign samples)
- Also acquired an increased quota (20k requests per day) on my VirusTotal Public API account for verifying malware.

Supervisor's Comments:

Private

Figure A.8: 20210528 Project Diary

EDINBURGH NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Fiona Walsh **Supervisor: Rich MacFarlane**

Date: 8th June 2021 **Last diary date: 28th May 2021**

Objectives:

- To complete my literature review chapter

Progress:

28.05.2021 - Present

- Literature review chapter is in progress. Key papers have been selected and sections headings drafted.

04.06.2021 - 06.06.2021

- Setup two VMs on a host-only connection. One is a victim host and the other provides network services (via iNetSim). Tested connections between them and confirmed that the victim host vm can find all standard services i.e. HTTP, FTP etc. on the network services vm.

Supervisor's Comments:

Private

Figure A.9: 20210608 Project Diary

EDINBURGH NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Fiona Walsh **Supervisor: Rich MacFarlane**

Date: 22nd June 2021 **Last diary date: 8th June 2021**

Objectives:

- To complete my literature review chapter
- To start the practical work for my dissertation

Progress:

20.06.2021

- Literature review chapter completed and sent to supervisor for feedback.

20.06.2021 - Present

- Implementation phase started. Malware and Benign samples selected and ran through the LiSa sandbox. System call trace logs collected.

Supervisor's Comments:

Private

Figure A.10: 20210622 Project Diary

EDINBURGH NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Fiona Walsh **Supervisor: Rich MacFarlane**

Date: 1st July 2021 **Last diary date: 22nd June 2021**

Objectives:

- To get feedback on my literature review chapter
- To continue with the practical work for my dissertation

Progress:

20.06.2021 - 30.06.2021

- Development (python scripts) for implementation phase started.

01.07.2021

- Feedback on literature review chapter received from supervisor.

Supervisor's Comments:

Private

Figure A.11: 20210701 Project Diary

EDINBURGH NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Fiona Walsh **Supervisor: Rich MacFarlane**

Date: 27th July 2021 **Last diary date: 1st July 2021**

Objectives:

- To complete the practical work for my dissertation
- To start the project write-up phase

Progress:

01.07.2021 - 07.07.2021

- Development (python scripts) for implementation phase completed. Shell script created to automate the experimental runs.

08.07.2021

- Experiments ran using automated process.

09.07.2021 - 21.07.2021

- Writing of the design chapter started and completed.

23.07.2021 - Present

- Writing the implementation chapter started.

Supervisor's Comments:

Private

Figure A.12: 20210727 Project Diary

Appendix B

Source Code

B.1 csv_to_json_convertor.py

```
#!/usr/bin/env python

# Converts the CSV output from VirusTotal to a JSON file in the required
↪ format for AVClass2.

# import packages
import argparse
import glob
import os
import csv
import json


def convert_to_json_file(csvFilePath, jsonFilePath):
    if os.path.isfile(jsonFilePath):
        os.remove(jsonFilePath)

    # create new json file
    jsonFile = open(jsonFilePath, 'a')

    # read csv file and add to data
    data = {}
    with open(csvFilePath) as csvFile:
```

```
23          csvReader = csv.DictReader(csvFile, delimiter=";")
24          for csvRow in csvReader:
25              if csvRow["Response"] == "1":
26                  data["sha1"] = csvRow["sha1"]
27                  data["av_labels"] = extract_av_results(csvRow["AVs"])
28                  data["scan_date"] = csvRow["Scan Date"]
29                  data["sha256"] = csvRow["sha256"]
30                  data["md5"] = csvRow["md5"]
31
32                  # write data to it
33                  jsonFile.write(json.dumps(data) + "\n")
34
35      # close file
36      jsonFile.close()
37
38
39  def extract_av_results(av_results):
40      extracted_av_results = []
41      raw_av_results = av_results.split(",")
42
43      for av_result in raw_av_results:
44          extracted_av_results.append(av_result.split("#"))
45      return extracted_av_results
46
47
48  if __name__ == '__main__':
49      print('Running Script: {}'.format(__file__))
50
51      parser = argparse.ArgumentParser(prog=__file__,
52                                       usage='%(prog)s [options]',
53                                       description='Performs malware
                                         ↪  classification')
54
55      # -i inputDir -o outputDir
56      parser.add_argument("-i", "--input", help="Input directory")
57      parser.add_argument("-o", "--output", help="Output directory")
58
59      args = parser.parse_args()
60
61      count = 1
```

```
62      for file in glob.glob(os.path.join(args.input, "*.csv")):
63          print("Reading csv #{}: {}".format(count, file))
64          convert_to_json_file(file, args.output + file[file.rfind("/") +
            ↪  1:-4] + ".json")
65          count += 1
```

B.2 dataset_generator.py

```
1   #!/usr/bin/env python
2
3   # Generates a dataset file (.dat) for each of the JSON files produced by
    ↪  LiSa.  This file contains the first x System Calls minus their input
    ↪  and return parameters.
4
5   # Load Libraries
6   import pandas as pd
7   import argparse
8   import glob
9   import os
10  import datetime
11
12  # monkeypatch using faster simplejson module
13  import simplejson
14
15  pd.io.json._json.loads = lambda s, *a, **kw: simplejson.loads(s)
16
17
18  def read_json_syscalls(json, args, now):
19      df = pd.read_json(json)
20      syscall_names = df.dynamic_analysis.syscalls
21      norm_table = pd.json_normalize(syscall_names)
22      sys_rows = pd.DataFrame(norm_table["name"].values)
23      sys_rows = sys_rows.iloc[:args.size].T
24      sys_rows.columns = ["syscall" + str(i) for i in
25                          range(sys_rows.shape[1])]  # name all columns as
                            ↪  syscallN where N is the number of the
                            ↪  syscall
```

```
26
27      file_name = df.file_name.syscalls
28      dataset = file_name + "_" + args.type + ".dat"
29      sys_rows.to_csv(args.output + "/" + dataset, header=None, index=None,
        ↪  sep=' ', mode='w')
30
31
32  if __name__ == '__main__':
33      print('Running Script: {}'.format(__file__))
34
35      parser = argparse.ArgumentParser(prog=__file__,
36                                       usage='%(prog)s [options]',
37                                       description='Generates System Call
                                         ↪  datasets')
38
39      parser.add_argument("-i", "--input", help="Input directory")
40      parser.add_argument("-o", "--output", help="Output directory")
41      parser.add_argument("-t", "--type", help="Sample type i.e. benign or
        ↪  malware family")
42      parser.add_argument("-s", "--size", help="Number of syscalls to use",
        ↪  type=int)
43
44      args = parser.parse_args()
45
46      print('\n\nArguments:\t{}\n\n'.format('\n\t\t'.join(f'{k}={v}' for k,
        ↪  v in vars(args).items())))
47
48      utc_time = datetime.datetime.utcnow()
49      now = utc_time.strftime('%Y%m%d%H%M%S')
50
51      count = 1
52      for file in glob.glob(os.path.join(args.input, "*.json")):
53          print("Reading log #{}: {}".format(count, file))
54          read_json_syscalls(file, args, now)
55          count += 1
```

B.3 feature_extraction_selection.py

```
#!/usr/bin/env python

# Creates System Call N-grams based on the specified parameter.  Calculates the TF-IDF weight for each of the System Call N-grams and discards all System Call N-grams outside the top N features (ordered by term frequency across the corpus of .dat files).  Converts all non-zero TF-IDF weights to 1.  A non-zero TF-IDF weight means that the System Call N-gram appears in particular trace log and as such can potentially be used for classification.  The results are formatted and output as a CSV file.

import argparse
import datetime
import glob
import os
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer


def perform_fet_ext_sel(arguments, now):
    raw_docs_filepath = glob.glob(os.path.join(arguments.input, "*.dat"))

    # Creates an instance of TfidfVectorizer
    tfidf = TfidfVectorizer(input='filename', ngram_range=(arguments.ngram, arguments.ngram),
                            max_features=arguments.topn)

    # Convert the collection of .DAT files to a matrix of TF-IDF features
    response = tfidf.fit_transform(raw_docs_filepath)

    # Create a dataframe to store the formatted results
    syscall_df = pd.DataFrame()
    syscall_df['system_calls'] = tfidf.get_feature_names()

    label_df = pd.DataFrame()

    count = 0
```

```
30      malware_count = 0
31      benign_count = 0
32
33      # Populate the dataframe with the TF-IDF weights
34      with open(args.output + "feature_ext_sel_processed_files.log", 'w')
        ↪  as processed_files:
35          for file in raw_docs_filepath:
36              syscall_df[str(count + 1)] = response.toarray()[count]
37              filetype = file[file.rfind("_") + 1:-4]
38
39              if filetype == "benign":
40                  benign_count += 1
41              else:
42                  malware_count += 1
43
44              label_df[str(count + 1)] = [filetype]
45              count += 1
46
47              processed_files.write("Processed file: {}".format(file))
48
49      print("Total Syscall Logs Processed: {} (Malware: {}, Benign:
        ↪  {})".format(count, malware_count, benign_count))
50
51      syscall_df.set_index('system_calls', inplace=True)
52
53      # Convert all TF-IDF scores greater than 0 to 1
54      syscall_df[syscall_df > 0] = 1
55
56      # Format final results before saving to CSV
57      final_df = pd.concat([syscall_df.T, label_df.T], axis=1)
58      final_df.columns = [*final_df.columns[:-1], 'label']
59      filename = arguments.output + "syscall_top" + str(arguments.topn) +
        ↪  "_features_" + now + ".csv"
60      final_df.to_csv(filename, index=False, sep=',', mode='a')
61
62      print("\nCreated system call top features datatset file:
        ↪  {}".format(filename))
63
64
65  if __name__ == '__main__':
```

```
66      print('Running Script: {}'.format(__file__))
67
68      parser = argparse.ArgumentParser(prog=__file__,
69                                       usage='%(prog)s [options]',
70                                       description='Performs feature
                                          ↪  extraction and selection')
71
72      parser = argparse.ArgumentParser()
73
74      parser.add_argument("-i", "--input", help="Input directory")
75      parser.add_argument("-o", "--output", help="Output directory")
76      parser.add_argument("-n", "--ngram", default=3, help="Size of
         ↪  N-grams", type=int)
77      parser.add_argument("-t", "--topn", default=20, help="Top max
         ↪  features ordered by term frequency across the corpus",
78                          type=int)
79
80      args = parser.parse_args()
81
82      print('Arguments:\t{}\n\n'.format('\n\t\t'.join(f'{k}={v}' for k, v
         ↪  in vars(args).items())))
83
84      utc_time = datetime.datetime.utcnow()
85      now = utc_time.strftime('%Y%m%d%H%M%S')
86
87      perform_fet_ext_sel(args, now)
```

B.4 malware_classification.py

```
1   #!/usr/bin/env python
2
3   # Performs malware classification
4
5   # Load libraries
6   import argparse
7   from pathlib import Path
```

```
8  import numpy as np
9  import pandas as pd
10 import pydotplus
11 from IPython.display import Image
12 from matplotlib import pyplot as plt
13 from six import StringIO
14 from sklearn.metrics import classification_report, plot_confusion_matrix
15 from sklearn.tree import DecisionTreeClassifier, _tree
16 from sklearn.model_selection import train_test_split
17 from sklearn.tree import export_graphviz
18
19
20 def perform_classification(arguments, dataset):
21     # Load dataset
22     train_test_dataset = pd.read_csv(dataset)
23
24     # Divided the dataset in features and target variable
25     feature_cols =
       ↪ train_test_dataset.columns[:train_test_dataset.columns.size - 1]
26     x = train_test_dataset[feature_cols]  # Features
27     y = train_test_dataset.label  # Target variable
28
29     # Split dataset into training and test sets
30     x_train, x_test, y_train, y_test = train_test_split(x, y,
       ↪ test_size=arguments.test_size, random_state=1, stratify=y)
31
32     # Create Decision Tree classifier object
33     if arguments.tree_depth > 0:
34         dt_clf = DecisionTreeClassifier(max_depth=arguments.tree_depth)
35     else:
36         dt_clf = DecisionTreeClassifier()
37
38     # Train Decision Tree Classifier
39     dt_clf = dt_clf.fit(x_train, y_train)
40
41     # Predict the response for test dataset
42     y_pred = dt_clf.predict(x_test)
43
44     # Model Results
45     print('################### - RESULTS - ###################')
```

```
46      print('\nClassification Report:\n')
47      print(classification_report(y_test, y_pred, digits=4,
        ↪  zero_division=0))
48      print('#####################################################\n')
49      fig, ax = plt.subplots(figsize=(10, 6))
50      plot_confusion_matrix(dt_clf, x_test, y_test, cmap="PuRd", ax=ax)
51      plt.savefig(arguments.output + dataset.stem +
        ↪  "_confusion_matrix.png")
52      print("Saved Confusion Matrix to {}".format(arguments.output +
        ↪  dataset.stem + "_confusion_matrix.png"))
53
54      # Visualizing Decision Trees
55      dot_data = StringIO()
56      export_graphviz(dt_clf, out_file=dot_data, filled=True, rounded=True,
        ↪  special_characters=True,
57                      feature_names=feature_cols,
                        ↪  class_names=dt_clf.classes_)
58      graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
59      graph.write_png(arguments.output + dataset.stem +
        ↪  "_decision_tree.png")
60      Image(graph.create_png())
61
62      print("Saved Decision Tree Image to {}".format(arguments.output +
        ↪  dataset.stem + "_decision_tree.png"))
63
64      # Export rules
65      rules = get_rules(dt_clf, list(x_train.columns), dt_clf.classes_)
66      with open(arguments.output + dataset.stem +
        ↪  "_decision_tree_rules.txt", 'w') as file:
67          for r in rules:
68              file.write(r + "\n")
69      print("Saved Decision Tree Rules to {}".format(arguments.output +
        ↪  dataset.stem + "_decision_tree_rules.txt"))
70
71
72  # Author: Piotr Płoński
73  def get_rules(tree, feature_names, class_names):
74      tree_ = tree.tree_
75      feature_name = [
76          feature_names[i] if i != _tree.TREE_UNDEFINED else "undefined!"
```

```
77          for i in tree_.feature
78      ]
79
80      paths = []
81      path = []
82
83      def recurse(node, path, paths):
84
85          if tree_.feature[node] != _tree.TREE_UNDEFINED:
86              name = feature_name[node]
87              threshold = tree_.threshold[node]
88              p1, p2 = list(path), list(path)
89              p1 += [f"({name} <= {np.round(threshold, 3)})"]
90              recurse(tree_.children_left[node], p1, paths)
91              p2 += [f"({name} > {np.round(threshold, 3)})"]
92              recurse(tree_.children_right[node], p2, paths)
93          else:
94              path += [(tree_.value[node], tree_.n_node_samples[node])]
95              paths += [path]
96
97      recurse(0, path, paths)
98
99      # sort by samples count
100     samples_count = [p[-1][1] for p in paths]
101     ii = list(np.argsort(samples_count))
102     paths = [paths[i] for i in reversed(ii)]
103
104     rules = []
105     for path in paths:
106         rule = "if "
107
108         for p in path[:-1]:
109             if rule != "if ":
110                 rule += " and "
111             rule += str(p)
112         rule += " then "
113         if class_names is None:
114             rule += "response: " + str(np.round(path[-1][0][0][0], 3))
115         else:
116             classes = path[-1][0][0]
```

```
117             l = np.argmax(classes)
118             rule += f"class: {class_names[l]}"
119         rules += [rule]
120
121     return rules
122
123
124 if __name__ == '__main__':
125     print('Running Script: {}'.format(__file__))
126
127     parser = argparse.ArgumentParser(prog=__file__,
128                                      usage='%(prog)s [options]',
129                                      description='Performs malware
                                         ↪ classification')
130
131     parser.add_argument("-i", "--input", help="Input directory")
132     parser.add_argument("-o", "--output", help="Output directory")
133     parser.add_argument("-t", "--test_size", help="Test split i.e. 0.3
         ↪ (30% of dataset should be used for test)",
134                         default=0.3, type=float)
135     parser.add_argument("-d", "--tree_depth", help="Max tree depth i.e.
         ↪ 2", default=0, type=int)
136
137     args = parser.parse_args()
138
139     print('Arguments:\t{}\n\n'.format('\n\t\t'.join(f'{k}={v}' for k, v
         ↪ in vars(args).items())))
140
141     dataset_files = Path(args.input).glob("*.csv")
142     latest_dataset = max(dataset_files, key=lambda f: f.stat().st_mtime)
143
144     print("Processing file: {}\n".format(latest_dataset))
145
146     perform_classification(args, latest_dataset)
```

B.5 signature_generator.py

```
#!/usr/bin/env python

# Generates YARA signatures from a set of decision tree rules
import argparse
import datetime
from pathlib import Path


def generate_signatures(arguments, rules):
    utc_time = datetime.datetime.utcnow()

    # Reads in the decision tree rules file
    with open(rules) as rules_file:
        lines = rules_file.readlines()

    # Build a YARA signatures file
    signatures_file = arguments.output + rules.stem[:-20] +
    ↪  "_signatures.yar"
    with open(signatures_file, 'w') as file:

        yara_rule_meta = "\t\tauthor = \"" + arguments.author +
        ↪  "\"\n\t\tdate = \"" + utc_time.strftime('%d/%m/%Y') \
                        + "\""

        rule_count = 0
        for line in lines:
            # Create YARA signatures for malware only
            if "benign" not in line:
                yara_rule = line.split("and")
                yara_rule_condition = ""
                rule_count += 1
                yara_rule_name = "rule{}_{}
                ↪  ".format(yara_rule[yara_rule.__len__() -
                ↪  1][str.find(yara_rule[yara_rule.__len__() - 1], ":")
                ↪  + 1:-1], rule_count)
                file.write(yara_rule_name + "{\n")

                # Build 'meta' and 'strings' for YARA signature
```

```
34                  file.write("\tmeta:\n {}\n".format(yara_rule_meta))
35                  file.write("\tstrings:\n")
36
37                  # Build 'condition' for YARA signature
38                  for string_count in range(0, yara_rule.__len__()):
39                      if str.find(yara_rule[string_count], "<") > 0:
40                          yara_string = "$syscall_ngram_{} =
                            ↪  \"{}\"".format(str(string_count + 1),
                            ↪  yara_rule[string_count]
41                          [str.find(yara_rule[string_count], "(") +
                            ↪  1:str.find(yara_rule[string_count], "<") -
                            ↪  1]) + "\n"
42                          yara_rule_condition += "(not
                            ↪  $syscall_ngram_{}".format(str(string_count +
                            ↪  1)) + ") and "
43                      else:
44                          yara_string = "$syscall_ngram_{} =
                            ↪  \"{}\"".format(str(string_count + 1),
                            ↪  yara_rule[string_count]
45                          [str.find(yara_rule[string_count], "(") +
                            ↪  1:str.find(yara_rule[string_count], ">") -
                            ↪  1]) + "\n"
46                          yara_rule_condition +=
                            ↪  "$syscall_ngram_{}".format(str(string_count +
                            ↪  1)) + " and "
47                      file.write("\t\t" + yara_string)
48
49                  yara_rule_condition += "(filesize > 0)"
50                  file.write("\n\tcondition:\n")
51                  file.write("\t\t" + yara_rule_condition + "\n}\n\n")
52
53      print("Saved {} generated YARA Signatures to {}".format(rule_count,
        ↪  arguments.output + signatures_file))
54
55
56  if __name__ == '__main__':
57      print('Running Script: {}'.format(__file__))
58
59      parser = argparse.ArgumentParser(prog=__file__,
60                                       usage='%(prog)s [options]',
```

```
                                    description='Performs signature
                                    ↪  generation')

    parser = argparse.ArgumentParser()

    parser.add_argument("-i", "--input", help="Input directory")
    parser.add_argument("-o", "--output", help="Output directory")
    parser.add_argument("-a", "--author", help="Author",
    ↪  default="FWalsh")

    args = parser.parse_args()

    print('Arguments:\t{}\n'.format('\n\t\t'.join(f'{k}={v}' for k, v in
    ↪  vars(args).items())))

    decision_tree_rules_files = Path(args.input).glob("*.txt")
    latest_decision_tree_rules = max(decision_tree_rules_files,
    ↪  key=lambda f: f.stat().st_mtime)

    print("Processing file: {}\n".format(latest_decision_tree_rules))

    generate_signatures(args, latest_decision_tree_rules)
```

B.6 malware_detection.py

```
#!/usr/bin/env python

# Performs malware detection on System Call Dataset files using YARA
↪  signatures
import argparse
import glob
import os
from pathlib import Path
import yara


```

```
11  def perform_mal_detection(arguments, latest_signatures):
12      # Compile YARA signatures and raise any errors found
13      signatures = yara.compile(str(latest_signatures))
14
15      total_benign_count = 0
16      total_malware_count = 0
17      overall_tp_count = 0
18      overall_fp_count = 0
19      overall_tn_count = 0
20      overall_fn_count = 0
21
22      print("Matching source files to YARA signatures......")
23      with open(args.output + "yara_match_results.txt", 'w') as
        ↪  yara_results_file:
24          input_files = glob.glob(os.path.join(arguments.input, "*.dat"))
25          for file in input_files:
26              filetype = file[file.rfind("_")+1:-4]
27
28              if filetype == "benign":
29                  total_benign_count += 1
30              else:
31                  total_malware_count += 1
32
33              with open(file, 'rb') as f:
34                  # Check for any matches to the auto-generated YARA
                    ↪  signatures and record the results
35                  matches = signatures.match(data=f.read())
36                  if len(matches) > 0:
37                      yara_results_file.write("{} matches the following
                        ↪  YARA rule(s): {}\n".format(file, matches))
38                      if filetype == "benign":
39                          overall_fp_count += 1
40                      else:
41                          overall_tp_count += 1
42                  else:
43                      yara_results_file.write("{} does not match any YARA
                        ↪  signatures.\n".format(file))
44                      if filetype == "benign":
45                          overall_tn_count += 1
46                      else:
```

```
47                          overall_fn_count += 1
48      print("Results saved to {}".format(args.output +
        ↪  "yara_match_results.txt"))
49      print('\n#################### - RESULTS - ####################')
50      print("Actual Malware Count: {} | Actual Benign Count:
        ↪  {}\n".format(total_malware_count,
51
                                                                    ↪  total_b
52      print("TP (True Positive) Count: {}, TPR:
        ↪  {}".format(overall_tp_count, "{:.2%}".format(
53                                                  (overall_tp_count /
                                                    ↪  (overall_tp_count
                                                    ↪  +
                                                    ↪  overall_fn_count)))))
54      print("TN (True Negative) Count: {}, TNR:
        ↪  {}".format(overall_tn_count, "{:.2%}".format(
55                                                  (overall_tn_count /
                                                    ↪  (overall_tn_count
                                                    ↪  +
                                                    ↪  overall_fp_count)))))
56      print("FP (False Positive) Count: {}, FPR:
        ↪  {}".format(overall_fp_count, "{:.2%}".format(
57                                                  (overall_fp_count /
                                                    ↪  (overall_tn_count
                                                    ↪  +
                                                    ↪  overall_fp_count)))))
58      print("FN (False Negative) Count: {}, FNR:
        ↪  {}".format(overall_fn_count, "{:.2%}".format(
59                                                  (overall_fn_count /
                                                    ↪  (overall_tp_count
                                                    ↪  +
                                                    ↪  overall_fn_count)))))
60      print('#####################################################')
61
62
63  if __name__ == '__main__':
64      print('Running Script: {}'.format(__file__))
65
66      parser = argparse.ArgumentParser(prog=__file__,
67                                       usage='%(prog)s [options]',
```

```
                                        description='Performs malware
                                        ↪  detection')

    parser = argparse.ArgumentParser()

    parser.add_argument("-i", "--input", help="Input directory")
    parser.add_argument("-o", "--output", help="Output directory")
    parser.add_argument("-s", "--signatures", help="Signatures
    ↪  directory")

    args = parser.parse_args()

    print('Arguments:\t{}\n'.format('\n\t\t'.join(f'{k}={v}' for k, v in
    ↪  vars(args).items())))

    signature_files = Path(args.signatures).glob("*.yar")
    latest_signatures = max(signature_files, key=lambda f:
    ↪  f.stat().st_mtime)

    print("Processing file: {}\n".format(latest_signatures))

    perform_mal_detection(args, latest_signatures)
```

B.7 main.sh

```
#!/bin/bash

INPUT=$1
[ ! -f $INPUT ] && { echo "$INPUT file not found"; exit 99; }

syscall_dataset_dir="/home/datausr/01_DataAnalysis/02_SysCall_Datasets/"

sed 1d $INPUT | while IFS=',', read -r run_num fes_ngram fes_topn
↪  mc_tree_depth
do
 start=$(date +%s.%N)
```

```
11  echo "Run Num: $run_num"
12
13  EXP_RUN_DIR="/home/datausr/02_Experiments/exp_results_run_num_$run_num/"
14  if [ -d "$EXP_RUN_DIR" ]; then rm -Rf $EXP_RUN_DIR; fi
15  mkdir $EXP_RUN_DIR
16
17  #1. Perform feature selection
18  echo ""
19  echo "@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
20  echo "Phase #1 - Performing feature extraction and selection...."
21  ~/00_Scripts/feature_extraction_selection.py -i "$syscall_dataset_dir"
    ↪  -o "$EXP_RUN_DIR" -n "$fes_ngram" -t "$fes_topn"
22
23  #2. Perform classification
24  echo ""
25  echo ""
26  echo "@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
27  echo "Phase #2 - Performing classification...."
28  ~/00_Scripts/malware_classification.py -i "$EXP_RUN_DIR" -o
    ↪  "$EXP_RUN_DIR" -d "$mc_tree_depth"
29
30  #3. Perform malware signature generation
31  echo ""
32  echo ""
33  echo "@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
34  echo "Phase #3 - Performing malware signature generation...."
35  ~/00_Scripts/signature_generator.py -i "$EXP_RUN_DIR" -o "$EXP_RUN_DIR"
36
37  #4. Perform malware detection
38  echo ""
39  echo ""
40  echo "@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
41  echo "Phase #4 - Performing malware detection...."
42  ~/00_Scripts/malware_detection.py -i "$syscall_dataset_dir" -o
    ↪  "$EXP_RUN_DIR" -s "$EXP_RUN_DIR"
43
44  duration=$(echo "$(date +%s.%N) - $start" | bc)
45  execution_time=$(printf "%.2f seconds" $duration)
46  echo ""
47  echo "Run Num: $run_num | Execution Time: $execution_time"
```

```
48 done
```

Appendix C

Experimental Runs

Run #	N-Gram Size	Top N	Tree Depth	Benign Precision	Benign Recall	Benign F1-Score	Benign Samples	Malware Precision	Malware Recall	Malware F1-Score	Malware Samples	Accuracy	Macro Precision	Macro Recall	Macro F1-Score	Weighted Precision	Weighted Recall	Weighted F1-Score
1	1	5	2	0.9	0.9844	0.9403	64	0.9846	0.9014	0.9412	71	0.9407	0.9423	0.9429	0.9407	0.9445	0.9407	0.9408
2	1	10	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
3	1	20	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
4	1	50	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
5	1	100	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
6	1	5	3	0.9545	0.9844	0.9692	64	0.9855	0.9577	0.9714	71	0.9704	0.97	0.9711	0.9703	0.9708	0.9704	0.9704
7	1	10	3	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
8	1	20	3	0.9403	0.9844	0.9618	64	0.9853	0.9437	0.964	71	0.963	0.9628	0.964	0.9629	0.964	0.963	0.963
9	1	50	3	0.9403	0.9844	0.9618	64	0.9853	0.9437	0.964	71	0.963	0.9628	0.964	0.9629	0.964	0.963	0.963
10	1	100	3	0.9403	0.9844	0.9618	64	0.9853	0.9437	0.964	71	0.963	0.9628	0.964	0.9629	0.964	0.963	0.963
11	1	5	4	0.9545	0.9844	0.9692	64	0.9855	0.9577	0.9714	71	0.9704	0.97	0.9711	0.9703	0.9708	0.9704	0.9704
12	1	10	4	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
13	1	20	4	0.9403	0.9844	0.9618	64	0.9853	0.9437	0.964	71	0.963	0.9628	0.964	0.9629	0.964	0.963	0.963
14	1	50	4	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
15	1	100	4	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
16	1	5	5	0.9545	0.9844	0.9692	64	0.9855	0.9577	0.9714	71	0.9704	0.97	0.9711	0.9703	0.9708	0.9704	0.9704
17	1	10	5	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
18	1	20	5	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
19	1	50	5	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
20	1	100	5	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
21	1	5	6	0.9545	0.9844	0.9692	64	0.9855	0.9577	0.9714	71	0.9704	0.97	0.9711	0.9703	0.9708	0.9704	0.9704
22	1	10	6	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
23	1	20	6	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
24	1	50	6	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
25	1	100	6	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
26	1	5	7	0.9545	0.9844	0.9692	64	0.9855	0.9577	0.9714	71	0.9704	0.97	0.9711	0.9703	0.9708	0.9704	0.9704
27	1	10	7	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
28	1	20	7	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
29	1	50	7	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
30	1	100	7	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
31	1	5	8	0.9545	0.9844	0.9692	64	0.9855	0.9577	0.9714	71	0.9704	0.97	0.9711	0.9703	0.9708	0.9704	0.9704
32	1	10	8	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
33	1	20	8	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
34	1	50	8	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
35	1	100	8	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704

Table C.1: Classification Results: Runs 1 - 35

Run #	Num of Signatures	TPR Count	TPR %	TN Count	TNR %	FP Count	FPR %	FN Count	FNR %
1	3	207	86.97	207	97.64	5	2.36	31	13.03
2	3	231	97.06	204	96.23	8	3.77	7	2.94
3	2	232	97.48	205	96.7	7	3.3	6	2.52
4	3	233	97.9	204	96.23	8	3.77	5	2.1
5	3	233	97.9	204	96.23	8	3.77	5	2.1
6	5	221	92.86	205	96.7	7	3.3	17	7.14
7	4	231	97.06	204	96.23	8	3.77	7	2.94
8	3	228	95.8	210	99.06	2	0.94	10	4.2
9	3	229	96.22	210	99.06	2	0.94	9	3.78
10	3	229	96.22	210	99.06	2	0.94	9	3.78
11	7	221	92.86	205	96.7	7	3.3	17	7.14
12	5	229	96.22	205	96.7	7	3.3	9	3.78
13	4	228	95.8	210	99.06	2	0.94	10	4.2
14	3	228	95.8	212	100	0	0	10	4.2
15	3	228	95.8	212	100	0	0	10	4.2
16	8	220	92.44	206	97.17	6	2.83	18	7.56
17	6	229	96.22	208	98.11	4	1.89	9	3.78
18	4	228	95.8	212	100	0	0	10	4.2
19	4	229	96.22	212	100	0	0	9	3.78
20	4	229	96.22	212	100	0	0	9	3.78
21	8	220	92.44	206	97.17	6	2.83	18	7.56
22	6	223	93.7	210	99.06	2	0.94	15	6.3
23	4	228	95.8	212	100	0	0	10	4.2
24	4	229	96.22	212	100	0	0	9	3.78
25	4	229	96.22	212	100	0	0	9	3.78
26	8	220	92.44	206	97.17	6	2.83	18	7.56
27	6	229	96.22	210	99.06	2	0.94	9	3.78
28	4	228	95.8	212	100	0	0	10	4.2
29	4	229	96.22	212	100	0	0	9	3.78
30	4	229	96.22	212	100	0	0	9	3.78
31	8	220	92.44	206	97.17	6	2.83	18	7.56
32	6	229	96.22	210	99.06	2	0.94	9	3.78
33	4	228	95.8	212	100	0	0	10	4.2
34	4	229	96.22	212	100	0	0	9	3.78
35	4	229	96.22	212	100	0	0	9	3.78

Table C.2: Detection Results: Runs 1 - 35

Run #	N-Gram Size	Top N	Tree Depth	Benign Precision	Benign Recall	Benign F1-Score	Benign Samples	Malware Precision	Malware Recall	Malware F1-Score	Malware Samples	Accuracy	Macro Precision	Macro Recall	Macro F1-Score	Weighted Precision	Weighted Recall	Weighted F1-Score
36	2	5	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
37	2	10	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
38	2	20	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
39	2	50	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
40	2	100	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
41	2	5	3	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
42	2	10	3	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
43	2	20	3	0.8767	1	0.9343	64	1	0.8732	0.9323	71	0.9333	0.9384	0.9366	0.9333	0.9416	0.9333	0.9333
44	2	50	3	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
45	2	100	3	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
46	2	5	4	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
47	2	10	4	0.9545	0.9844	0.9692	64	0.9855	0.9577	0.9714	71	0.9704	0.97	0.9711	0.9703	0.9708	0.9704	0.9704
48	2	20	4	0.8767	1	0.9343	64	1	0.8732	0.9323	71	0.9333	0.9384	0.9366	0.9333	0.9416	0.9333	0.9333
49	2	50	4	0.8767	1	0.9343	64	1	0.8732	0.9323	71	0.9333	0.9384	0.9366	0.9333	0.9416	0.9333	0.9333
50	2	100	4	0.9014	1	0.9481	64	1	0.9014	0.9481	71	0.9481	0.9507	0.9507	0.9481	0.9533	0.9481	0.9481
51	2	5	5	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
52	2	10	5	0.9552	1	0.9771	64	1	0.9577	0.9784	71	0.9778	0.9776	0.9789	0.9778	0.9788	0.9778	0.9778
53	2	20	5	0.8767	1	0.9343	64	1	0.8732	0.9323	71	0.9333	0.9384	0.9366	0.9333	0.9416	0.9333	0.9333
54	2	50	5	0.9275	1	0.9624	64	1	0.9296	0.9635	71	0.963	0.9638	0.9648	0.963	0.9656	0.963	0.963
55	2	100	5	0.9275	1	0.9624	64	1	0.9296	0.9635	71	0.963	0.9638	0.9648	0.963	0.9656	0.963	0.963
56	2	5	6	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
57	2	10	6	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
58	2	20	6	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
59	2	50	6	0.9275	1	0.9624	64	1	0.9296	0.9635	71	0.963	0.9638	0.9648	0.963	0.9656	0.963	0.963
60	2	100	6	0.9275	1	0.9624	64	1	0.9296	0.9635	71	0.963	0.9638	0.9648	0.963	0.9656	0.963	0.963
61	2	5	7	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
62	2	10	7	0.9552	1	0.9771	64	1	0.9577	0.9784	71	0.9778	0.9776	0.9789	0.9778	0.9788	0.9778	0.9778
63	2	20	7	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
64	2	50	7	0.9275	1	0.9624	64	1	0.9296	0.9635	71	0.963	0.9638	0.9648	0.963	0.9656	0.963	0.963
65	2	100	7	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
66	2	5	8	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
67	2	10	8	0.9552	1	0.9771	64	1	0.9577	0.9784	71	0.9778	0.9776	0.9789	0.9778	0.9788	0.9778	0.9778
68	2	20	8	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
69	2	50	8	0.9275	1	0.9624	64	1	0.9296	0.9635	71	0.963	0.9638	0.9648	0.963	0.9656	0.963	0.963
70	2	100	8	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704

Table C.3: Classification Results: Runs 36 - 70

Run #	Num of Signatures	TPR Count	TPR %	TN Count	TNR %	FP Count	FPR %	FN Count	FNR %
36	3	231	97.06	204	96.23	8	3.77	7	2.94
37	3	231	97.06	204	96.23	8	3.77	7	2.94
38	3	233	97.9	204	96.23	8	3.77	5	2.1
39	2	231	97.06	204	96.23	8	3.77	7	2.94
40	2	231	97.06	204	96.23	8	3.77	7	2.94
41	4	231	97.06	204	96.23	8	3.77	7	2.94
42	4	231	97.06	204	96.23	8	3.77	7	2.94
43	3	219	92.02	212	100	0	0	19	7.98
44	3	231	97.06	204	96.23	8	3.77	7	2.94
45	3	233	97.9	204	96.23	8	3.77	5	2.1
46	6	231	97.06	204	96.23	8	3.77	7	2.94
47	5	229	96.22	206	97.17	6	2.83	9	3.78
48	4	221	92.86	212	100	0	0	17	7.14
49	3	221	92.86	212	100	0	0	17	7.14
50	3	221	92.86	212	100	0	0	17	7.14
51	7	231	97.06	206	97.17	6	2.83	7	2.94
52	6	230	96.64	209	98.58	3	1.42	8	3.36
53	4	221	92.86	212	100	0	0	17	7.14
54	4	228	95.8	212	100	0	0	10	4.2
55	4	228	95.8	212	100	0	0	10	4.2
56	7	231	97.06	206	97.17	6	2.83	7	2.94
57	6	231	97.06	210	99.06	2	0.94	7	2.94
58	5	230	96.64	210	99.06	2	0.94	8	3.36
59	5	229	96.22	212	100	0	0	9	3.78
60	5	229	96.22	212	100	0	0	9	3.78
61	7	231	97.06	206	97.17	6	2.83	7	2.94
62	7	230	96.64	210	99.06	2	0.94	8	3.36
63	5	230	96.64	211	99.53	1	0.47	8	3.36
64	5	229	96.22	212	100	0	0	9	3.78
65	6	231	97.06	212	100	0	0	7	2.94
66	7	231	97.06	206	97.17	6	2.83	7	2.94
67	8	230	96.64	210	99.06	2	0.94	8	3.36
68	5	230	96.64	211	99.53	1	0.47	8	3.36
69	5	229	96.22	212	100	0	0	9	3.78
70	6	231	97.06	212	100	0	0	7	2.94

Table C.4: Detection Results: Runs 36 - 70

Run #	N-Gram Size	Top N	Tree Depth	Benign Precision	Benign Recall	Benign F1-Score	Benign Samples	Malware Precision	Malware Recall	Malware F1-Score	Malware Samples	Accuracy	Macro Precision	Macro Recall	Macro F1-Score	Weighted Precision	Weighted Recall	Weighted F1-Score
71	3	5	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
72	3	10	2	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
73	3	20	2	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
74	3	50	2	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
75	3	100	2	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
76	3	5	3	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
77	3	10	3	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
78	3	20	3	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
79	3	50	3	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
80	3	100	3	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
81	3	5	4	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
82	3	10	4	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
83	3	20	4	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
84	3	50	4	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
85	3	100	4	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
86	3	5	5	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
87	3	10	5	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
88	3	20	5	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
89	3	50	5	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
90	3	100	5	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
91	3	5	6	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
92	3	10	6	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
93	3	20	6	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
94	3	50	6	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
95	3	100	6	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
96	3	5	7	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
97	3	10	7	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
98	3	20	7	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
99	3	50	7	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
100	3	100	7	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
101	3	5	8	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
102	3	10	8	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
103	3	20	8	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
104	3	50	8	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
105	3	100	8	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704

Table C.5: Classification Results: Runs 71 - 105

Run #	Num of Signatures	TPR Count	TPR %	TN Count	TNR %	FP Count	FPR %	FN Count	FNR %
71	2	230	96.64	204	96.23	8	3.77	8	3.36
72	2	233	97.9	206	97.17	6	2.83	5	2.1
73	2	233	97.9	206	97.17	6	2.83	5	2.1
74	2	233	97.9	206	97.17	6	2.83	5	2.1
75	2	233	97.9	206	97.17	6	2.83	5	2.1
76	3	230	96.64	204	96.23	8	3.77	8	3.36
77	3	233	97.9	206	97.17	6	2.83	5	2.1
78	2	230	96.64	208	98.11	4	1.89	8	3.36
79	2	230	96.64	208	98.11	4	1.89	8	3.36
80	2	230	96.64	208	98.11	4	1.89	8	3.36
81	4	230	96.64	204	96.23	8	3.77	8	3.36
82	4	233	97.9	206	97.17	6	2.83	5	2.1
83	3	229	96.22	210	99.06	2	0.94	9	3.78
84	3	229	96.22	210	99.06	2	0.94	9	3.78
85	3	229	96.22	210	99.06	2	0.94	9	3.78
86	4	230	96.64	204	96.23	8	3.77	8	3.36
87	5	233	97.9	206	97.17	6	2.83	5	2.1
88	5	231	97.06	210	99.06	2	0.94	7	2.94
89	5	231	97.06	210	99.06	2	0.94	7	2.94
90	5	231	97.06	210	99.06	2	0.94	7	2.94
91	4	230	96.64	204	96.23	8	3.77	8	3.36
92	5	233	97.9	206	97.17	6	2.83	5	2.1
93	6	231	97.06	210	99.06	2	0.94	7	2.94
94	6	231	97.06	210	99.06	2	0.94	7	2.94
95	6	231	97.06	210	99.06	2	0.94	7	2.94
96	4	230	96.64	204	96.23	8	3.77	8	3.36
97	5	233	97.9	206	97.17	6	2.83	5	2.1
98	7	231	97.06	210	99.06	2	0.94	7	2.94
99	7	231	97.06	210	99.06	2	0.94	7	2.94
100	7	231	97.06	210	99.06	2	0.94	7	2.94
101	4	230	96.64	204	96.23	8	3.77	8	3.36
102	5	233	97.9	206	97.17	6	2.83	5	2.1
103	7	231	97.06	210	99.06	2	0.94	7	2.94
104	8	231	97.06	210	99.06	2	0.94	7	2.94
105	7	229	96.22	212	100	0	0	9	3.78

Table C.6: Detection Results: Runs 71 - 105

Run #	N-Gram Size	Top N	Tree Depth	Benign Precision	Benign Recall	Benign F1-Score	Benign Samples	Malware Precision	Malware Recall	Malware F1-Score	Malware Samples	Accuracy	Macro Precision	Macro Recall	Macro F1-Score	Weighted Precision	Weighted Recall	Weighted F1-Score
106	4	5	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
107	4	10	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
108	4	20	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
109	4	50	2	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
110	4	100	2	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
111	4	5	3	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
112	4	10	3	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
113	4	20	3	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
114	4	50	3	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
115	4	100	3	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
116	4	5	4	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
117	4	10	4	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
118	4	20	4	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
119	4	50	4	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
120	4	100	4	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
121	4	5	5	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
122	4	10	5	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
123	4	20	5	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
124	4	50	5	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
125	4	100	5	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
126	4	5	6	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
127	4	10	6	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
128	4	20	6	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
129	4	50	6	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
130	4	100	6	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
131	4	5	7	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
132	4	10	7	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
133	4	20	7	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
134	4	50	7	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
135	4	100	7	0.9412	1	0.9697	64	1	0.9437	0.971	71	0.9704	0.9706	0.9718	0.9704	0.9721	0.9704	0.9704
136	4	5	8	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
137	4	10	8	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
138	4	20	8	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
139	4	50	8	0.9697	1	0.9846	64	1	0.9718	0.9857	71	0.9852	0.9848	0.9859	0.9852	0.9856	0.9852	0.9852
140	4	100	8	0.8889	1	0.9412	64	1	0.8873	0.9403	71	0.9407	0.9444	0.9437	0.9407	0.9473	0.9407	0.9407

Table C.7: Classification Results: Runs 106 - 140

Run #	Num of Signatures	TPR Count	TPR %	TN Count	TNR %	FP Count	FPR %	FN Count	FNR %
106	2	230	96.64	204	96.23	8	3.77	8	3.36
107	2	232	97.48	204	96.23	8	3.77	6	2.52
108	2	233	97.9	204	96.23	8	3.77	5	2.1
109	2	233	97.9	206	97.17	6	2.83	5	2.1
110	1	230	96.64	208	98.11	4	1.89	8	3.36
111	3	230	96.64	204	96.23	8	3.77	8	3.36
112	3	232	97.48	204	96.23	8	3.77	6	2.52
113	3	233	97.9	204	96.23	8	3.77	5	2.1
114	3	233	97.9	206	97.17	6	2.83	5	2.1
115	3	231	97.06	208	98.11	4	1.89	7	2.94
116	3	230	96.64	204	96.23	8	3.77	8	3.36
117	4	232	97.48	204	96.23	8	3.77	6	2.52
118	4	233	97.9	204	96.23	8	3.77	5	2.1
119	4	233	97.9	206	97.17	6	2.83	5	2.1
120	4	231	97.06	208	98.11	4	1.89	7	2.94
121	3	230	96.64	204	96.23	8	3.77	8	3.36
122	4	232	97.48	204	96.23	8	3.77	6	2.52
123	5	233	97.9	204	96.23	8	3.77	5	2.1
124	5	233	97.9	206	97.17	6	2.83	5	2.1
125	5	231	97.06	208	98.11	4	1.89	7	2.94
126	3	230	96.64	204	96.23	8	3.77	8	3.36
127	4	232	97.48	204	96.23	8	3.77	6	2.52
128	6	233	97.9	204	96.23	8	3.77	5	2.1
129	6	233	97.9	206	97.17	6	2.83	5	2.1
130	6	231	97.06	208	98.11	4	1.89	7	2.94
131	3	230	96.64	204	96.23	8	3.77	8	3.36
132	4	232	97.48	204	96.23	8	3.77	6	2.52
133	7	233	97.9	204	96.23	8	3.77	5	2.1
134	7	233	97.9	206	97.17	6	2.83	5	2.1
135	7	231	97.06	208	98.11	4	1.89	7	2.94
136	3	230	96.64	204	96.23	8	3.77	8	3.36
137	4	232	97.48	204	96.23	8	3.77	6	2.52
138	7	233	97.9	204	96.23	8	3.77	5	2.1
139	8	233	97.9	206	97.17	6	2.83	5	2.1
140	7	223	93.7	212	100	0	0	15	6.3

Table C.8: Detection Results: Runs 106 - 140

Run #	N-Gram Size	Top N	Tree Depth	Benign Precision	Benign Recall	Benign F1-Score	Benign Samples	Malware Precision	Malware Recall	Malware F1-Score	Malware Samples	Accuracy	Macro Precision	Macro Recall	Macro F1-Score	Weighted Precision	Weighted Recall	Weighted F1-Score
141	5	5	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
142	5	10	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
143	5	20	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
144	5	50	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
145	5	100	2	0.9403	0.9844	0.9618	64	0.9853	0.9437	0.964	71	0.963	0.9628	0.964	0.9629	0.964	0.963	0.963
146	5	5	3	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
147	5	10	3	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
148	5	20	3	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
149	5	50	3	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
150	5	100	3	0.9403	0.9844	0.9618	64	0.9853	0.9437	0.964	71	0.963	0.9628	0.964	0.9629	0.964	0.963	0.963
151	5	5	4	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
152	5	10	4	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
153	5	20	4	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
154	5	50	4	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
155	5	100	4	0.9403	0.9844	0.9618	64	0.9853	0.9437	0.964	71	0.963	0.9628	0.964	0.9629	0.964	0.963	0.963
156	5	5	5	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
157	5	10	5	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
158	5	20	5	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
159	5	50	5	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
160	5	100	5	0.9403	0.9844	0.9618	64	0.9853	0.9437	0.964	71	0.963	0.9628	0.964	0.9629	0.964	0.963	0.963
161	5	5	6	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
162	5	10	6	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
163	5	20	6	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
164	5	50	6	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
165	5	100	6	0.9403	0.9844	0.9618	64	0.9853	0.9437	0.964	71	0.963	0.9628	0.964	0.9629	0.964	0.963	0.963
166	5	5	7	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
167	5	10	7	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
168	5	20	7	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
169	5	50	7	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
170	5	100	7	0.9403	0.9844	0.9618	64	0.9853	0.9437	0.964	71	0.963	0.9628	0.964	0.9629	0.964	0.963	0.963
171	5	5	8	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
172	5	10	8	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
173	5	20	8	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
174	5	50	8	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
175	5	100	8	0.9403	0.9844	0.9618	64	0.9853	0.9437	0.964	71	0.963	0.9628	0.964	0.9629	0.964	0.963	0.963

Table C.9: Classification Results: Runs 141 - 175

Run #	Num of Signatures	TPR Count	TPR %	TN Count	TNR %	FP Count	FPR %	FN Count	FNR %
141	2	230	96.64	204	96.23	8	3.77	8	3.36
142	2	232	97.48	204	96.23	8	3.77	6	2.52
143	2	232	97.48	204	96.23	8	3.77	6	2.52
144	2	233	97.9	204	96.23	8	3.77	5	2.1
145	1	230	96.64	206	97.17	6	2.83	8	3.36
146	3	230	96.64	204	96.23	8	3.77	8	3.36
147	3	232	97.48	204	96.23	8	3.77	6	2.52
148	3	232	97.48	204	96.23	8	3.77	6	2.52
149	3	233	97.9	204	96.23	8	3.77	5	2.1
150	3	231	97.06	206	97.17	6	2.83	7	2.94
151	3	230	96.64	204	96.23	8	3.77	8	3.36
152	4	232	97.48	204	96.23	8	3.77	6	2.52
153	4	232	97.48	204	96.23	8	3.77	6	2.52
154	4	233	97.9	204	96.23	8	3.77	5	2.1
155	4	231	97.06	206	97.17	6	2.83	7	2.94
156	3	230	96.64	204	96.23	8	3.77	8	3.36
157	4	232	97.48	204	96.23	8	3.77	6	2.52
158	5	232	97.48	204	96.23	8	3.77	6	2.52
159	5	233	97.9	204	96.23	8	3.77	5	2.1
160	5	231	97.06	206	97.17	6	2.83	7	2.94
161	3	230	96.64	204	96.23	8	3.77	8	3.36
162	4	232	97.48	204	96.23	8	3.77	6	2.52
163	5	232	97.48	204	96.23	8	3.77	6	2.52
164	6	233	97.9	204	96.23	8	3.77	5	2.1
165	6	231	97.06	206	97.17	6	2.83	7	2.94
166	3	230	96.64	204	96.23	8	3.77	8	3.36
167	4	232	97.48	204	96.23	8	3.77	6	2.52
168	5	232	97.48	204	96.23	8	3.77	6	2.52
169	6	233	97.9	204	96.23	8	3.77	5	2.1
170	7	231	97.06	206	97.17	6	2.83	7	2.94
171	3	230	96.64	204	96.23	8	3.77	8	3.36
172	4	232	97.48	204	96.23	8	3.77	6	2.52
173	5	232	97.48	204	96.23	8	3.77	6	2.52
174	6	233	97.9	204	96.23	8	3.77	5	2.1
175	8	231	97.06	206	97.17	6	2.83	7	2.94

Table C.10: Detection Results: Runs 141 - 175

Run #	N-Gram Size	Top N	Tree Depth	Benign Precision	Benign Recall	Benign F1-Score	Benign Samples	Malware Precision	Malware Recall	Malware F1-Score	Malware Samples	Accuracy	Macro Precision	Macro Recall	Macro F1-Score	Weighted Precision	Weighted Recall	Weighted F1-Score
176	6	5	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
177	6	10	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
178	6	20	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
179	6	50	2	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
180	6	100	2	0.9403	0.9844	0.9618	64	0.9853	0.9437	0.964	71	0.963	0.9628	0.964	0.9629	0.964	0.963	0.963
181	6	5	3	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
182	6	10	3	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
183	6	20	3	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
184	6	50	3	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
185	6	100	3	0.9403	0.9844	0.9618	64	0.9853	0.9437	0.964	71	0.963	0.9628	0.964	0.9629	0.964	0.963	0.963
186	6	5	4	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
187	6	10	4	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
188	6	20	4	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
189	6	50	4	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
190	6	100	4	0.9403	0.9844	0.9618	64	0.9853	0.9437	0.964	71	0.963	0.9628	0.964	0.9629	0.964	0.963	0.963
191	6	5	5	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
192	6	10	5	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
193	6	20	5	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
194	6	50	5	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
195	6	100	5	0.9403	0.9844	0.9618	64	0.9853	0.9437	0.964	71	0.963	0.9628	0.964	0.9629	0.964	0.963	0.963
196	6	5	6	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
197	6	10	6	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
198	6	20	6	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
199	6	50	6	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
200	6	100	6	0.9403	0.9844	0.9618	64	0.9853	0.9437	0.964	71	0.963	0.9628	0.964	0.9629	0.964	0.963	0.963
201	6	5	7	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
202	6	10	7	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
203	6	20	7	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
204	6	50	7	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
205	6	100	7	0.9403	0.9844	0.9618	64	0.9853	0.9437	0.964	71	0.963	0.9628	0.964	0.9629	0.964	0.963	0.963
206	6	5	8	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
207	6	10	8	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
208	6	20	8	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
209	6	50	8	0.9692	0.9844	0.9767	64	0.9857	0.9718	0.9787	71	0.9778	0.9775	0.9781	0.9777	0.9779	0.9778	0.9778
210	6	100	8	0.9403	0.9844	0.9618	64	0.9853	0.9437	0.964	71	0.963	0.9628	0.964	0.9629	0.964	0.963	0.963

Table C.11: Classification Results: Runs 176 - 210

Run #	Num of Signatures	TPR Count	TPR %	TN Count	TNR %	FP Count	FPR %	FN Count	FNR %
176	2	232	97.48	204	96.23	8	3.77	6	2.52
177	2	232	97.48	204	96.23	8	3.77	6	2.52
178	2	232	97.48	204	96.23	8	3.77	6	2.52
179	2	233	97.9	204	96.23	8	3.77	5	2.1
180	1	231	97.06	206	97.17	6	2.83	7	2.94
181	3	232	97.48	204	96.23	8	3.77	6	2.52
182	3	232	97.48	204	96.23	8	3.77	6	2.52
183	3	232	97.48	204	96.23	8	3.77	6	2.52
184	3	233	97.9	204	96.23	8	3.77	5	2.1
185	2	231	97.06	206	97.17	6	2.83	7	2.94
186	4	232	97.48	204	96.23	8	3.77	6	2.52
187	4	232	97.48	204	96.23	8	3.77	6	2.52
188	4	232	97.48	204	96.23	8	3.77	6	2.52
189	4	233	97.9	204	96.23	8	3.77	5	2.1
190	3	231	97.06	206	97.17	6	2.83	7	2.94
191	4	232	97.48	204	96.23	8	3.77	6	2.52
192	5	232	97.48	204	96.23	8	3.77	6	2.52
193	5	232	97.48	204	96.23	8	3.77	6	2.52
194	5	233	97.9	204	96.23	8	3.77	5	2.1
195	4	231	97.06	206	97.17	6	2.83	7	2.94
196	4	232	97.48	204	96.23	8	3.77	6	2.52
197	5	232	97.48	204	96.23	8	3.77	6	2.52
198	5	232	97.48	204	96.23	8	3.77	6	2.52
199	5	233	97.9	204	96.23	8	3.77	5	2.1
200	5	231	97.06	206	97.17	6	2.83	7	2.94
201	4	232	97.48	204	96.23	8	3.77	6	2.52
202	5	232	97.48	204	96.23	8	3.77	6	2.52
203	5	232	97.48	204	96.23	8	3.77	6	2.52
204	5	233	97.9	204	96.23	8	3.77	5	2.1
205	6	231	97.06	206	97.17	6	2.83	7	2.94
206	4	232	97.48	204	96.23	8	3.77	6	2.52
207	5	232	97.48	204	96.23	8	3.77	6	2.52
208	5	232	97.48	204	96.23	8	3.77	6	2.52
209	5	233	97.9	204	96.23	8	3.77	5	2.1
210	7	231	97.06	206	97.17	6	2.83	7	2.94

Table C.12: Detection Results: Runs 176 - 210