Cardiff University

A thesis submitted in partial fulfilment of the requirement for the degree of

Doctor of Philosophy

Racing Demons: Malware Detection in Early Execution

by

Matilda Rhode

August, 2021

Abstract

Racing Demons: Malware Detection in Early Execution

Malicious software (malware) causes increasingly devastating social and financial losses each year. As such, academic and commercial research has been directed towards automatically sorting malicious software from benign software. Machine learning (ML) has been widely proposed to address this challenge in an attempt to move away from the time consuming practice of hand-writing detection rules. Building on the promising results of previous ML malware detection research, this thesis focuses on the use of dynamic behavioural data captured from malware activity, arguing that dynamic models are more robust to attacker evasion techniques than code-based detection methods.

This thesis seeks to address some of the open problems that security practitioners may face in adopting dynamic behavioural automatic malware detection. First, the reliability in performance of different data sources and algorithms when translating laboratory results into real-world use; this has not been analysed in previous dynamic detection literature. After highlighting that the best-performing data and algorithm in the laboratory may not be the best-performing in the real world, the thesis turns to one of the main criticisms of dynamic data: the time taken to collect it. In previous research, dynamic detection is often conducted for several minutes per sample, making it incompatible with the speed of code-based detection. This thesis presents the first model of early-stage malware prediction using just a few seconds of collected data. Finally, building on early-stage detection in an isolated environment, real-time detection on a live machine in use is simulated. Real-time detection further reduces the computational costs of dynamic analysis. This thesis further presents the first results of the damage prevention using automated malware detection and process killing during normal machine use.

Acknowledgements

First and foremost I would like to thank my supervisor Professor Pete Burnap for his support, enthusiasm and critique of this work. As well as supporting the research itself I must thank Pete for supporting me as a researcher in allowing the freedom to both pursue and discard lines of enquiry, for giving me the encouragement and support to present, to collaborate with others, and to pursue a variety of opportunities from summer schools to teaching to hackathons, all of which have been chances to grow.

Thank you to my second supervisor Professor Omer Rana, for his thoughtful comments and for encouraging me to apply for this PhD in the first instance.

Thank you to Airbus and the Engineering and Physical Sciences Research Council for funding this work.

I am indebted to many colleagues at Airbus for their practical knowledge and research expertise. I would like to thank Professor Kevin Jones and Adam Wedgbury for creating a window into the professional cyber security space and for their time given to reading paper drafts and providing invaluable feedback. The Cyber Innovation Team at Airbus have been a sounding board over the past few years. In particular I would like to thank Dr Richard French for pushing me to think beyond the applicability of exiting algorithms and for the many diagrams that accompanied our conversations. And latterly Éireann Leverett for illustrating how to keep the wider picture in view and for some very useful reference books.

My fellow PhD students provided both practical and emotional support, in particular the Cyber Analytics Demo team: Irene, Adi and Amir who could be relied on for challenging discussions and for sympathy concerning data collection.

I am grateful to all of the reviewers and conference attendees who have provided suggestions and new perspectives to consider.

Finally I would like to thank my family. My parents, sister and husband have all sacrificed time in order to enable the writing of this this thesis and have been reliable proof-readers. Thank you all for your patience and support. Thank you to Ivor and Bluto for keeping up morale.

Abbreviations

API	Application programming interface
APT	Advanced persistent yhreat
AV	Antivirus (engine)
CPU	Central processing unit
DLL	Dynamic link library
DQN	Deep Q-Network
DT	Decision Tree
GBM	Gradient boosted machine
GPU	Graphics processing unit
HPC	Hardware performance counters
IDS	Intrusion detection system
LSTM	Long-Short Term Memroy Networks
ML	Machine learning
MLP	Multi-layer perceptron
NN	Neural Network
OS	Operating system
PCAP	Packet capture
RF	Random Forest
RL	Reinforcment learning
RNN	Recurrent neural network
SVM	Support vector machine

VM	Virtual machine

In memory of my grandmother, Willow, and her

password recovery system

Contents

Abstract **ii**

Acknowledgement **iv**

Abbreviations **vi**

Contents **ix**

List of Publications **xv**

List of Publications **xv**

List of Figures **xvi**

List of Tables **xx**

1 Introduction **1**

1.1 The Need for the Research 1

1.2 Automatic Malware Detection 2

1.3 Dynamic Malware Detection 3

1.4 Gaps in the Knowledge 4

1.4.1 Robustness 4

1.4.2 Early Stage Dynamic Detection 5

1.4.3 Early Stage Real-time Detection . . . 6
1.4.4 Process Killing for Damage-prevention . . . 6
1.5 Research Questions . . . 7
1.6 Contributions . . . 8
1.7 Thesis Structure . . . 9

2 Background: Literature Review and Research Methodologies 11
2.1 Automatic Malware Detection Skeleton . . . 13
2.2 Malware Types and Ease of Detection . . . 14
2.3 Data Types . . . 16
2.3.1 Static Data . . . 17
2.3.2 Dynamic Data . . . 22
2.3.3 Hybrid Approaches . . . 31
2.3.4 Empirical Comparisons . . . 32
2.3.5 Focus on Dynamic Data . . . 37
2.4 Detection Rules . . . 38
2.4.1 Human-generated Rules . . . 38
2.4.2 Automatically-generated Rules . . . 39
2.4.3 Types of Machine Learning Problem . . . 40
2.4.4 Machine Learning for Malware Detection . . . 42
2.4.5 Algorithm Choice in Recent Work . . . 44
2.5 Practitioner Choice . . . 49
2.5.1 Evaluation Metrics . . . 51
2.6 Model Robustness . . . 52
2.7 Detection Time . . . 54
2.8 Real-Time Malware Detection . . . 56
2.8.1 Latency and power consumption . . . 57
2.8.2 Signal separation . . . 58

2.8.3 Automated Response: Process Killing 59
2.9 Thesis Scope 60
2.10 Conclusions 61

3 Robust Models 63
3.1 Motivation and Definitions 64
3.1.1 Lack of Benchmark Datasets 64
3.1.2 Robustness 65
3.1.3 Contextual Differences 66
3.1.4 Concept Drift 67
3.1.5 Adversarial Samples 68
3.1.6 Malware Detection Robustness - Focus on Contextual Difference 69
3.2 Methodology 69
3.2.1 Feature Selection 70
3.2.2 Feature Representation 71
3.3 Choosing Data 71
3.3.1 Features 71
3.3.2 API calls 72
3.3.3 Network data 73
3.3.4 Machine Metrics 74
3.3.5 Summary of feature sets and representations 74
3.4 Algorithms 75
3.4.1 Neural network 76
3.5 A Machine Learning Model from Laboratory to Practice 78
3.5.1 Samples 78
3.5.2 Sparsity 80
3.5.3 Data Collection 81
3.5.4 A Note on Time Series Data with Non-Time Series Algorithms . . 82

3.6 Contextual Robustness Results 82
3.6.1 EXE 84
3.6.2 PDF 86
3.6.3 DOC 88
3.6.4 XLS 89
3.6.5 Discussion: EXE Results 90
3.7 Limitations and Conclusions 91

4 Early Stage Prediction 93
4.1 The Case for Early Stage Prediction 93
4.1.1 Dynamic Malware Detection: Time Series Algorithms and Time-saving Approaches 95
4.2 Methodology: Recurrent Neural Networks for Early Stage Malware Prediction 97
4.2.1 Features 97
4.2.2 Algorithm Choice 98
4.2.3 Data Collection 101
4.2.4 Samples 101
4.2.5 Data Distributions 103
4.2.6 Data Preprocessing 104
4.3 Experimental Results 106
4.3.1 Hyperparameter Configuration 106
4.3.2 Predicting Malware Using Early-Stage Data 108
4.3.3 Simulation of Zero-day Malware Detection 111
4.3.4 Ransomware Case Study 117
4.3.5 Improving Prediction Accuracy With an Ensemble Classifier . . . 120
4.3.6 Feature Importance 123
4.4 Limitations and Future work 125

4.4.1 Robustness to Adversarial Samples 125
4.4.2 Process Blocking 126
4.4.3 Portability to Other Machines and Operating Systems 126
4.5 Conclusions 127

5 Real-time Malware Detection and Process Killing 130
5.1 The case for real-time detection with an automated response 131
5.2 Research Challenges 134
5.3 Methodology: Supervised Learning to Automated Agent 136
5.3.1 Statistical Approach: Alert Filtering 138
5.3.2 Reinforcement Learning: Q-learning with Deep Q Networks . . . 138
5.3.3 Regression using Q-Values 141
5.4 Evaluation Methodology: Ransomware detection 143
5.5 Experimental design 143
5.5.1 Features 144
5.5.2 Data Capture 145
5.6 Experimental Results 149
5.6.1 Dataset 149
5.6.2 Hyperparameter Search for Process-Level Supervised RNN . . 153
5.6.3 Supervised Learning for Process Killing 156
5.6.4 Accuracy vs. Resource consumption 160
5.6.5 How to solve a problem like process killing? 164
5.6.6 Further experiment: Favouring high TNR 168
5.7 Measuring damage prevention in real time 169
5.8 Discussion: Measuring execution time in a live environment 173
5.9 Future Work 175
5.10 Conclusions 176

6 Conclusions, Limitations, Future Work 179
6.1 Thesis summary 179
6.2 Limitations 182
6.2.1 Dataset labelling 182
6.2.2 Ransomware as a Case Study 183
6.2.3 Machine Metrics 184
6.3 Future Work 184
6.3.1 Other file types and operating systems 184
6.3.2 Robustness 184
6.3.3 Earlier detection 185
6.3.4 Improving Real-time Detection Accuracy 185
6.3.5 When Does Harm Begin? 186
6.4 Concluding remarks 186

References 220

List of Publications

This thesis contains work first published in the following:

Rhode, M., Burnap, P. and Jones, K., 2018. Early-stage malware prediction using recurrent neural networks. In *Computers & Security*, 77, pp.578-594.

Rhode, M., Tuson, L., Burnap, P. and Jones, K., 2019, June. Lab to soc: Robust features for dynamic malware detection. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks–Industry Track* (pp. 13-16). IEEE.

[Preprint] Rhode, M., Burnap, P. and Jones, K., 2019. Real-time malware process detection and automated process killing. *arXiv preprint* arXiv:1902.02598.

Other works published during this thesis:

Anthi, E., Williams, L., Rhode, M., Burnap, P. and Wedgbury, A., 2021. Adversarial attacks on machine learning cybersecurity defences in industrial control systems. *Journal of Information Security and Applications*, 58, p.102717.

Marques, P., Rhode, M. and Gashi, I., 2021. Waste not: using diverse neural networks from hyperparameter search for improved malware detection. In *Computers & Security*, p.102339.

List of Figures

2.1 Minimum essential elements of an automatic malware detection engine 13

2.2 Structure of Windows Portable Executable (PE) files 19

2.3 Figure of trending malware samples reproduced with permission from PolySwarm [151] e-Newsletter 15th April 2021 showing thousands of samples of the same malware families observed in a single week. . . . 30

2.4 Number of samples used in 92 papers (Only 92 of 97 report number of software samples) coded by use of self-collected dynamic data. *N.B. The penultimate two papers use the same dataset.* 35

2.5 Approximate number of search matches per annum reported by Google Scholar [118] in academic texts between 2009 and 2019 for 'malware detection' only 'malware detection' and ('machine learning' or 'neural network' or neural networks' or 'deep learning') 'malware detection' and 'machine learning' 'malware detection' and ('neural network' or neural networks') 'malware detection' and 'deep learning' 42

2.6 Algorithms used by 97 recent malware detection papers. NN=neural network, SVM = support vector machine, RF = random forest, DT = decision tree, KNN = K-nearest neighbours, LR = logistic regression, HMM = Hidden Markov Model, Human = human-generated rules. The ensemble algorithms may use a number of techniques to combine the predictions of different models. 45

2.7 Total papers using neural networks (NN), support vector machines (SVM) and random forests (RF) in 97 recent malware detection papers by year 46

2.8 Most popular algorithms winning Kaggle machine learning competitions to 2016 [70] 49

2.9 Most commonly uploaded filetypes to VirusTotal in the 7 days preceding 21st August 2019 [203] 61

3.1 Representation of a machine learning malware detection classifier pipeline with intermediate steps between collecting raw data and parsing by a rule set to obtain a final decision. Feature selection excludes features that do not meet a certain criteria whereas feature representation describes the data structure used by the rule set. 70

3.2 Percentage of instances with non-zero network data, machine metrics and API calls by filetypes in the public dataset. Grey dashed line denotes 50% mark, the threshold for sparsity. 80

3.3 Mean (μ) F1-scores for withheld public test set, organisation test set, absolute difference between mean F1-scores, mean absolute difference between F1-scores per model, mean absolute difference in true positive rate (TPR) and true negative rate (TNR) per model. Evaluated using a public and commercial dataset for 4 filetypes, 3 algorithms, 6 input feature sets averaged over 10 models (10-fold cross-validation). **N.B.** Metrics have been multiplied by 100 for easier comparison with percentages 83

3.4 F1-score, TPR, FNR on public test sets and organisation test set for EXE files with different feature sets and algorithms 84

3.5 Mean μ Executable classification F1-scores for public and organisation test set as well as difference metrics for additional different feature sets. 87

3.6 F1-score, TPR, FNR on public test sets and organisation test set for PDF files with different feature sets and algorithms 88

3.7 F1-score, TPR, FNR on public test sets and organisation test set for DOC files with different feature sets and algorithms 88

3.8 F1-score, TPR, FNR on public test sets and organisation test set for XLS files with different feature sets and algorithms 89

4.1 High-level model overview 98

4.2 Box plots of input features for benign and malicious samples 105

4.3 Classification accuracy (percentage with correct labels) for various machine learning algorithms and a recurrent neural network as time into file execution increases 108

4.4 Features over time for each malware family and benignware 114

4.5 Comparative detection accuracy on various malware families with examples of the family omitted from the training set 115

4.6 Comparative detection accuracy on various malware variants with examples of the variant omitted from the training set 116

4.7 Classification accuracy on ransomware for one model which has not been trained on ransomware (omitted), and for one which has (half included) 118

4.8 Impact scores for features with 1, 2 and 3 features turned off 4 seconds into file execution 124

5.1 The damage caused over time by two imaginary malware, one which contacts a server before inflicting damage (left) and one which does so afterwards (right). 133

5.2 High-level depiction of three approaches taken 137

5.3 Benignware sample, normalised process-level metrics 6 observations made without process being killed 148

5.4 Malware sample, no observations made yet 148

5.5 Validation set F1-score variance with hyperparameter values and feature sets . . . 155
5.6 F1 scores, true positive rates (TPR) and true negative rates (TNR) for partial-trace detection (process measurements), full-trace detection (whole process), whole application (process tree) and with process level measurements + process killing (process killing) for validation set (left column) and test set (right column) . . . 157
5.7 3 levels of data collection: Each measurement, each process, each process tree . . . 159
5.8 CPU use of different trained models . . . 162
5.9 RAM use of different trained models . . . 162
5.10 Inference duration of different trained models . . . 162
5.11 Total number of files corrupted by ransomware with no process killing and with three process killing models within the first 30 seconds of execution. 171
5.12 Mean time to collect data, analyse data with Random Forest, and kill varying numbers of processes . . . 174

List of Tables

2.1 Dynamic data sources used in 64 malware detection and classification papers. *N.B. Percentages sum to more than 100 as some research uses more than one data source.* 26

2.2 Reported accuracies, data types and test samples ratio for 5 papers comparing of static, dynamic and hybrid data. *=Area Under the Curve of Receiver Operating Characteristic multiplied by 100 reported instead of accuracy. Metric notes column comments on the metrics used to evaluate the classification, especially when using unbalanced dataset. None of these papers reports F1 but all except [47] report FNR (false negative rate) and FPR (false positive rate) or inverse metrics (true rates). 33

2.3 Dataset size statistics for 92 papers (Only 92 of 97 report number of software samples) using dynamic data (includes hybrid approaches), static data only, and those papers using dynamic data that was pre-collected prior to the research paper being presented. 36

2.4 F-Scores vs. percentage with correct labels (accuracy) for three sets of predictions on a dataset of 4 samples (2 positive, 2 negative) 52

2.5 Reported data sample sizes and times collecting dynamic behavioural data per sample 54

2.6 Real-time malware detection literature problems considered. OS = operating system; HPCs = Hardware performance counters; DT=Decision Tree; MLP = multi-layer perceptron; NN = Neural Network; RF=Random Forest 57

3.1 References using API calls as data, representation method and additional data collected. Where papers have used multiple representations (e.g. [61]) the best-performing representation is recorded) 72

3.2 Dataset provenance by filetype. Org. = Commercial data provided by the organisation. Benign publicly available samples validated by checking for detections using VirusTotal [202] 79

4.1 Possible hyperparameter values and the hyperparameters of the best-performing configuration on the training set 101

4.2 Number of instances of different malware families in dataset 103

4.3 Minimum and maximum values of each input feature for benign and malicious samples 104

4.4 Highest average accuracy (percentage with correct labels) over 10-fold cross validation on training set during first 20 seconds of execution with corresponding false positive rate (FP) and false negative rate (FN) . . . 109

4.5 Highest accuracy (percentage with correct labels) on unseen test set during first 20 seconds of execution with corresponding false positive rate (FP) and false negative rate (FN) 109

4.6 RNN prediction Accuracy (Acc.) (percentage with correct labels), false negative rate (FN) and false positive rate (FP) on test set from 1 to 20 seconds into file execution time 111

4.7 Number of samples belonging to different families and variants with over 100 samples in the dataset 112

4.8 Classification accuracy on different malware families with all instances of that family removed from training set 113
4.9 Classification accuracy on different malware variants with all instances of that variant removed from training set 113
4.10 Classification accuracy (percentage with correct labels) on ransomware for one model which has not been trained on ransomware (omitted), and for one which has (half included) 118
4.11 Maximum accuracy scores in predicting ransomware with only one and two features turned on for a model not trained on ransomware and for a model trained on ransomware 119
4.12 Highest accuracy-scoring configurations during first 5 seconds in 10-fold cross validation on training set 121
4.13 Ensemble accuracy (acc.), false positive rate (FP) and false negative rate (FN) compared with highest accuracy of configurations A, B and C. Those marked with a ‘*’ signify predictions that were statistically significantly more confident by at the confidence level of 0.01 121

5.1 26 process-level features: 22 features + 4 port status values 144
5.2 Malware families with more than 10 samples in the dataset. 315 families were represented in the dataset, with 27 having being represented more than 10 times. Basic description provided which does not cover the wide range of behaviours carried out by some malware families but is intended to indicate the range of behaviours in the top 27 families included in the dataset. PUA = potentially unwanted application, RAT = remote access trojan 150
5.3 Hyperparameter search space and the hyperparameters of the model giving the lowest mean false-positive and false-negative rates 154

5.4 F1-score, true positive rate (TPR) and true negative rates (TNR) (all * 100) on test and validation sets for classification and process killing . . 158
5.5 Average resource consumption over 100 iterations for a batch size of 100 vs. F1-scores on validation and test set for classification and process killing across 14 models. 26 features = process-level only, 37 features = machine and process level features. 164
5.6 Summary of best process killing models by model training methodology. F1, TNR and TPR for validation and test datasets. Full results in Appendix Table 2. 168
5.7 Two models' F1-score, TNR, TPR for the validation and test set scoring the highest TNR on the validation and test sets. 169
5.8 Random Forest and Decision Tree each with a minimum requirement of two alerts ('malicious classifications') to kill a process. F1, TNR and TPR reported on validation and test set. 170
5.9 Total number of files corrupted by ransomware with no process killing and with three process killing models within the first 30 seconds of execution. Damage reduction is the percentage of files spared when no killing is implemented 172

1 Mean (μ) F-scores for withheld public test set, organisation test set, absolute difference between mean F-scores, mean absolute difference between f-scores per model, mean absolute difference in true positive rate (TPR) and true negative rate (TNR) per model. Evaluated using a public and commercial dataset for 4 filetypes, 3 algorithms, 6 input feature sets averaged over 10 models (10-fold cross-validation). **N.B.** Scores have been multiplied by 100 for easier reading 188
2 Summary of process killing models, validation and test set score metrics [Table 1 of 3] 189

3 Summary of process killing models, validation and test set score metrics [Table 2 of 3] 190

4 Summary of process killing models, validation and test set score metrics [Table 3 of 3] 191

Chapter 1

Introduction

1.1 The Need for the Research

Opportunities for cybercrime are growing as the world becomes increasingly digitised. Malicious software (malware) in particular has been judged the most financially damaging form of cybercrime [119]. In recent years destructiveware, software which damages digital artefacts, has taken public health services and government entities hostage [55, 56, 94, 111, 198] and is an increasing concern at company board level [38].

The serious consequences of a successful malware attack have led to a well-populated antivirus (AV) and endpoint protection market [138]. Endpoint protection refers to the mitigation and prevention of damage to client devices on a network. These technologies are constantly challenged by malware authors, who are highly motivated to evade detection. As such, the prevention of malicious attacks is both a compelling and complicated problem.

One of the greatest challenges in malware detection is responding to the volume of new unseen samples that appear each day; which frequently exceeds one million on the malware analysis platform, VirusTotal [203]. Automated malware detection is essential to handle this volume of samples since humans cannot be the sentries of every digital device being used globally, there being more of the latter than of the former.

1.2 Automatic Malware Detection

Historically, AVs have used hand-written rules to detect malware by matching software against elements of known malicious code, known as signatures. Signature-based detection is still used today [27]. This approach is well-suited to stopping known malware but may be bypassed by code obfuscation techniques and is unable (92% of the time according to [84]) to detect new malicious samples. This is a significant limitation as the malware landscape evolves over time in response to new opportunities for attackers [150, 204].

Machine learning (ML), a process of deriving a rule set through iterative exposure to relevant examples, has been widely proposed as a means to automatically detect malware e.g. [64, 179, 217]. This method may replace or complement hand-written rules with the key advantage being that ML rules can be automatically generated when the malware landscape or defensive environment changes.

One limitation of ML is that it can also learn incidental patterns between malware and benignware which do not relate to the core malicious functionality. Incidental patterns are those data trends which correlate with the two classes but are not true indicators of being malicious or benign and therefore may cause issues for machine learning accuracy if the trends shift alignment with the binary classification problem. In some cases, this may not be evident if the model is performing well in detection trials. But if these incidental patterns do not align with the malware/benignware split in the general population of samples, there is likely to be a degradation in model performance known as a 'generalisation error' or 'out-of-sample error' [130]. Due to the huge volume of malware and its changeability, it is very difficult to obtain a representative sample of the entire malware ecosystem. Moreover, malware families and benignware may vary in different contexts. The types of malware seen in different industry verticals, in different technology stacks, and even different geographic locations may vary [119]. For instance, the financial industry may see a higher proportion of credential-stealing malware whilst biotech and engineering start ups may see more data exfiltration malware trying to steal intellectual property.

ML-based detection requires data in order to learn detection rules. Within the domain of malware detection, there is considerable debate as to which attributes of software should form the basis of these rules [47, 89, 217]. Dynamic data is behavioural data captured whilst a malware sample is executing and is difficult to manipulate since the attacker must leave some behavioural footprint of their activity; this is in contrast to static data which is extracted directly from code and can therefore be obfuscated using various techniques such as inserting junk code to mislead a classifier [108, 221]. Code based approaches are often preferred however due to the speed of data collection and low computational overheads [57, 100, 135, 179, 218].

This thesis investigates the use of dynamic data for automated malware detection and automated malicious process killing, addressing the slowness of dynamic data capture in particular.

1.3 Dynamic Malware Detection

Whilst dynamic malware detection using machine learning has been demonstrated to be a promising approach with high ($>$99%) detection accuracy (percentage of correct predictions) [45, 48, 87]. There are many different data sources and kinds of machine learning algorithms and a number of these have been used with success by security researchers.

Though algorithms and data vary, the use of virtual machines (VMs) for data collection is common to many dynamic detection models [3, 7, 54]. A VM is an emulated computer running on another host computer that uses software-defined resources instead of using underlying hardware resources. An *isolated* VM, also called a sandbox, is a VM that is disconnected from the communication network. Sandboxes are typically used for dynamic software analysis to protect the wider network from potentially malicious software. The use of VMs incurs both computational overheads [170] and may enable attackers to hide malicious behaviour [176]. For example, malware may wait for a specific calendar date to initiate malicious behaviour as was observed with the Shamoon malware attacking Saudi Arabian oil company, Aramco, launched during Ramadan [51] or a malware variant which only attacks computers with certain keyboards installed [132]. Thus the behaviour in the VM would not be indicative of the damage on the endpoint. The only way to ensure that the real-world behaviour is observed is to replicate the necessary requirements for damage, which requires the analyst to know what the necessary requirements are (e.g. for it to be a certain date). Therefore, the simplest way to do this is to observe behaviour on the target endpoint. This removes the possibility that monitored behaviour in the VM could deviate from the real-world behaviour. Monitoring the behaviour on the target is known as real-time detection as it must occur in real-time. Real-time detection monitors software live on the endpoint being protected, thus removing the need for an isolated machine and its associated computational costs as well as eliminating a means for malware to disguise itself. Relatively little research has been conducted in this domain likely due to the risk incurred by running malware on the endpoint that is supposed to be protected. This thesis pursues the viability of real-time detection due to the reduced computational overheads and the alignment of the data used for analysis and observed in the real world.

An analysis of the recent work in this domain has revealed the following gaps when considering how these models could be used to prevent malware attacks in practice:

1.4 Gaps in the Knowledge

1.4.1 Robustness

There is a lack of analysis into the robustness of different dynamic data sources and machine learning algorithms with respect to laboratory performance and ‘real-world’ performance. This is a problem because many research papers report high classification accuracy metrics [45, 48, 87] but if this does not hold in reality users of the detection model may have no way of knowing that malware is passing to the endpoint undetected, potentially causing serious damage and disruption.

Testing robustness is necessary but not sufficient to address the challenges to adopting dynamic analysis.

1.4.2 Early Stage Dynamic Detection

Previous work has not sought to address one of the biggest limitations to using dynamic data as part of endpoint protection - the time taken to collect this data. Code-based (static) data is sometimes preferred over dynamic data due to speed of collection [4, 61] but static data can easily be manipulated since a single program can be written many ways to achieve the same goal [220, 221]. If dynamic data can be made competitive with static data, this could reduce adversaries' ability to circumvent malware detection software. This thesis makes an implicit assumption that 5 minute analysis time is one of the reasons that static analysis is preferred over dynamic, as stated by recent literature [100, 125, 153], but also notes that much dynamic analysis work does not justify the time used for data recording e.g. [29, 135], therefore seeks to investigate whether several minutes are necessary to collect sufficient data for a model to discriminate between malware and benignware.

The dynamic malware detection paradigm is rooted in the use of VMs, which consume computational resources and may not always coax malicious behaviour from malware samples. For example, if a different operating system version is used in the virtual environment which does not have the security vulnerability that a user's machine has. Real-time detection whereby data is collected on the user's machine would address this issue, but there has been limited work in this field to date.

1.4.3 Early Stage Real-time Detection

Within the existing literature, real-time malware detection literature has not addressed the need for sufficiently fast automated defensive measures such as process blocking as a means to prevent damage to the endpoint. If malware is detected on a real endpoint, as would be the case with real-time detection, humans are unlikely to be quick enough to respond to prevent damage since this can occur in seconds [43], therefore if malware damage is to be prevented rather than simply detected the response needs to be automatic. Some work has been conducted towards process killing (see Sun et al. [188]) but has not evaluated whether it truly prevents damage. As well as using early data, an algorithm with a quick processing time, or low latency, may also help to reduce the time malware has to cause damage.

1.4.4 Process Killing for Damage-prevention

Real-time detection research has not considered damage-preventing automated defence in the context of the background noise created by normal machine use. In typical dynamic analysis, a single sample is executed in a sandbox [87, 89, 177]; but in real-world use many benign programs are also running - this introduces a signal separation problem between benignware and malware which must now be analysed at the process level rather than looking at the overall impact of a single application on a machine as can be done in a sandbox. This raises a wider philosophical question around the ability to distinguish benignware and malware when the subprocesses being used may be very similar, or even identical. Some initial research has been conducted in this sphere but only using a maximum of 5 simultaneous applications [188]. As more computational resources are consumed by a user's legitimate processes, the workload on the detection model increases, which could delay malicious process killing and therefore allow more damage to be caused. It is therefore important to trial a more realistic number of applications.

In addition to the number of concurrent processes, existing work [188] has waited one minute for programs to execute when it is known that destructiveware can impact 100,000 files in 17 seconds [43]. Whilst one may assume that the sooner a malicious process is killed, the better, cross-comparison of time across machines may not be appropriate where hardware is different. Increased processing power allow malware to cause more damage more quickly. In addition to this, the monitoring and killing system itself should have a low computational impact in order not to interfere with user activity and consume too many resources. Thus the work on process killing in this thesis measures damage partly through files destroyed and considers the computational complexity of the detection algorithms.

1.5 Research Questions

Given these gaps in previous literature identified above, this thesis seeks to address the following research questions which are raised in response:

- *RQ1: To what extent do dynamic data and algorithm choices impact the robustness of malware detection accuracy across different datasets?*
- *RQ2: Is it possible to predict that software is malicious from its early stages of execution?*
- *RQ3: Does the use of lower latency algorithms impact upon the speed of accurate malware detection?*
- *RQ4: Is it possible to separate the signals of tens of simultaneous benign and malicious processes?*
- *RQ5: Is it possible to detect and kill malicious processes early enough to prevent malicious damage?*

1.6 Contributions

The broad contribution of this thesis to the field of malware detection is to demonstrate that it is possible to detect malicious software from incomplete behavioural traces within a far shorter time scale than has previously been reported. The specific contributions are:

- *C1: A demonstration that the most accurate type of dynamic data in laboratory testing may not be the best-performing or most robust data type when tested on data of a different provenance.*

  This finding was first published in [158]

  M Rhode, L. Tuson, P. Burnap, and K. Jones. *Lab to soc: Robust features for dynamic malware detection.* In 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks–Industry Track, pages 13–16. IEEE, 2019

- *C2: The first model to predict whether or not software is malicious based on its early behaviour.*

  This was first published in [156]

  M. Rhode, P. Burnap, and K. Jones. *Early-stage malware prediction using recurrent neural networks.* Computers & Security, 77:578–594, 2018.

- *C3: Real-time malware detection against a significantly larger number of user background applications than have previously been investigated with analysis of the impact of the increase*

- *C4: The first model to try and mitigate malware damage in real-time using malicious process detection and killing*

  C3 and *C4* are related contributions and currently under review for publication; a preprint of the paper can be found here [157].

1.7 Thesis Structure

This chapter has introduced the domain, set out the research questions, and indicated the key contributions of this research. This thesis is structured as follows to establish the research context, methods used, results, discussion, key findings, conclusion and indicators for future work.

Chapter 2 - Background

This chapter aims to provide the primitives to the field of automated malware detection as well as a literature review and gap analysis of the domain and an overview of the thesis scope. Within the subsequent three chapters (3,4,5), which outline methodologies and experimental results, there is a short literature review of concepts that relate directly to the topic of that chapter.

Chapter 3 - Robust Models

This chapter explores the practical challenges to malware detection accuracy when transferring a machine learning model into industrial use. A generic malware detection ML framework is presented followed by an analysis of the most common elements used for dynamic malware detection. The experimental methodology and experimental results are detailed together with an analysis of the results and their implications for security practitioners. The experiments reveal that the most commonly used behavioural attributes for executable files do in fact have the highest accuracy in the lab but are outperformed by continuous numeric machine data when tested with real-world industrial data in support of contribution *C1*.

Chapter 4 - Early stage prediction

This chapter presents the arguments for early-stage prediction using behavioural data and reports the results of extensive experiments using different machine learning algorithms, time windows, and the ability of such models to classify previously unseen (zero-day) malware families in support of contribution *C2*.

Chapter 5 - Real-time Malware detection and process killing

This chapter explores the more difficult problem of classifying and automatically blocking malware on-the-fly on a machine in use. The experiments in this chapter simulate real machine use and try to detect and kill malware as early in execution as possible. Further results are presented on the damage prevented by this model in practice using a live trial in support of contributions *C3* and *C4*.

Chapter 6 - Limitations, Future Work, Conclusions

Finally, this chapter concludes the thesis by summarising the contributions to the field of automated malware detection, the limitations of these contributions and the questions that have arisen from this research.

Chapter 2

Background: Literature Review and Research Methodologies

Cybercrime leaves progressively larger economic and social footprints each year [123]. A 2019 report [119] from consultancy firm, Accenture, records that the average cost of cybercrime to an organisation is $13m per annum, with malware claiming the largest portion of this sum at $2.6m. This figure rises to $3.3m when ransomware, a type of malware which extorts victims, is included [119]. A 2020 report from anti-virus vendor, Sophos, [185] notes that the average cost to remediate a ransomware attack is $750K. These costs describe the immediate financial impact of regulatory fines, business impact, technological recovery and investigation and not the long-term implications of reputation damage, intellectual property loss, or economic impact to the wider supply chain.

Malware is a broad term describing any code which ‘causes harm to users, software or networks’ [180]. Historically, the rate at which new malware appeared was sufficiently low [124] that expert reverse engineers could analyse each new malicious sample and develop ad-hoc defences.

Today, the volume of new malware presents a significant challenge for security practitioners. VirusTotal [202], which offers a free platform to aggregate the verdicts of around 60 commercial anti-virus engines for a given file or website address, routinely receives more than one million new distinct files for analysis in a single day[1]. This large volume makes it difficult for small teams of experts to match the pace of analysis to the volume of new files, as each instance may take a few minutes to determine whether or not it is malicious. Even if all reverse engineers shared data globally, malware authors could respond by flooding the ecosystem with slight variants in order to make their workload impossible.

Automated detection of malware is necessary to combat the sheer volume of both known and unknown files seen by an organisation on a daily basis. Automated detection can be carried out at any time of day, it is not dependent on human availability.

An additional challenge to detecting malware, beyond the sheer volume, is the incentive for malware authors to evade detection. Approaches to automated detection vary, and some are more difficult to bypass than others.

Malware detection on the endpoint aims to detect malicious software before it is able to cause damage. In this context an endpoint refers to any user device or server, but within the scope of this thesis, typically an 'endpoint' is a Windows PC user machine since this is the most commonly used device within business contexts [138]. This chapter presents a skeleton of the minimal components required for automatic malware detection in Section 2.1 followed by a brief discussion about different types of malware in Section 2.2 and the importance and likelihood of detecting different varieties. A literature review and analysis of the data types used in previous work is outlined in Section 2.3 and of the methodologies used to devise detection rules in Section 2.4. Section 2.5 outlines the challenges that practitioners face in trying to adopt this research for malware detection. Sections 2.6, 2.7 and 2.8 discuss some of the unanswered questions relating to using current research as part of an automated defensive system and the research questions arising from these gaps. Section 2.9 describes the scope of the thesis, in particular, the key filetype and operating system considered. Finally, Sec-

[1]1,131,484 new distinct files uploaded to VirusTotal on 21st March 2020 [203]

tion 2.10 concludes this chapter, summarising the contributions of this thesis to the field of automatic malware detection research.

2.1 Automatic Malware Detection Skeleton

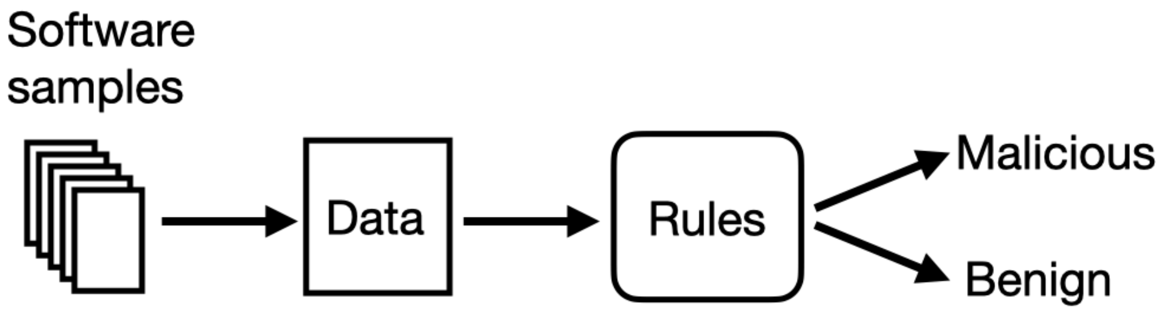


Figure 2.1: Minimum essential elements of an automatic malware detection engine

Automatic malware detection is now essential to handle the rate of previously unseen software samples requiring analysis. Figure 2.1 illustrates the minimal components required for an automatic malware detection model. Some information is collected from incoming samples, which is then filtered through a set of rules to give a verdict of ‘malicious’ or ‘benign’. Additional components may be added to this pipeline but taking just the essential elements, there are two necessary modelling decisions which can be taken from a wide set of possibilities many of which perform well, as the next sections will elaborate.

The first modelling decision is selecting which data to use. A variety of data can be collected from a software sample. The choice of data may confer advantages or limitations onto the detection model with respect to analysis time, robustness in detecting future malware variants and families, and the types of attacks that the model is more likely to detect. For example, a key debate surrounds the use of static (code-based) and dynamic (behavioural) data [47, 89], the former is quicker to obtain but more vulnerable to evasion techniques while the latter tends to take longer to collect but is more difficult to evade [179]. The behaviour of malware is closely aligned with the goals of the software and is therefore difficult to obfuscate without compromising the malicious activity itself.

The second compulsory modelling decision is selecting the mechanism for generating rules to parse the data. This choice impacts the ease with which the model can be updated, the resources required for a single analysis, and other characteristics. Anti-virus engines (AVs) have used blacklists of known malicious files, hand-written rules and, more recently, automatically learned rules. Machine learning (ML) describes the process of automatic rule induction from data. This has benefits for domains like as malware detection where the rule set is likely to need updating as new malware families come into being.

Machine learning for automatic malware detection seems promising with some work demonstrating $>$99% detection accuracy [45, 48, 87]. However, there are some open questions when one considers how these models might be adopted in practice.

2.2 Malware Types and Ease of Detection

Whilst this thesis is not concerned with one particular type of malware over another and assumes that all malware are undesirable, it is worth mentioning that there are some malware which may be easier to detect with automated detection than others and some which may be more important to detect than others.

It is important to note that malware are diverse in the damage they cause and the ease with which they can be detected. The detection of malware on an endpoint is a means to preventing damage to that endpoint, rather than an end in itself. There are various malware types and there can be different consequences for each victim of the same malware attack e.g. a difference in the value of data stolen to the victim. As well as the malware type, the security posture of the victim also influences the impact of malware, whereby those with a suitable recovery plan may suffer less than those without e.g. recently backed-up data to remediate the consequences of ransomware.

For some malware families, it is debated whether they even cause damage. There is an entire group of software that is acknowledged as occupying a grey-area between malware and benignware. These software are known as potentially unwanted applications or programs (PUAs or PUPs) which are downloaded alongside wanted software [199] and may carry out undesirable activities like tracking user data without notifying the user. PUPs are often forms of adware [199] which is malware that generates revenue from showing advertisements to users. Unlike legitimate adverts, however, it may do so without gaining appropriate user consent. The impact on the user in practice is similar to legitimate advertising software for which the user may have been required to agree to some terms and conditions in order to gain access to a particular software or platform. Ransomware, by contrast, can cause a lot of damage, as noted above, since it can lead to huge financial losses [141] and potentially loss of life when one considers the impact on healthcare services [123].

The ease of detection also varies. One of the more challenging types to detect are those malware which form part of an Advanced Persistent Threat (APT). APT attacks and APT groups, as the name implies, launch advanced and persistent attempts to access an endpoint or network and may be bespoke to a particular target [169]. As such they may share fewer behavioural or code attributes with other malware. Research into APT malware detection often uses data generated by researchers rather than using real-world examples since the real examples may require a high fidelity representation of the target victim's infrastructure [67, 79]. This limitation on data collection makes it difficult to collect a realistic dataset, which is the foundation from which ML models learn [72]. Code reuse by malware authors as well as wide propagation of malware makes it more likely that a machine learning model will infer similarity between malicious samples. Uniqueness makes malware more difficult to detect automatically, since the rules (ML- or human-generated) must distinguish malware and benignware either using broad strokes or specific attributes, such that if the malware behaves unusually or has never been seen before it is difficult to create a rule that will capture it without some kind of clairvoyance. Though some work by David and Netanyahu has been done to try and automatically generate malware signatures to this end [50].

The purpose of this discussion is to highlight that there are many different forms of malware and that detecting them may be both more or less valuable (relating to damage) and more or less difficult (relating to novelty and uniqueness). This thesis will return to the detection of ransomware since its behaviour is not unique (it has been observed in the wild many times [151]) and it is highly destructive and therefore well-suited to the task of automated detection.

Overall this thesis is concerned with detecting all varieties of malware since a new variant may appear at any time, and these often cause the most damage since there are not adequate defences in place [123]. Therefore it is important to select a data source which is capable of capturing software attributes common to existing and, hopefully, future malware. The next section will discuss data types used for malware detection in depth.

2.3 Data Types

In analysing a piece of music, one can read the score, play it at full speed or step through each note one by one. Each of these may be considered a different kind of data. Similarly, software can be read (static analysis), played (dynamic analysis) or stepped through (disassembly).

Data are the basis upon which an automatic malware detection system distinguishes between malicious and benign samples. Numerous data types can be collected from software. These types can be broadly categorised as either static or dynamic data sources. Static data are collected from code without executing a sample, dynamic data are collected whilst a sample executes. Both static and dynamic data continue to be widely used by researchers as there are advantages and disadvantages to both; with some research using a combination of the two as a hybrid approach. The following section will introduce the use of static, dynamic and hybrid data in malware detection literature.

2.3.1 Static Data

Static data is data stored in the file(s) comprising a piece of software, including code that never runs, developer comments, and file metadata. In the case of executables, these files are stored in compiled machine code, designed to be read by computers not by humans. Despite this code not being friendly to human readers, all the information required to produce the desired functionality is still contained in the binary and as such static data has the potential to reveal the functionality of software.

This section lists some static data sources used in recent works and their respective strengths and weaknesses for automated malware detection.

File hashes

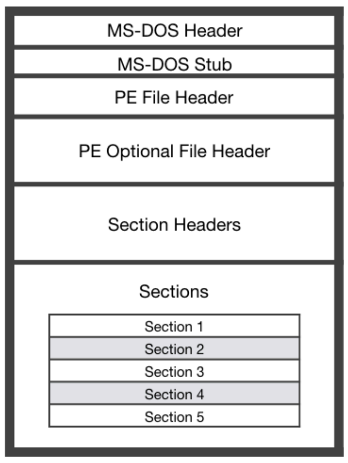
The string of bytes comprising a file can be used to generate unique cryptographic hashes using one-way strong hashing [58], meaning that the hashing is irreversible (one-way) and that the likelihood of two distinct strings giving the same hash is very low [163]. This allows automatic malware detection tools to recognise duplicate file contents even if the files have different names. Hashes can then be stored in blacklists (never allowed to run) or whitelists (only these files may run). Whitelists are impractical for most users and modern businesses as it requires administrative authorisation to run any new file [143]. Blacklisting is an efficient way to rule out known malware but can be thwarted by manipulating a single byte of a known malware to make it appear unknown [58].

Fuzzy Hashing, Import Hashing and Imports

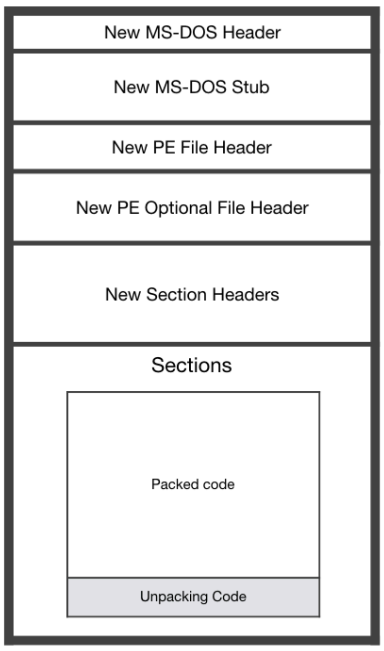
Fuzzy hashing seeks to mitigate this weakness, at least for malware belonging to a known family or variant [133]. Fuzzy hashing is a technique typically whereby a file is split into smaller pieces which are then hashed; these constitute the fuzzy hash. An algorithm then compares the fuzzy hash of two programs to determine how similar they are, high similarity indicates that the two may be variants of the same malware.

Import hashing can similarly be used for Windows executable files to identify variants of the same malware as it creates a hash of the imported functions from other files (such as operating system libraries) and the order of these imports used by the malware and ignores the rest of the code [133]. Changing an import or the order of imports will alter this feature so that variants no longer match.

A number of recent malware detection papers have used imported functions data to automatically detect malware e.g. [117, 140, 218]. This static data can be extracted quickly but can also be evaded by dynamically loading imports at run-time, for example using packing and unpacking techniques[180]. Packing is a method used to compress files which can make static analysis much more difficult (see Figure 2.2) since only the wrapper code for unpacking the malware may be visible prior to unpacking the code. Some packed malware may be unpacked automatically, but when coupled with obfuscation techniques such as dynamic loading of imports manual analysis is required to reveal the source code; rarer and custom-build packers also tend to require manual analysis. Dynamic analysis on the other hand resolve many of these difficulties by allowing the malware to unpack itself in the virtual machine (VM).



(a) Typical PE file



(b) Typical packed PE file

Figure 2.2: Structure of Windows Portable Executable (PE) files

API Calls - Statically Extracted

Some approaches to malware detection look specifically at the Application Programming Interface (API) calls made to the operating system themselves rather than library imports (e.g. [1, 104, 149, 165, 224]. These API calls enable the program to obtain resources and permissions from the host operating system and perform a wide variety of actions. Using debugging software (on unpacked malware) it is possible to pick out every instance of an API call in code, including those that are never called in practice. API calls can also be collected with dynamic analysis, which shows only those that are actually called. The order in which API calls are made may help identify certain malicious behaviours [169] and this can be extracted from static code to generate call graphs [57, 101, 104, 121, 224]. These are graphical representations of the API calls made by software. Call graphs are constructed using debugging tools; debuggers allow analysts to introspect code instruction-by-instruction if necessary.

Malware authors may try to prevent analysts from discovering the code's functionality using a debugger. Anti-debugging refers to the practice of trying to make reverse engineering more difficult. Anti-virtualisation attempts to hide malicious functionality if the sample detects that it is being executed in a virtual environment. Anti-debugging practices are as prevalent if not more prevalent than anti-virtualisation tools. Chen et al. [33] find that 3% of malware samples analysed use anti-virtualisation techniques whilst 40% use anti-debugging and Branco et al. [22] find that 81% use anti-virtualisation with 43% using anti-debugging. There are ways to circumvent anti-debugging techniques just as there are methods for disarming anti-VM manoeuvres. However, the use of a debugger to construct call graphs can be a time-consuming manual process [154].

Strings

Another static feature, which works best with unpacked malware, is simply analysing the human-readable strings in a file. Several researchers have used printable string data as part of automatic detection models (e.g. [100, 178]). Automatic detection systems may look for patterns matching API calls, domain names, shell code instructions or cryptocurrency wallet addresses [65] for example. This can be a powerful attribution tool but may be easily obfuscated or changed as the malware author seeks to evade bring fingerprinted.

Bytecode

Previous work has used histograms (frequency representations) of bytes in a file [192] or created grayscale images from bytes [122], [153] to distinguish malicious and benign software or different malware families. Encrypted malware could thwart this approach as a strong encryption mechanism should give no information away about the original plain-text such that the characters of the ciphertext appear randomly generated, though some structural elements of the file such as the size may still be visible. There are some computational overhead costs to encryption but even if the attacker did not use encryption, recent research into adversarial samples by Kolosnjaji et al. [108] has demonstrated that a state of the art machine learning classifier using bytecode can be fooled by simply manipulating the bytes in the padding of a file. With less than 1% of bytes being altered the detection accuracy of the tool falls by 50%.

Static Signatures

Signature-based detection is the catch-all term for hand-written rules used to distinguish malicious files from other files [133]. These rules are pattern-matching requirements (often strings or bytecode) often combined together with logical operators. Signatures can be a computationally efficient way to detect known attacks and some variants of these attacks. Their detection efficacy on unseen variants depends on the pattern matching filters written by cyber analysts and the sophistication of variant generation used by malware authors. Signature-based methods are not generally well-suited to detecting zero-day attacks. Holm [84] argued that signature-based intrusion detection systems (IDS) may still be able to detect some zero-day attacks due to the prevalence of code reuse among malware authors. The results of the author's research estimated that the signature-based IDS tested, Snort [39], was able to detect just 8.2% of zero-day attacks.

Strengths and Weaknesses

Static data can be a computationally efficient way to evaluate a piece of software and may be carried out in a very short time since data is simply collected directly from code.

Despite the advantage of speed, static data is vulnerable to obfuscation techniques. In essence, this is because it is possible to write software with identical or equivalent functionality in an infinite number of ways, since junk or useless non-executing code can always be appended. Small manipulations to code can render known malware unrecognisable to some antivirus engines [12]. In some cases obfuscated malware can be de-obfuscated as in [101, 104] but this requires manual inspection, which is incompatible with the goal of automation. Bacci et al. [12] demonstrated an advantage of dynamic analysis; that it can process obfuscated and non-obfuscated samples uniformly. The authors found that using dynamic data, an automated detection model achieved comparable detection rates across obfuscated and non-obfuscated samples whilst 'static analysis-based detection is essentially ineffective on obfuscated malware'. With this in mind, the next section considers dynamic data types.

2.3.2 Dynamic Data

Dynamic data captures behavioural information about software as it executes. This data is more difficult to obfuscate than static data because malicious functionality cannot avoid leaving a footprint of its activity. For example, data exfiltration malware cannot help moving or copying data in order to achieve its goals. Obfuscation of behaviours can alter the time taken for functionality to occur or overlay 'decoy' behaviours on top of the malicious functionality but the underlying behaviour cannot be removed without changing the nature of the malware itself.

Dynamic Signatures

Malware analysts have expert knowledge of the common techniques used by malicious software. Some researchers leverage their knowledge of malicious behaviours to look for pre-defined events [120, 196, 206]. This can be considered behavioural signature detection. For example, a pre-defined event may be 'Internet explorer is launched'. These behavioural signatures must be hand-chosen by an expert whilst other data sources can be collected and selected automatically when retraining is required.

Network Traffic

Many common behaviours of malware will use network communication. Command and control centres, botnets and data exfiltration all require network activity. For this reason, a significant proportion of malware detection research has relied either wholly or partially on network traffic data (see Table 2.1 below). Sometimes this traffic is extracted from raw packet capture (PCAP) files. Other approaches take higher-level data from network flows [134], others still use content-based data such as the size of packets being sent [211]. One approach uses entire traffic traces and converts these to images for classification with a convolutional neural network [206]. Network traffic may be a strong indicator of many types of malware but it is not necessary for malware to cause irreparable damage. For example, malware that is designed to operate in air-gapped environments or destructiveware can cause significant damage without contacting out of the network.

API Calls - Dynamically Collected

API calls are the most commonly used dynamic data source in previous literature for malware detection (see Table 2.1 below). Unlike statically collected API calls, dynamic API calls capture those loaded at run-time. Dynamic API call data can be augmented using the inputs passed to the calls [80] and the return values of the calls [15], thus giving more information to the detection tool about how they were used. API calls are discrete events which may or may not be invoked by a given software [158] and can be represented in a number of different ways. Over time the way in which API calls are used may change due to depreciation of the API call or trends in software development. Pendlebury et. al [150] found that a state of the art Android malware detection model using API calls as a data source saw a fall from a 0.85 f1-score to 0.62 when the data is divided according to date. Some API call models have been demonstrated theoretically vulnerable to adversarial attacks [85, 88] but this was validated using only feature manipulation and not through the generation of actual malicious samples.

Machine Metrics

Machine metrics is an umbrella term used in this thesis to refer to measurements of hardware and virtual resources being used by malware at a given time. Examples of these resources include CPU usage, RAM usage, and power consumption. These resources are typically being used by all programs constantly. This contrasts with API calls and expert-defined events which are discrete data points that may or may not appear for a given sample. Previous work has used hardware performance counters [13] and memory read-write counts [139]. Some work, including that presented in this thesis, uses processor and memory usage together with read-write counters [25, 28, 156, 158].

Memory Images

Malware must load data into memory in order to run. Memory images are commonly used in forensic investigations but can also be used for automated analysis. Mosli et al. [131] used data from memory images following an infection to inspect API calls and loaded libraries (often dynamic link libraries (DLLs) in Windows) as well as changes to the system registry. The authors argue that using memory images enables memory-only malware, also known as fileless malware, to be detected. The drawback of memory-image based detection is that the data remaining at the time of the image capture is all the data that can be used, caches may have been cleared that contained evidence of malicious activity. One way around this problem is to take periodic memory images of the endpoint.

User Behaviour

Some automated malware detection systems incorporate data about user behaviour. Saracino et al. [167] create an anomaly detection model to detect malicious Android applications using user behaviour as a metric. Typically, user behaviour is only used in anomaly detection models, those which flag deviations from some baseline of normal behaviour. This is because benign user behaviour is typically bespoke to individual users or subgroups of users. One of the drawbacks of anomaly detection models is that they usually require human investigation to differentiate the innocuous anomalies from the malicious ones, meaning that the system is not fully automated and is therefore exploitable by attackers through flooding a tool with anomalous activity to create a smokescreen for the desired malicious payload.

Dynamic Data use in Previous Work

Year	Reference	API calls	DLLs	Dynamic events	Dynamic logs	File containment	File system changes	Machine	Memory	Network	Opcodes	Prefetch files	Process behaviour	Static API calls	Static Artefacts	Static Bytes	Strings	User behaviour
2007	[14]			X			X			X								
2008	[205]	X																
2008	[21]	X																
2008	[159]	X																
2009	[4]	X																
2009	[15]	X																
2010	[194]	X																
2010	[144]	X																
2010	[61]	X																
2011	[135]								X							X		
2011	[160]	X																
2011	[24]	X																
2011	[10]	X																
2011	[93]	X																
2011	[32]				X													
2012	[175]							X										X
2012	[96]					X												
2012	[60]	X																
2013	[3]	X																
2013	[92]	X															X	
2013	[166]	X									X							
2013	[109]	X						X			X							
2013	[134]									X								
2013	[28]	X																
2013	[45]	X																
2014	[13]							X										
2014	[164]	X																
2014	[193]									X								
2014	[214]	X		X														
2015	[98]	X																
2015	[90]	X																
2015	[139]								X									
2015	[145]	X																
2015	[49]				X													
2015	[178]	X																
2015	[206]	X		X										X	X			
2016	[211]								X	X								
2016	[223]			X											X			
2016	[219]	X																
2016	[177]									X								
2016	[105]	X																
2016	[106]	X																
2016	[80]	X	X															
2016	[87]	X																
2016	[131]	X	X						X									
2016	[196]												X					
2016	[48]	X																
2017	[188]	X																
2017	[7]											X						
2017	[207]				X													
2017	[47]	X									X							
2017	[208]									X								
2017	[161]	X																
2018	[120]			X											X			
2018	[167]	X		X										X	X			
2018	[191]	X																
2018	[99]	X																
2018	[85]	X																
2018	[25]							X										
2018	[171]							X										
2018	[170]							X										
2018	[54]	X																
2019	[215]	X																
2019	[88]	X																
	Total	43	2	6	3	1	1	6	4	6	3	1	1	2	4	1	1	1
	% of 64 references	67%	3%	9%	5%	2%	2%	9%	6%	9%	5%	2%	2%	3%	6%	2%	2%	2%

Table 2.1: Dynamic data sources used in 64 malware detection and classification papers. *N.B. Percentages sum to more than 100 as some research uses more than one data source.*

Table 2.1 shows the dynamic data sources for 64 papers surveyed (64 of 97 surveyed use dynamic data) and gives the percentage of papers for each data source. Some of these attributes were not mentioned in the previous section due to their obscurity; the following list gives a brief description of all data types in the table:

- API calls: Operating system API calls
- DLL = Dynamic Link Libraries
- Dynamic API-based events: user-defined events based on API calls
- Machine: execution data relating to machine statistics e.g. hardware performance counters, CPU use, I/O operation data
- Memory: data relating to memory addresses or memory images during or following software execution
- Network: data relating to network traffic
- Opcodes: machine code instruction
- Dynamic API-based events: user-defined events based on API calls
- Dynamic events: user-defined events during execution
- Dynamic Logs: custom logging from a software provider e.g. operating system logs, AV engine logs, sandbox logs
- File containment relationship graph: graph of contents of compressed file
- File System changes: differences in the file system before and after software activity
- Prefetch files: Temporary files to cache real-time data about software
- Process behaviour: activity relating to individual software processes

- Static API calls: system calls collected without executing software
- Static Bytes: bytecode data collected from static files
- Static artefacts: user-defined static elements of interest
- User behaviour: data relating to user activity

67% of the papers surveyed use API call data with the joint second being expert-defined dynamic events, machine data and network data (9%). As indicated by the publication dates of the paper in the table, the use of API calls has been prevalent from 2008 through to 2019, Machine data on the other hand was used first in 2018 by three separate publications and previously not since 2012/2013. This codification broadly groups the data types but API calls are the most commonly used throughout the 10 year span analysed.

Some papers have compared dynamic data sources. Mosli et al. [131], for example, found that memory image data outperformed API calls and DLLs but existing research is not conclusive as to whether one dynamic data source consistently outperforms others.

Strengths and Weaknesses

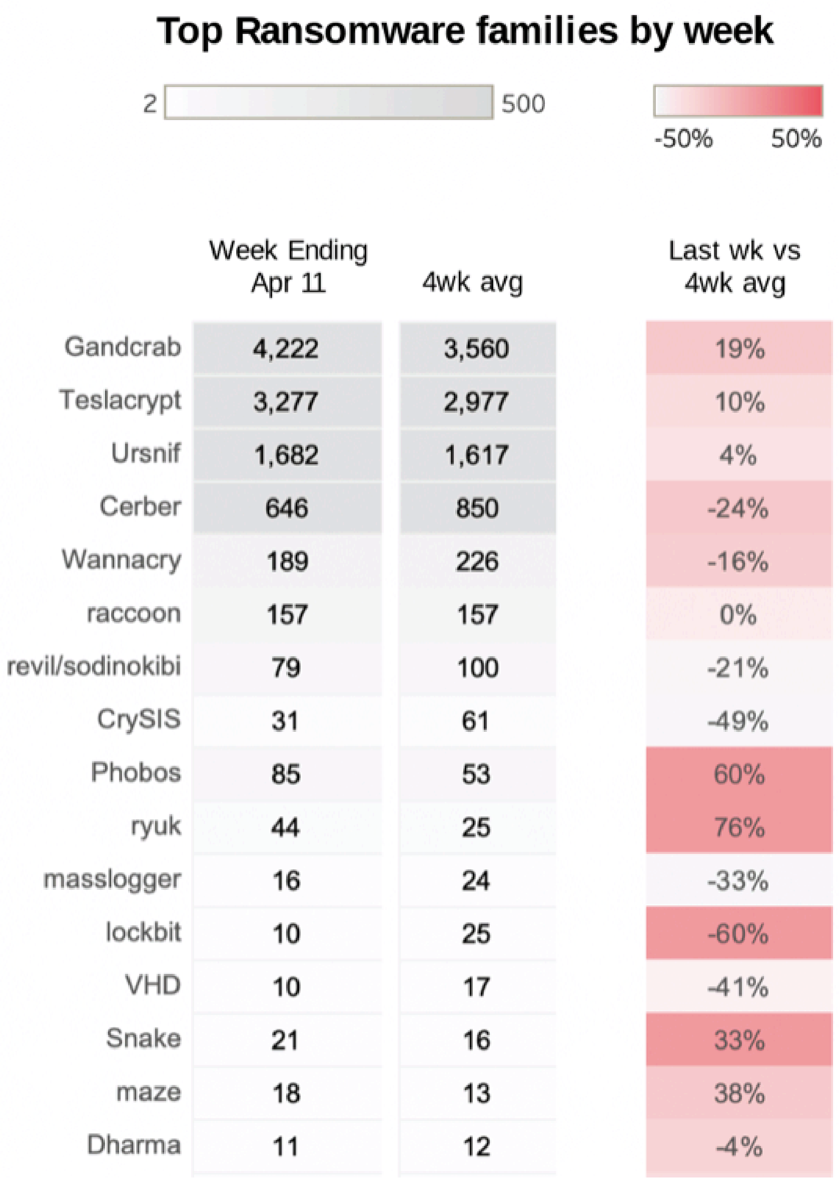
One criticism of dynamic analysis, by comparison with static analysis, is that collecting the data whilst software executes in a sandbox takes (typically several minutes [156]) longer than scraping data from code. The time required for analysis may not matter depending on the use case of the tool. If the tool is intended to be used by analysts as part of an investigation after the file has already executed on an endpoint, then waiting several minutes is not significantly worse than waiting several seconds. If detection is taking place pre-attack rather than post-attack, the time taken for a verdict is more critical as it has the potential to slow business processes or frustrate users.

A second criticism of dynamic analysis is that using a sandbox creates significant computational overheads by comparison with extracting static data. Malware must be allowed to run in order to collect dynamic behavioural data. In order to protect real-world networks and endpoints, malware is often executed in a sandbox. A sandbox is a virtual machine isolated from the internet and other local networks. Sandboxes can consume equivalent resources to a normal endpoint, which some argue is too expensive and unrealistic for private users [139, 170].

A third criticism is that a sandbox may not elicit malicious behaviour. This can occur if the running conditions of the software have not been met; for example, if only a specific version of the software is vulnerable to the exploit used by the malware and that software version is not installed in the sandbox, the malware will not exhibit (all of) its behaviours. Malware authors may also try to detect whether malware is running in a VM and if so, hide the malicious behaviour to avoid analysis.

Turning to the strengths of dynamic data models, these approaches have been shown effective at detecting new malware families. The ability to detect malware that exploits new (publicly unknown) vulnerabilities, known as ‘zero-days’, in particular, is highly prized [73, 84]. Malware detection models use data from known malware to try and detect unknown malware. This has been demonstrated using both static (e.g. [34, 100, 121]) and dynamic (e.g. [120, 140, 211]) data sources. Once a malicious file is known, its unique hash can be computed and referenced and tell-tale signatures extracted in order to block it in the future. Malicious actors make changes to existing malware in order to alter the hash and evade signature detection, this is evident in analysis from organisations like PolySwarm who count thousands of variants per week belonging to a single ransomware family (see Fig 2.3).

Less often, an entirely new *type* of malware appears or becomes prevalent. Whilst dynamic malware behaviours can be diverse, the total possible set of behaviours is far smaller than the total number of ways that malware can be represented in code. Dynamic or behavioural attributes are more likely to be common to different malware families than code components. New malware may well use familiar injection mechanisms, techniques for hiding execution [137], or malicious behaviours such as data



Top Ransomware families by week

2 – 500   -50% – 50%

	Week Ending Apr 11	4wk avg	Last wk vs 4wk avg
Gandcrab	4,222	3,560	19%
Teslacrypt	3,277	2,977	10%
Ursnif	1,682	1,617	4%
Cerber	646	850	-24%
Wannacry	189	226	-16%
raccoon	157	157	0%
revil/sodinokibi	79	100	-21%
CrySIS	31	61	-49%
Phobos	85	53	60%
ryuk	44	25	76%
masslogger	16	24	-33%
lockbit	10	25	-60%
VHD	10	17	-41%
Snake	21	16	33%
maze	18	13	38%
Dharma	11	12	-4%

Figure 2.3: Figure of trending malware samples reproduced with permission from PolySwarm [151] e-Newsletter 15th April 2021 showing thousands of samples of the same malware families observed in a single week.

exfiltration. Conversely, the code to achieve a particular behaviour can be written in an infinite number of ways.

It is possible to set some objectives of a dynamic detection model based on these weaknesses including *(i)* quick to run *(ii)* low computational complexity and *(iii)* generates useful data for distinguishing malware and *(iv)* is able to detect new variants and families. This thesis will tackle *(i, ii)* and *(iv)* with *(iii)* being out of scope but critical future work to this research field.

2.3.3 Hybrid Approaches

Hybrid approaches combine both static and dynamic data. Using both static and dynamic data is often justified by arguing that more data will produce a better-performing model [92, 166, 167, 178, 206, 219, 223]. This argument seems reasonable, but by using both data sources, researcher are often introducing the problems associated with *both* static and dynamic data into a model. Hybrid approaches which include static strings [92, 178] do not account for adversaries compiling strings during real-time, and static approaches can easily be manipulated [108]. The addition of dynamic data will not necessarily mitigate these weaknesses. Machine learning models often find the shortest route to solving a problem; if there is a static single string that is highly correlated with the presence of malware but can easily be obfuscated but there are also a set of dynamic behaviours that are well-correlated, the model does not know to prefer the dynamic behaviours over the string since all data is presented equally to the model. The introduction of dynamic data will however introduce temporal and computational overheads, the absence of which is one of the key advantages of using static data.

Multi-stage approaches, which mimic a defence-in-depth security architecture. may use static analysis to filter out known malware variants *followed by* dynamic analysis. This model of hybrid analysis better harnesses the respective strengths of each data type [136, 177]. Sequential approaches reduce the mean time required for data collection whilst maintaining in-depth analysis for those samples which require further examination, but these approaches still take several minutes to perform dynamic analysis by comparison with several seconds for static analysis. Some researchers have attempted to prove the benefits of hybrid approaches but the results have been inconclusive, as the following section describes.

2.3.4 Empirical Comparisons

The choice of data used for automatic malware detection is arguably more important than the choice of rule-making procedure (machine- or human-generated) as, without useful data, rules will struggle to make distinctions between malicious and benign software. This subsection briefly outlines why existing literature comparing dynamic, static and hybrid approaches cannot be used to draw a conclusion about which is best and explains that the frequency with which static data is used may be due to the relative ease of collection.

Empirical results (see Table 2.2) could be used to try and determine which approach is best but this is contingent on using the same test set for all experiments and not drawing unsubstantiated conclusions. For example, Islam et al. [92] argue that using dynamic and static data used together creates a more robust model. The authors test their ML model, trained with data from 2003-2007 and then with data from 2009-2010 and find that there is only a small decrease in classification accuracy percentage with accurate labels. However, the authors do not compare these results against static-only and dynamic-only data which makes it difficult to conclude that the hybrid model is *more* robust even if it performs well. Some research has directly compared static, dynamic and hybrid results on the same set of samples [47, 92, 166, 178, 223].

4 of the 5 papers (Santos et al. [166], Islam et al. [92], Yuan et al. [223], and Shijo and Salim [178] all find that the hybrid combination of static and dynamic data produces better results than static or dynamic alone; reporting between 1 and 6.5 per-

Ref.	Detection Accuracy (%)			Data Type			Test Samples Ratio	Metric notes
	Static	Dynamic	Hybrid	Static	Dynamic	Hybrid	Malware : Benignware	
Damodaran et al. [47]	87.43*	**98.11***	81.02*	Opcode + API calls	Opcode + API calls	train on dynamic test on static to save time	745:40 (approx 13:1)	Weighted AUC-ROC average used, accuracy not reported
Islam et al. [92]	87.8	90.4	**97.01**	function length frequency and printable string information	API calls and parameters frequency	combineds	2400:541 (approx 5:1)	weighted accuracy results to reflect class imbalance, FNR and FPR rank reflected in accuracy rankings
Shijo and Salim [178]	95.88	97.16	**98.71**	printable string information	API-[3 and 4]-grams	combined	997:490 (approx 2:1)	model ranking the same for FNRs and FPRs
Santos et al. [166]	95.9	77.26	**96.6**	opcode sequences	Author-defined behaviours from API call data	combined	1:1	All models achive a high TPR but the dynamic model has a significantly higher FNR
Yuan et al. [223]	89.03	59.09	**96.76**	Pre-defined sensitive API calls and permissions	Pre-defined sensitive API calls	combined	1:1	Hybrid has highest TPR and lowest FNR, Static has slightly higher FNR but also much higher TPR

Table 2.2: Reported accuracies, data types and test samples ratio for 5 papers comparing of static, dynamic and hybrid data. *=Area Under the Curve of Receiver Operating Characteristic multiplied by 100 reported instead of accuracy. Metric notes column comments on the metrics used to evaluate the classification, especially when using unbalanced dataset. None of these papers reports F1 but all except [47] report FNR (false negative rate) and FPR (false positive rate) or inverse metrics (true rates).

centage point detection accuracy increase. Damodaran et al. [47] and Ye et al. [219] report dynamic and static data respectively to achieve the highest classification accuracy. This appears to make a strong case for the use of hybrid methods in order to produce maximum detection accuracy. However, this thesis argues that the inclusion of any static features introduces vulnerabilities through manipulation of static data, especially when used by machine learning models due to the threat of adversarial ML, as argued above.

Of all 5 papers, 3 [47, 92, 178] report that dynamic data achieves better results than static and 2 [166, 223] report the reverse. Nataraj et al. [135] do not attempt hybrid classification but compare static binary texture analysis to dynamic forensic image data and find that the dynamic data outperforms the static data for malware family classification (98% vs 95%), though dynamic analysis takes 4,000 times longer.

Although the research summarised in Table 2.2 uses the same test data, it is difficult to draw conclusions about which data type performs better. Each paper has chosen different static and dynamic data features for comparison, thus the conclusions are that feature set A (which happens to use static data) is better than feature set B (which happens to use dynamic data) for the given test set, but it may be that feature set A uses highly indicative static features and set B particularly poor dynamic features or vice versa.

The metrics also make comparison difficult. Accuracy and true/false positive/negative rates are the most commonly reported (see Table 2.2 caption) but even these do not see consistent results in comparing static and dynamic data.

The Convenience of Static Data

In selecting a type of data, it should be noted that static data is less resource-intensive for researchers to collect. More data is widely acknowledged to increase the performance of machine learning models (to a point) [226] as the model is more likely to form rules based on trends that apply to the general population of cases rather than just those of the dataset. To summarise: more data leads to better machine learning performance on external test sets and given a fixed time period and fixed resources more static data can be collected than dynamic. A number of papers criticising dynamic data also infer that the use of dynamic data is desirable in future work [100, 125, 153]. This may be taken as an indication that dynamic data collection is seen as a more involved or arduous process but a valuable one, as Raff et al. [153] explain: ‘[the] dynamic analysis component is likely to be an important component for a long term solution, we avoid it at this time due to time complexity’.

Figure 2.4 shows the number of benign and malicious samples used by 97 malware detection and malware family classification papers. Some use all malicious datasets as these papers are attempting to cluster malware into families or groups of similar behaviour. Though this is not the focus of this thesis, many of the methods used to group

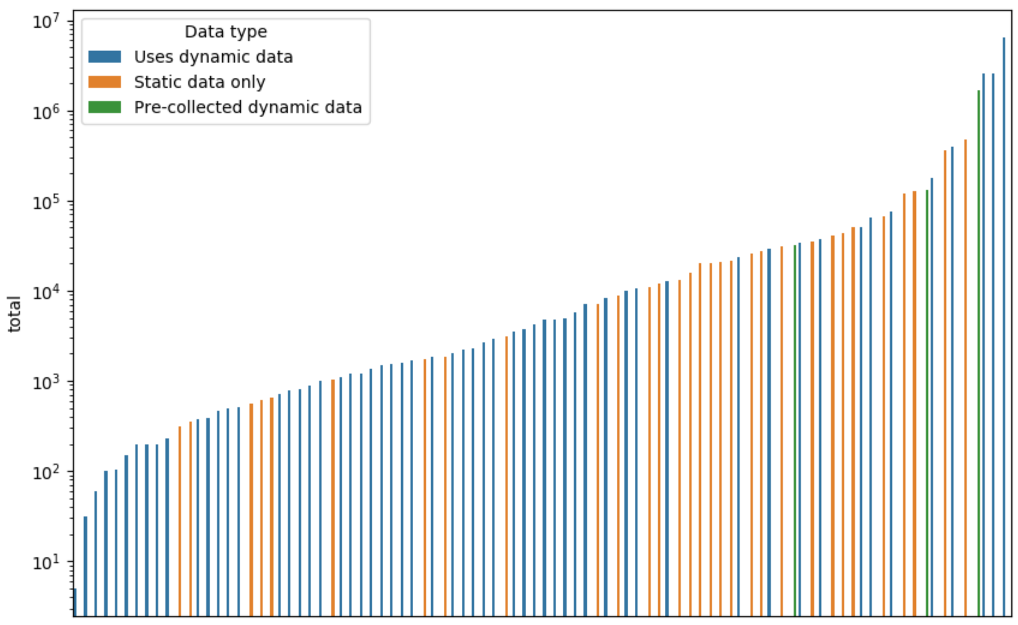


Figure 2.4: Number of samples used in 92 papers (Only 92 of 97 report number of software samples) coded by use of self-collected dynamic data. *N.B. The penultimate two papers use the same dataset.*

malware families are similar to those used for malware detection and thus relevant to the discussion. Figure 2.4 gives an overview of the number of samples used by a number of recent papers. A logarithmic scale is required to present this data since the smallest dataset uses just 5 samples [175] (though the number of benign samples used by the authors appears to be missing from the paper), whilst the largest uses 6.5 million samples [87], with the median size for dynamic data being less than 2,000. It is clear that the papers using static data tend towards the right-hand side of the graph, i.e. using more data.

		Dataset size			
Data type	# papers reviewed	minimum	maximum	median	mean
Uses dynamic data	58	5	6,500,000	1,916	219,058
Static data only	31	314	477,349	19,987	50,693
Pre-collected dynamic data	3	32,078	1,647,000	131,650	603,576

Table 2.3: Dataset size statistics for 92 papers (Only 92 of 97 report number of software samples) using dynamic data (includes hybrid approaches), static data only, and those papers using dynamic data that was pre-collected prior to the research paper being presented.

There is a negative correlation between the use of dynamic data and dataset size. Table 2.3 records statistics around the dataset sizes for 92 malware detection and malware classification papers. The three largest datasets (6.5m, 2.6m and 2.6m samples) all use dynamic data, but these papers [45, 87, 96] were all published by Microsoft's research labs and are anomalously large (6.5m, 2.6m and 2.6m) by comparison with other commercial and academic dataset sizes ; the mean is 220K with Microsoft datasets included and more than 10 times less at 19K with these datasets excluded. To illustrate how these outliers skew the mean, the table reports the median dataset sizes. The median size when dynamic data is included is 20% of the size of datasets only using static data. Since more static data can be collected in a given time period per computational resource, larger datasets can be generated.

2.3.5 Focus on Dynamic Data

This thesis focuses on dynamic malware detection based on the analysis of relevant literature above. This thesis argues that dynamic data would be a valuable addition to endpoint protection if it were as fast and easy to deploy as static analysis. Fundamentally, it is more difficult for malware authors to obfuscate the behaviour of malware than the source code. However, this is not reflected in the popularity of static malware detection; as discussed in the previous section.

The computational and time resources required for collecting dynamic data are certainly greater than for static data which means that (1) static malware detection has been researched more often and (2) dynamic detection is part of a deeper layer when used in a defence-in-depth model. Defence-in-depth is a cyber security concept referring to the 'enclosure of an asset by a succession of barriers, all of which must be penetrated for the asset to be acquired' [181], these obstacles may promote any and all of the detection, delay and mitigation of a cyber attack [181] through their diversity, such that the attacker cannot bypass more than one using the same technique. All layers of a defence-in-depth model add security to the endpoint, with the strongest models seeing each layer complement the weaknesses of the other layers. Therefore this thesis is concerned with dynamic data for malware detection and the robustness of this data in a number of respects which have not yet been addressed by existing work and are elaborated in Sections 2.6 2.7 and 2.8. Before describing these gaps in detail, the next section analyses the second fundamental aspect of automatic malware detection: the rules.

2.4 Detection Rules

The rules used for malware detection may be signature-based or heuristic [179]. Signature-based detection uses the existence or absence of specific attributes, sometimes together with logical operators to identify malware. In practice, signatures are an efficient way to filter out known malware variants and are widely used (e.g. [8]) but are unlikely to capture new variants and require manual effort to update.

Heuristic approaches do not depend on specific attributes of known malicious software but instead depend on broader patterns delineating malicious and benign software. As such, heuristic rule sets are usually better-suited to detecting unknown malware variants and families [179]. Machine learning (ML) is one automated method to generate heuristic rule sets.

Rules may be created using expert human knowledge, statistics, an algorithm or some combination of the above.

2.4.1 Human-generated Rules

Some researchers have used expert knowledge to craft the rules used to distinguish malware and benignware. Isohara et al. [93] use hand-written regular expressions to find matches in the dynamic behavioural logs collected during software execution. This approach found suspicious activity in a significant proportion of the 230 applications tested by the researchers, but the results did not indicate the usefulness of such an approach in the wild since the authors do not verify whether any of the suspicious applications are actually malicious. Hand-written rules can be highly effective but may not be suited to detecting new threats. For example, the market value of cryptocurrency has led to an increase in cryptomining malware but this malware may not be captured by filters written prior to cryptominers being a prevalent type of malware. ML-generated rules may also fail to detect cryptominers if they have not been exposed to them during training, but it is possible to retrain the model (generate a new set of rules) in minutes and without using up the time of experts.

2.4.2 Automatically-generated Rules

ML is a sub-field of artificial intelligence that uses an algorithm and data to generate the rules required to solve a given problem. The same algorithm can be employed to tackle different tasks by changing the underlying dataset rather than explicitly programming rules for each problem. A machine learning algorithm approximates complex functions in order to solve a particular task. The tasks can be varied but include classification (e.g. malware or benignware), anomaly detection, and clustering[72].

ML algorithms automatically generate rules from data to map inputs to outputs. For malware detection, software collected from data is mapped via (a series of) rules onto a label of malicious or benign. These rules are typically mathematical expressions and can be difficult to explain in easily interpretable language [126]. Over 60% of 2019 papers in a search of the Google Scholar archive mentioning ‘malware detection’ also mentioned ‘machine learning’, ‘deep learning’ or ‘neural networks’ (see Fig.2.5). Many papers cite the success of machine learning and deep learning in other fields as reason enough to apply it to malware detection (e.g. [87, 107, 145]). Others such as Hardy et al. [81] and Zhang et al. [224] proffer that ML can be a solution to obfuscation and polymorphic techniques.

The rules that are generated depend not only on the data but the problem that the algorithm is trying to solve.

2.4.3 Types of Machine Learning Problem

ML models take data as input and output some value. Models undergo a training (learning) phase, which is usually followed by a testing (inference) phase. During the training phase, the parameters of the algorithm are updated with respect to some optimisation expression e.g. for malware detection the greatest overall agreement between the algorithm's labels of malware or benignware and the ground truth as to whether the software is malicious or benign. The models may undergo subsequent training with new data after the initial training phase has concluded, e.g. to update the model with new data.

ML models can be categorised according to the feedback provided to the adjustment algorithm: supervised, unsupervised and reinforcement learning all describe different methods for rating the output of a model during training.

Supervised learning

Supervised learning occurs when the ML algorithm has access to the ground-truth labels associated with the training data.

Supervised learning algorithms iteratively adjust the parameters of the model to minimise the disparity between the ground truth labels and the output of the model. In the case of supervised learning for malware detection, the inputs are data collected from software and the output is a label of 'malicious' or 'benign'.

Unsupervised learning

Unsupervised learning does not use any labels and semi-supervised learning uses a combination of labelled and unlabelled data.

Unsupervised learning algorithms may have some other goal such as clustering similar data together. At some point (e.g. number of times the data has been parsed or a certain error rate), the training of the model stops and testing begins. During testing new, unseen data is fed to the model to generate an output, to mimic the real-world performance of model inference on unseen data. The outputs of unsupervised learning models can be numeric indications of how unusual a sample is or how closely it resembles (a subset of) the training data.

Reinforcement Learning

Reinforcement learning (RL) uses a reward function to assign a positive or negative value to a (sequence of) actions within an environment e.g. the points scored in a computer game environment can be used to reinforce the likelihood of repeating the actions preceding the score if the model finds itself in similar circumstances again.

Malware detection as a Machine learning problem

The volume of data produced by the hundreds of thousands of new malware each day [203] presents a challenge for analysts. One way to address this problem is to use machine learning to draw out patterns from a large quantity of data.

Malware detection is typically framed as a supervised (is this malicious or benign?) or unsupervised learning problem (how anomalous is this compared with benignware?). Supervised learning required the labelling of samples, which can be labour intensive, though many researchers use VirusTotal as a proxy labelling oracle for research e.g. [25, 87]. Unsupervised learning does not require labelling but it does require the model creator to select a baseline and a threshold of deviation from that baseline over which a sample is judged to be malicious. Deciding the threshold presents a number of challenges if labelling is not used otherwise how can it be certain that the baseline dataset does not contain any malware. Both approaches have been used in previous work.

2.4.4 Machine Learning for Malware Detection

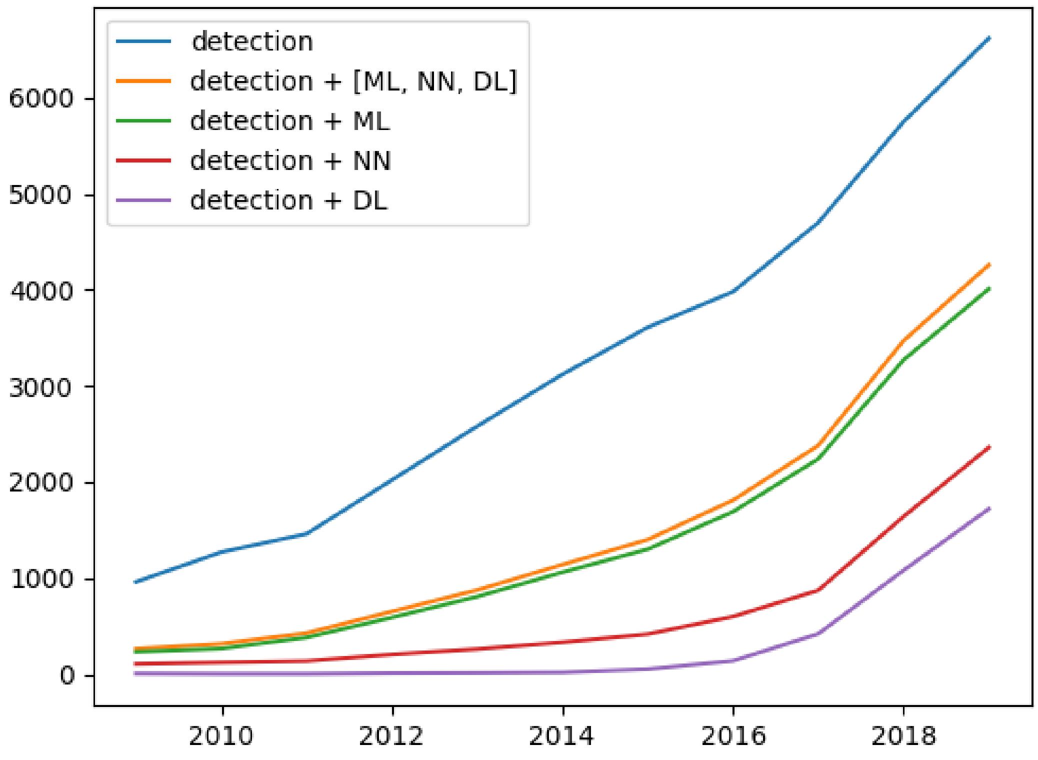


Figure 2.5: Approximate number of search matches per annum reported by Google Scholar [118] in academic texts between 2009 and 2019 for
‘malware detection’ only
‘malware detection’ and (‘machine learning’ or ‘neural network’ or neural networks’ or ‘deep learning’)
‘malware detection’ and ‘machine learning’
‘malware detection’ and (‘neural network’ or neural networks’)
‘malware detection’ and ‘deep learning’

Machine learning has become an increasingly popular tool for malware detection as in other domains. Figure 2.5 illustrates the year-on-year increase in papers using the term ‘malware detection’ with papers containing both ‘malware detection’ and ‘machine learning’ from the Google Scholar [118] archive. This is a crude metric since papers containing the term will not always have malware detection as a research topic but it is possible to see a linear increase in the number of ‘malware detection’ papers but a growing rate for those papers containing both ‘malware detection’ and ‘machine learning’. In 2009 24% of papers containing the term ‘malware detection’ also contained the term ‘machine learning’, in 2019 this figure was 61%.

The reason that more malware detection publications are using (or referring to) machine learning may be explained by the changing problem of malware detection and/or the recent ubiquity of machine learning.

Machine learning addresses a number of the challenges in automatic malware detection. The volume of new malware is too great for humans to parse but ML models are capable of drawing out patterns from very large datasets, as demonstrated by [45, 86]. The two categories ‘malicious’ and ‘benign’ describe a wide variety of software; malware is often sub-categorised into families or types. Malicious software is any software that causes harm; this describes a very broad spectrum of software behaviour. Benignware is any software that is not malware; an even broader set of behaviours. It is very difficult to hand-write a set of rules to make a binary distinction between these internally heterogeneous groups. As malware evolves over time in response to new vulnerabilities and anti-malware techniques, this rule-set may need to be rewritten or augmented. The expertise required to reverse engineer a program is rare and as such it is valuable and expensive. ML offers a quick and cheap means to generate a new rule set.

Benefits of Machine Learning for Malware Detection

In academic research, ML is a popular mechanism to generate rules for malware detection e.g. [5, 25, 34, 47, 57, 76, 81, 85, 100, 100, 131, 140, 153, 167, 196, 208, 215, 218], but it is difficult to know if AV engines are making use of this technology due to commercial secrecy.

Fleshman et al. [62] tried to partially answer this question by comparing 2 ML models against 4 AV products. The AV products were significantly worse at detecting these unseen malware despite good detection on their known counterparts, by comparison with the ML models; 13%-78% error rate for AVs, 0.003% and 0.02% for ML models. Not only does it appear that the AVs tested are not making use of the technology used by ML models but that the detection mechanisms being used are not able to detect new variants easily.

Despite the large number of publications, and perhaps this is one reason why there are so many, there is a lack of consensus for how best to tackle the problem.

2.4.5 Algorithm Choice in Recent Work

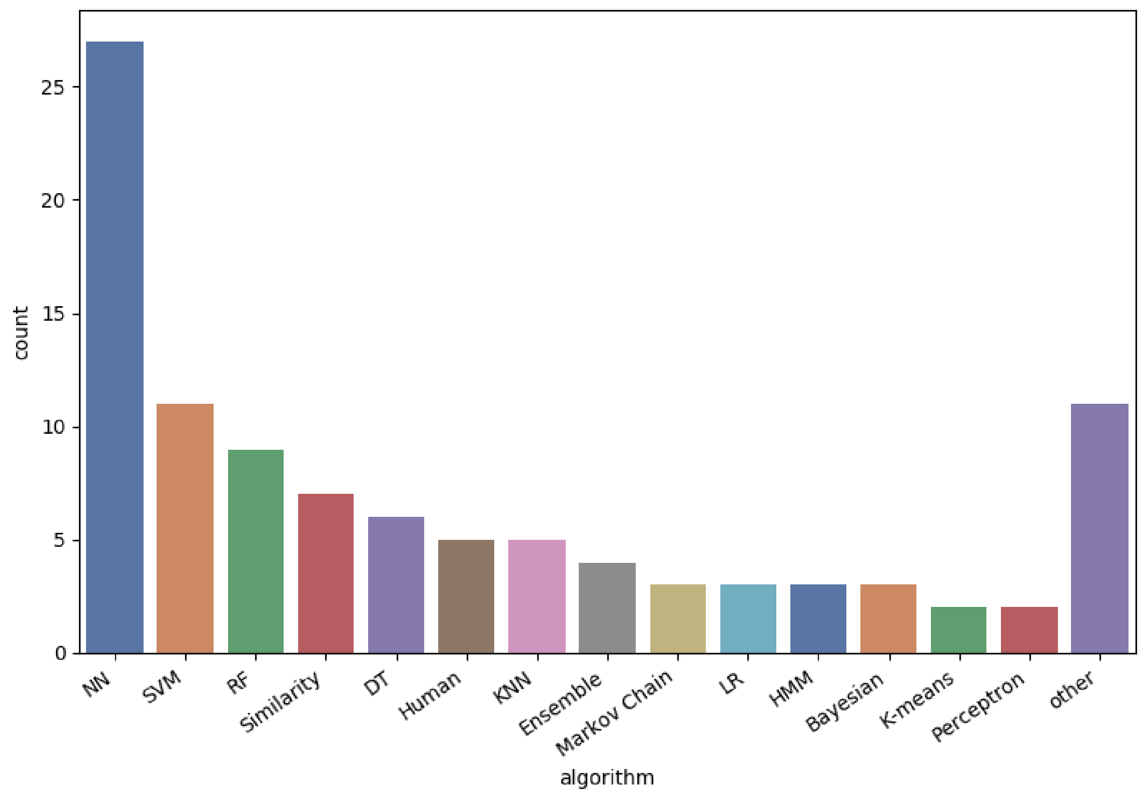


Figure 2.6: Algorithms used by 97 recent malware detection papers. NN=neural network, SVM = support vector machine, RF = random forest, DT = decision tree, KNN = K-nearest neighbours, LR = logistic regression, HMM = Hidden Markov Model, Human = human-generated rules. The ensemble algorithms may use a number of techniques to combine the predictions of different models.

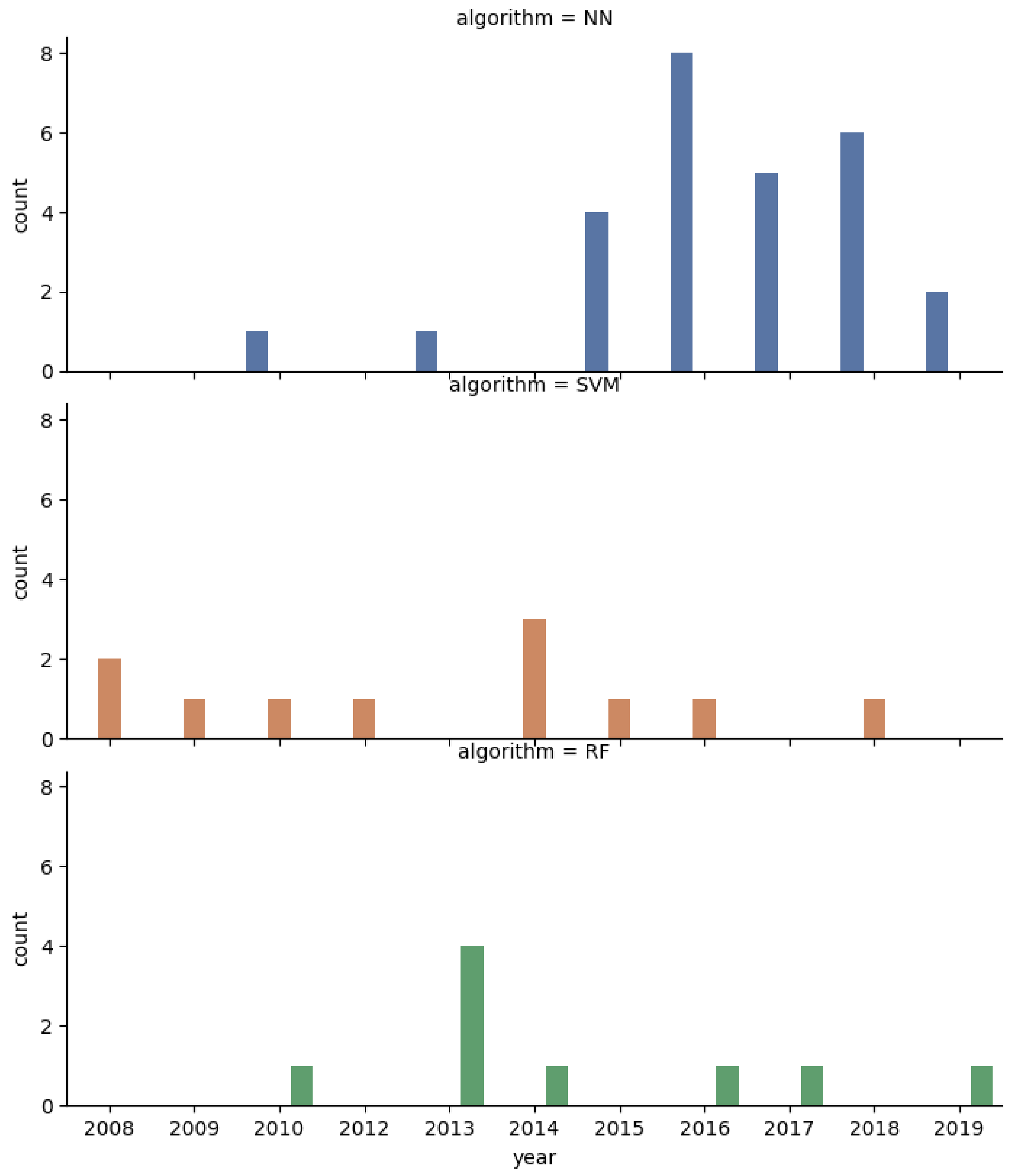


Figure 2.7: Total papers using neural networks (NN), support vector machines (SVM) and random forests (RF) in 97 recent malware detection papers by year

There are many different ML algorithms, and many of them have been applied to the problem of malware detection. Common considerations in the choice of the algorithm are the error rate and computational overhead during training and inference. Despite the wide range of ML algorithms available, some are more prevalent than others. In an extension of the work published in [158], a survey of the algorithms and data used by 97 recent malware detection and malware family classification papers has been conducted. Figure 2.6 summarises the most popular algorithms used by the papers in the survey. Some papers compare multiple algorithms, in which case the best-performing algorithm is selected according to the evaluation criteria set by the paper authors. There are more algorithms than papers as some approaches use the output of one algorithm to feed into another for a final decision (e.g. Burnap et al. [25] use a self-organising feature map together with a logistic regression). Any algorithms appearing in only one paper have been grouped together as ‘other’. The three most popular algorithms used for malware detection are NNs, support vector machines and random forests. Figure 2.7 shows a significant increase in the popularity of neural networks since 2015. Support vector machines (SVM) have fallen in popularity by contrast, whilst random forests (RF) have seen a steady appearance in the literature since 2010. It is possible to speculate that the rise of deep learning, which is often attributed to developments in accelerating hardware making training on large datasets possible [127], has led to a preference for the use of large datasets where possible. More data often means that a wider sub sample of the population data is captured and therefore the model will generalise well. SVMs on the other hand do not handle large datasets well due to their training complexity since the separating hyperplane must take more data points into account [30]. Random Forests and gradient boosting machines (thanks to efficient implementations [59]) on the other hand still train quickly with large datasets.

Neural Networks

The most popular algorithm by far is the neural network, but this is partly because neural networks describe a range of diverse architectures including but not limited to: long-short-term memory networks, convolutional neural networks and autoencoders. Perceptrons are counted separately in Figure 2.6, but these are also a type of (fully connected feed-forward) neural network. Neural networks are general function approximators, which map input data to outputs through a network of connected neurons. First conceived of in the 1950s, NNs are resource-intensive to train by comparison with many other ML models. The advancement of computer hardware and greater accessibility to this hardware has allowed these algorithms to realise their many uses. It is possible that the increase in the proportion of malware detection papers using machine learning can be attributed to the wider adoption of neural networks following their success in other domains such as computer vision e.g. [110], natural language processing e.g. [46], and game-playing e.g. [128] amongst others. Some malware detection research has explicitly borrowed approaches from computer vision papers, e.g. turning malware into images prior to classification [122, 219], but there are domain-specific challenges in malware detection that do not translate from computer vision problems, one of which is that labelling software as malicious or benign requires expertise that is more difficult to come by than the expertise required to label benchmark image datasets.

Comparison with Wider Algorithm Choice

Whilst this survey is not exhaustive, its findings are consistent with the general popularity of these algorithms as mirrored in Goldblum's analysis [70] of the most commonly-used winning algorithms in machine learning competitions hosted by Kaggle [91] with the exception of Gradient Boosted Machines (GBM), of which XGBoost is one. Figure 2.8, from Goldblum's analysis, shows that the algorithms chosen for malware detection mirror the wider trends in machine learning. This may indicate that there are a limited

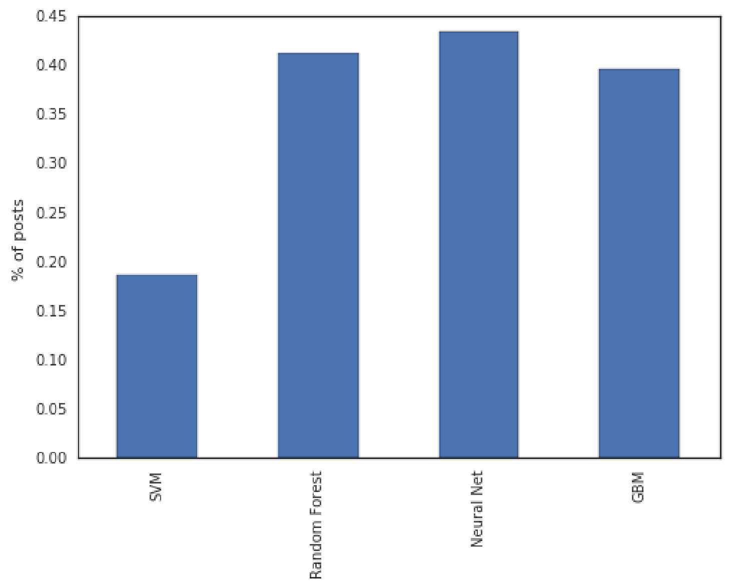


Figure 2.8: Most popular algorithms winning Kaggle machine learning competitions to 2016 [70]

number of data types or classification problems and that the optimal algorithm depends on which category the problem falls into, rather than the problem space. Whilst there are trends, there is not a best algorithm for malware detection, the best-performing algorithm depends on the input data, both the features chosen and the specific samples in the dataset. This makes it difficult for practitioners to know how to adopt one of these proposed models in practice.

2.5 Practitioner Choice

ML for automated malware detection is motivated by a desire to prevent the destructive impacts of malware. Imagine a practitioner wanting to adopt an automated malware detection model using dynamic data to complement existing static antivirus tools - how will she choose which of the many proposed approaches to adopt?

Unlike other popular machine learning domains such as computer vision, benchmark datasets for malware detection are not readily available. The lack of benchmark datasets makes it difficult to compare approaches. The changing landscape of real-world malware is a fundamental challenge to overcoming the issue of benchmarking. Researchers can reasonably argue that a more up-to-date dataset should be used than was used by previous work.

The composition of the testing and training set can artificially inflate the reported performance of a model by making the classification problem slightly easier. The details of datasets are not always reported but one can imagine that if malware and benignware are delineated on other grounds e.g. all the benignware is pre-2010 and the malware is post-2015 or the malware is almost all from just one family, the model may be learning to distinguish software by date rather than malware and benignware. Often grouping is incidental to the fact that malware is collected from one source and benignware another (an argument in favour of honeypots or commercial datasets). As an example, some papers (e.g. [71, 98, 173, 188, 225]) derive all of the benign samples in the dataset from the operating system being defended. This is likely to limit the diversity of benign samples that the model learns for detecting malware. Furthermore, if the samples used to test the model are also from this pool, the reported accuracy metrics of the model may be artificially high by comparison with how it will perform in classifying benign software created by other software developers.

The test set can comprise any combination of the millions of malware not in the training set. In order to try and gauge real-world performance, the test set(s) should mimic the conditions under which it will be used. This can be challenging if it is intended for use in many different environments. Furthermore, the test set should comprise samples created after all the training set samples - this is because a training set including malware from *after* the test set samples may have leaked information to the model about future malware behaviours, even if they belong to different families. Pendlebury et. al [150] have demonstrated the exaggerated accuracy metrics of android malware detection models that can be achieved when tested with samples created before or during the creation times of the training set. Whilst separating the data according to date approximates some of the conditions that the tool will be subject to in live deployment but it does not help the problem of direct comparison between approaches.

To judge whether a model has learned anything at all and in order to compare approaches on the same dataset, empirical evaluations are reported, and are used by every paper in this literature review. However, they are not always used consistently.

2.5.1 Evaluation Metrics

All malware detection papers report some kind of accuracy metric. Different metrics are used by different papers and the datasets are often also distinct, making these numbers virtually impossible to compare. Some papers use highly imbalanced datasets (many more malicious than benign samples or vice-versa) which changes the usefulness of different metrics. A dataset comprising 90% benignware and 10% malware can class all samples as benign and report 90% accuracy (percentage with correct labels) on this dataset since accuracy is simply the percentage of correct classifications over the whole test set but in reality, the model is useless at distinguishing benignware and malware.

The murkiness of the accuracy metric can be illuminated by reporting the false positive (misclassified benignware) and false negative (misclassified malware) rates. In the real-world software ecosystem, there is often a higher percentage of benignware than of malware [150]. A high false-positive rate could hinder user experience if it means that benign programs are often blocked or have to be investigated by human analysts. A low false-negative rate may be difficult to achieve in practice if there are only two malware samples in one month, failing to detect one will yield a false negative rate of 50%, even though only a single sample was undetected.

Comparing more than one metric can make it difficult to ascertain which of several models is the best. A number of metrics try to account for imbalance and still report a single figure.

Often the F-Score metric, also known as F1-Score or F1, is used to describe the accuracy of predictions on an unbalanced dataset. F-Score gives the harmonic mean of precision ($\frac{\#\ true\ positives}{\#\ positive\ predictions}$) and recall ($\frac{\#\ true\ positives}{\#\ positive\ samples}$) so the true positive rate dominates the true negative rate for this metric (see Table 2.4). In the case of malware detection, it is generally true that if no malware is detected, the model is not performing well, therefore the F-Score is used along with other metrics in this thesis when the dataset is imbalanced.

Predictions for dataset:			
+ + - -	Text description	Accuracy (% with correct labels)	F-Score
+ + - -	True labels	100%	1
+ + + +	All positive	50%	0.67
- - - -	All negative	50%	0

Table 2.4: F-Scores vs. percentage with correct labels (accuracy) for three sets of predictions on a dataset of 4 samples (2 positive, 2 negative)

Often papers will report multiple metrics to try and capture the strengths and weaknesses of the proposed models over previous works but these metrics are somewhat irrelevant for comparing papers unless the same dataset is used and may also fail to reflect performance on an unseen test set drawn from a different environment for the reasons given above.

2.6 Model Robustness

Metrics are not uniform across papers and good performance on one dataset does not necessarily imply good detection accuracy on another. This is a crucial problem for practitioners seeking to adopt these models since ultimately they are interested in how well a model will perform in their environment.

Testing a model on multiple datasets gives additional confidence that a model may perform well on another dataset. Some researchers have tested this [135] but, this has only been conducted with static malware data. Nataraj et al. [135] test a binary texture analysis approach on 4 separate datasets for malware family classification (not detection) and observer varying results (72% to 97% accurate labels assigned); the authors also retrain the (k-nearest neighbour) models on each dataset rather than deploying pre-trained models. Dovom et al. [57] look at static malware family classification, rather than detection, as well as malware detection and see a detection accuracy variance between 93 and 100% accuracy across 4 distinct datasets. Pendlebury et al. [150] test the impact of splitting datasets strictly by date and the impact of altering the percentage of malware in the training and testing set on accuracy metrics, across three different algorithms, but these experiments use static features and do not compare datasets form different sources in during testing. McLaughlin et al. [125] also test a static analysis approach, this time using three datasets (one collected by the researchers and two provided by a commercial entity). The authors observe very good results on the researcher-collected dataset (98% accuracy, 0.97 F1-score) but poorer results when tested on the commercial dataset (80% accuracy, 0.78 F-Score), finding that another method gives more stable accuracy results across the two datasets. The researchers investigate the use of different data sources but not different algorithms.

Previous work has not conducted this kind of multi-dataset analysis for dynamic malware detection; perhaps owing to the effort of collecting dynamic data. McLaughlin et al.'s [125] work indicates that the best-performing model on one dataset may not perform so well on another. For the benefit of researchers and practitioners, it is important to understand whether results are likely to be stable across datasets. This presents an open question in dynamic malware detection literature:

RQ1: To what extent do dynamic data and algorithm choices impact the robustness of malware detection accuracy across different datasets?

Following this question around the robustness of dynamic malware detection models between lab and real-world environments, there are additional barriers to adopting dynamic data models. One of these is time; the processing required to collect dynamic data is much more intensive than that required for static detection.

2.7 Detection Time

After looking at the performance and robustness of different algorithms and dynamic data types in various contexts, practitioners may still resist dynamic data collection due to the time required to process a sample.

Ref.	Reported time collecting dynamic data
Malware Detection	
[195]	30 seconds
[164]	2 minutes
[161]	2 minutes
[98]	2 minutes
[80]	3.33 minutes (200 seconds)
[135]	3 to 5 minutes
[196]	5 minutes
[214]	5 minutes
[29]	9 minutes
[211]	10 minutes
[134]	15 minutes
[47]	Fixed time and 5-10 minutes mentioned but overall time cap not explicitly stated
[145]	At least 15 steps - exact time unreported
[159]	No time cap specified but ‘The samples are executed for a limited time’
[61]	No time cap mentioned - implicit full execution
[4]	No time cap mentioned - implicit full execution
[87]	No time cap mentioned - implicit full execution
Malware family classification	
[107]	No time cap mentioned - implicit full execution
[59]	No time cap mentioned - implicit full execution

Table 2.5: Reported data sample sizes and times collecting dynamic behavioural data per sample

Table 2.5 outlines the dataset sizes and reported recording time for several dynamic detection papers. The shortest time used is 30 seconds, the longest is 15 minutes, others are measured in the number of instructions, API calls or other discrete events carried out by the malware. Many papers do not report this data at all [4, 59, 87, 107].

Previous dynamic malware detection work has used data recording times chosen for experimental convenience [47, 196] or for the time deemed long enough by the researchers for malware to enact its payload [4, 61]. Researchers have not asked how long malware needs to distinguish itself from benignware because the goal of these works was not to conduct analysis in a short period of time.

To fully leverage the usefulness of dynamic detection in order to prevent the execution of harmful code, this thesis argues that it should be used *before* malware is able to cause harm. This leads to the second research question:

RQ2: Is it possible to predict that software is malicious from its early stages of execution?

If it is possible to predict that a file is malicious based on the early stages of execution, this could be a viable malware detection model to challenge the speed with which static data can be collected and use dynamic detection as part of front-line defence mechanisms rather than for post-infection analysis, which, as highlighted earlier could save a several million-euro remediation cost[141]. Even if the time is reduced to a few seconds of analysis, there is an easily-exploited weakness in early detection models, as explained below.

Looking to the future, it is important to consider likely vulnerabilities of any new approach to malware detection. Using a very short behavioural trace could be circumvented by ‘sleeping’ the program or running a benignware program for a specific amount of time. Logic bombs, by which a certain condition must be met before the malicious payload is triggered are well-documented in malware and difficult to detect [95]. If malware is allowed to run indefinitely, e.g. to allow for a logic bomb that releases the payload only at 2AM, this defeats the early prediction model. Some malware will only execute when there is no user activity, some in the middle of the night, some only if it is connected to the internet. There are ways to try and convince the malware these conditions have been met despite the true conditions and continue analysis in a VM but this is time-consuming and requires human analysis on a case-by-case basis and is therefore incompatible with quick, dynamic malware detection.

One way to ensure that the behaviour exhibited by malware is representative of the damage it could inflict on a target endpoint is to run the malware on said target endpoint and examine the sample as it executes. This is known as online, run-time, on-the-fly or real-time malware detection [48, 170, 188]. The lack of naming convention indicates the relative immaturity of this sub-field of dynamic malware detection. The term ‘real-time’ will be used henceforth.

2.8 Real-Time Malware Detection

Real-time detection builds on the weaknesses of early-detection in a VM since it does not require an additional sandboxed environment and uses the target endpoint to monitor malicious processes. The first advantage is that, this eliminates the computational resources required to run the VM and second, it ensures that conditionally executing malware will show it’s true behaviour if it is to cause any damage at all. The second advantage is that any malicious activity that will impact the endpoint will also be seen by the analysis system, unlike the case of the sleeping malware using the early-detection approach. Real-time detection does not invalidate the usefulness of early-detection, it depends on it since real-time detection is carried out on a live endpoint in use by a user which presents new challenges beyond the use of a dedicated virtual machine running single software applications one-by-one.

The next section will set out these challenges in detail but briefly, the impact of power consumption and model latency needs to be considered; the issue of signal separation between malicious and benign processes needs to be tested, and consideration

	Problem considered										
Ref.	Resource consumption	Latency	Signal separation	Early detection	Impact of detection time	Real-time tested	Malware types	OS	# Samples	Features	Algorithm
[170]	X	X					General	Linux	200	HPCs	Boosted DT
[171]	X	X					General	Linux	200	HPCs	Boosted DT
[48]	X	X		X		X	General	Linux	798	API calls	MLP
[139]	X			X		X	General	Windows	1554	memory addresses, instructions	NN
[222]		X				X	General	Windows, Linux	500		NN
[188]		X	X	X		X	General	Windows	9992	API calls	RF + NN
[172]		X		X	X	X	Crypto Ransomware	Windows	497	File data	Rules

Table 2.6: Real-time malware detection literature problems considered. OS = operating system; HPCs = Hardware performance counters; DT=Decision Tree; MLP = multi-layer perceptron; NN = Neural Network; RF=Random Forest

of how to prevent damage should also be examined. Table 2.6 gives an overview of the research gaps in this domain before the following subsections discuss it in more detail.

2.8.1 Latency and power consumption

The data collection and machine learning model cannot be too heavyweight computationally. If the computation is very slow user experience may be impacted and the benefits of avoiding the VM are less persuasive. This is the most well-addressed aspect of real-time detection in previous work. Sayadi et al. [170, 171] analyse the impact of a number of features and machine learning algorithms and find that using fewer features with an ensemble classifier improves accuracy whilst maintaining low resource consumption. Das et al. [48] use a field-programmable gate array (FPGA) to implement a low-power, low-latency multi-layer perceptron.

Sun et al. [188], Ozsoy et al. [139] and Yuan [222] propose two-stage models to address the need for lightweight computation. The first stage comprises a lightweight ML model such as a Random Forest to flag suspicious processes, the second being a deep learning model which is more accurate but more computationally intensive to run. Two-stage models, as Sun et al. [188] note, can get stuck in an infinite loop of analysis in which the first model flags a process as suspicious but the second model deems it benign and this labelling cycle continues repeatedly. Furthermore, if the first model of the two is prone to false negatives, malware will never be passed to the second model for deeper analysis.

This previous work gives valuable insights into real-time malware detection algorithms but does not examine the tangible impact of using a low-power or low-latency model on the ability to prevent harm or detect malware earlier, leaving open the following question:

RQ3: Does the use of lower latency algorithms impact upon the speed of accurate malware detection?

2.8.2 Signal separation

Real-time detection on an endpoint in use requires us to distinguish between malware and benignware that are running simultaneously. Typically malware detection models collect behavioural data for the entire (virtual) machine whilst a single application executes. However, on an endpoint in use, it is necessary to look at the behaviours of the individual processes. This is more challenging than application-level detection because a model is only able to observe partial components of the behavioural trace in each constituent process. Sun et al.[188] present the only previous work to attempt process-level detection with up to 5 applications running simultaneously and were able to detect between 88% of applications successfully after 5 minutes of execution. A user may well run more than 5 programs, which leads us to ask whether there is an impact on malicious process detection when a greater number of programs are running:

RQ4: Is it possible to separate the signals of tens of simultaneous benign and malicious processes?

2.8.3 Automated Response: Process Killing

Finally, real-time analysis introduces the risk that the malware will execute. Real-time analysis aims to prevent problems rather than simply detecting that they have occurred after the fact. In the same way that self-driving cars need to detect a ‘STOP’ sign as early as possible (rather than 10 minutes after it has passed) in order to mitigate the chances of a road accident, real-time malware detection should detect malware before the malicious payload is delivered. This is more challenging than full-trace dynamic classification because it must use a partial execution trace rather than the complete behavioural footprint. The earlier a malicious process is detected and killed, the more likely it is that damage to the victim machine/endpoint has been mitigated or even prevented. This means making decisions based on short behavioural traces and automating the response since humans may not be able to see the alert and react quickly enough to prevent damage, which can begin within seconds [43]. Automated responses must be carefully calibrated in order not to hinder business operations, which may happen if there are a significant number of false positives (benignware classified as malicious).

Das et al. [48] look at early detection of entire malicious applications (signal separation problem not addressed) and report the classification accuracy as a percentage of the software execution such that 46% of malware was detected in the first 30% of execution but 97% were detected using the full trace. It is not clear what detection at 30% of the way through execution means for the user. Sun et al.'s work on malware process killing [188] has used behavioural traces of 5 to 10 minutes, but this is sufficient time for malware to cause damage; one security vendor has found that Chimera ransomware can encrypt 70 MB worth of files (stored in 1,000 documents) in 18 seconds [43]. Scaife et al. [172] do address this gap by measuring the damage of ransomware, with most files being saved by the model. However, the researchers only look at ransomware malware rather than building a general malware detection model and only test on 5 benignware samples so an indication of false-positive rate is not evident. The final research question is therefore:

RQ5: Is it possible to detect and kill malicious processes early enough to prevent malicious damage?

2.9 Thesis Scope

This thesis is concerned with malware affecting Microsoft Windows, which is the most-used desktop operating system (OS) globally [138]. Many malware detection papers address malicious applications for Android mobile OS e.g. [1, 24, 73, 76, 93, 99, 100, 113, 125, 149, 167, 173, 175, 206, 213–215, 219, 223, 224] as this OS is even more widely used than Windows [69] and sees limited scrutiny of third-party applications. This thesis is concerned with desktop rather than mobile operating systems due to the prevalence of desktop machines in business by comparison with mobile devices; therefore the malware and benignware relate to Microsoft Windows.

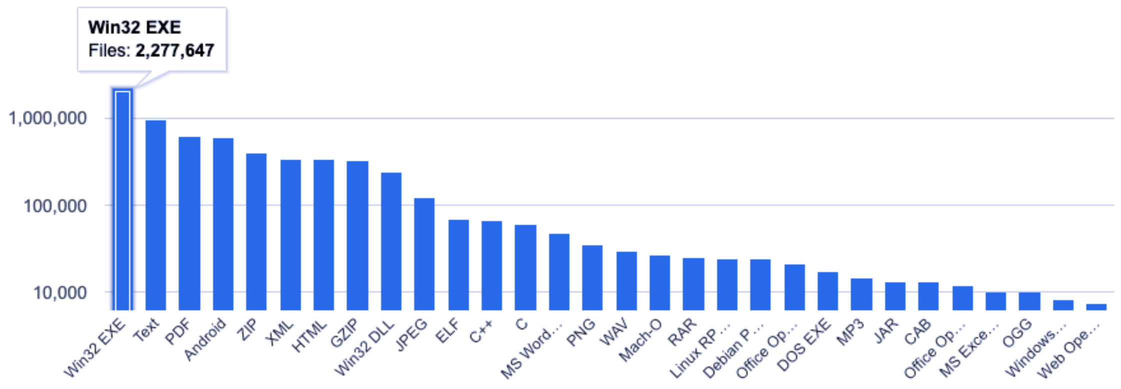


Figure 2.9: Most commonly uploaded filetypes to VirusTotal in the 7 days preceding 21st August 2019 [203]

The primary filetypes considered in this thesis are executables. VirusTotal [203] reports these as the most commonly uploaded filetype for analysis (see Figure 2.9); though Chapter 3 also reports some experimental results for PDF, Microsoft Word and Microsoft Excel filetypes.

2.10 Conclusions

The literature review in this chapter has presented the state-of-the art in dynamic malware detection work. The motivation for malware detection research is clear: malicious software is responsible for economic and social disruption [82, 185] and the associated costs of malware attacks are worsening with time [119]. The very large volume of malware seen each day [203] necessitates automated detection. Dynamic malware detection models have not been demonstrated to be vulnerable to obfuscation to the same degree as static models [76, 108, 207] but the time taken to collect dynamic data has made these models preferred for post-attack analysis over pre-attack detection.

Whilst in-depth analysis of malware can give valuable insights into existing vulnerabilities and the threat landscape, being able to detect malware before it causes damage has the potential to eliminate the cost of recovery.

This research advances the problem space of automatic malware detection through the experimental results presented in Chapters 3, 4, and 5. These chapters address the research questions presented in the previous chapter for which existing literature has provided only partial answers; this has resulted in several contributions to the field of automated malware detection. Related research questions not addressed by this thesis and those arising from the work presented herein are discussed in Chapter 6: *Limitations, Future Work, Conclusions*.

Chapter 3

Robust Models

This chapter considers a common scenario: a model is trained and tested by researchers and then used in a new environment. As noted in the previous chapter, McLaughlin et al.'s work [125] on static data indicates that the most robust model may not be the best-performing in the 'lab' setting. Previous work has not examined whether this robustness may be impacted by the choice of *dynamic* malware features or the machine learning algorithms used.

This chapter addresses *RQ1: To what extent do dynamic data and algorithm choices impact the robustness of malware detection accuracy across different datasets?*.

Section 3.1 outlines the motivating scenario for this chapter, the problems collecting datasets for malware detection, and defines 'robustness'. Section 3.2 describes the methodology used, Sections 3.3 and 3.4 outline the data sources and algorithms that will be used in experimentation. Section 3.5 describes the data collection and Section 3.6 reports and analyses the experimental results. Finally, section 3.7 summarises the implications of these results.

3.1 Motivation and Definitions

The key motivation for this chapter is the following scenario: a security practitioner reads that multiple academic groups have developed high-accuracy dynamic malware detection models and would like to use one in her environment - should she choose the model reporting the highest accuracy?

Ultimately the practitioner's goal is to prevent the most malware damage using the chosen model whilst simultaneously minimising the false positive rate. Over time the model which works best for the practitioner may change, as the malware landscape and attacker techniques change. Here the question is limited to thinking about the best model to implement today, though longevity is an important concern.

Two questions that the practitioner may ask are: 'How can I compare models A and B?' and 'How does the test set used by model A reflect my real-world environment?'. The first is difficult to answer due to a lack of benchmark datasets. The second either requires the practitioner to be able to check the similarity between their own malware/benignware ecosystem against the samples used by the researcher for testing *or* the practitioner may look for evidence that model performance is robust under datasets from different provenances.

3.1.1 Lack of Benchmark Datasets

Chapter 2 illustrated that the datasets used to train and test machine learning models are rarely the same. This presents a challenge for the practitioner seeking to compare models.

In the case of malware detection, the samples evolve over time [150] as malware and benignware evolve due to changes in technology, opportunities for cybercrime and other factors. Researchers can thus justify the need for newer datasets, since old datasets may be considered sufficiently good indicators how a model may perform today. Obtaining the samples is a non-trivial exercise. Malware is produced in the hundreds of thousands on a daily basis and is not difficult to find using open-source repositories such as VirusShare [201]. More laboriously it can be collected using honeypots, with the effort justified in potentially collecting lesser-known, newer or target-specific malware [6, 66]. Benignware can be more difficult to obtain in large volumes as there is less motivation to collecting and catalogue it.

So why do researchers not simply reuse datasets collected by previous work? There may be several reasons behind this phenomenon. *(1)* Some researchers use industry-collected datasets which are not shared for security or commercial secrecy reasons e.g. [45, 87] *(2)* Academic-collected datasets are often not shared due to security concerns around responsible malware distribution and legal concerns about proprietary benignware distribution. *(3)* Commonly used benignware is often not free [44], thus creating financial barriers. *(4a)* Some researchers publish the datasets derived from the software samples *but* if a new research team would like to try collecting different data types these datasets are redundant and *(4b)* as mentioned above, these datasets go out of date quickly. *(5)* Models are typically not released publicly, possible also for security concerns, making it difficult to replicate results. Given the difficulty of comparing the various approaches, how can security professionals discern which approaches are likely to yield the best results in practice?

3.1.2 Robustness

The term ‘robustness’ can be used to broadly describe the resilience to a range of testing conditions. The National Institute for Standards and Technology (NIST) Special Publication 800-137A *Assessing Information Security Continuous Monitoring (ISCM) Programs* defines robustness as “The ability of an information assurance (IA) entity to operate correctly and reliably across a wide range of operational conditions and to fail gracefully outside of that operational range.”[52]. More broadly the Oxford English dictionary gives the following definition of ‘Robust’ relating to computing: “Of a program:

able to recover from errors; unlikely to fail, reliable. Also: (of a program's feature set) powerful, full."[53]. In machine learning, it is important to consider the robustness of models, especially those that may be used in critical decision-making contexts [142]. Robustness may be discussed for different forms of testing. For example, a building may be robust over time or robust to earthquake tremors but it may not be robust to both.

For malware detection, the robustness of a machine learning model refers to the stability of the testing accuracy as the underlying distribution of the testing data changes. This underlying distribution may change for several reasons, three likely trials are (i) different malware and benignware seen in different digital contexts or ecosystems e.g. different companies, geographic regions (ii) changes in malware and benignware over time, known as 'concept drift' (iii) adversarial samples; which are samples designed to exploit weaknesses in the ML model.

3.1.3 Contextual Differences

Contextual differences may be used to describe the differences between datasets collected from two different contexts. The contexts may be two different organisations, different malware or benignware repositories, the same organisation but in two different countries etc. In statistics, sampling bias is the term used to describe the deviation from the true population. It is easy to imagine that a detection model could be deployed in two different organisations, a local government office and a technology start-up for example. The technology start-up may see more malware seeking to steal intellectual property and the local government office may see more data exfiltration malware seeking to capture public records. How might one increase the chances that a model performs well in both contexts? A simple answer is to ensure that the model has seen both types of malware during training.

In practice, the distribution of malware families seen in two distinct ecosystems may not be known. If they are known, it is still difficult to capture the variety of malware that an organisation may be targeted by since, as illustrated in Chapter 2, the dataset used is usually thousands - at most millions [45, 87]) of the available malware and benignware, which is a very small fraction when one considers that there can be more than 1m new distinct files produced in a single day [203]. Afroz notes that it takes an average of 1 month for commercial antivirus products to settle on a decision as to whether or not a given file is malicious (and to which family it belongs) [2].

3.1.4 Concept Drift

Concept drift describes this degradation of model performance owing to changes in the underlying distribution of the population being classified [197]. Due to the constant evolution of malware, concept drift should be an important concern for malware detection researchers. Saxe and Berlin [168] distinguish malware from benignware using a deep feed-forward neural network trained on static features with a true-positive rate of 95.2%. However, the true-positive rate falls to 67.7% when the model is trained using files only seen before a given date and tested using those discovered for the first time after that date, indicating the weakness of static methods in detecting completely new malware. This last example illustrates that even when reliance on hashes is removed, static features can still fail to capture new malwares. The novelty of a malware sample by comparison to previous ones may be difficult to measure, some malware families such as Emotet or Zeus have been in existence for several years; the connecting thread between successive iterations may be the infection mechanism, spreading mechanism, evasion techniques used or some other method. Since this is different for different malwares it can be difficult to say how ‘new’ a malware is; many borrow code from predecessors. However, the results of Saxe and Berlin’s work indicates that even within the same year new samples may have evolved significantly to evade detection. Pendlebury et al. [150] recently highlighted the oversight of several well-cited papers to test the robustness of detection models against malware that will be written in the future.

3.1.5 Adversarial Samples

Adversarial samples are samples generated by an attacker to try and fool a machine learning model into misclassification. This is a concern for automatic malware detection as malware authors are highly motivated to try and bypass any antivirus tools. Some work has already been conducted to show the vulnerability of ML malware detection models [76, 85, 88, 108, 153, 207]. There are many kinds of adversarial attack. Grosse et al. [76] were the first to demonstrate adversarial attacks on ML malware detection models; with a 90% accurate model falling to 40% accurate when samples were crafted to evade model detection using injected code. More recently, Kolosnjaji et al. [108] showed that simply altering the padded code at the end of a compiled binary could cause the 94%-accurate *MalConv* model proposed by Raff et al. [153] to misclassify 60% of samples. One notable caveat in these works highlighted by Afroz [2] is that researchers have often failed to validate whether the malware still carries out its malicious goals after manipulation. Similar approaches have been proposed for dynamic data models [85, 88], specifically for dynamic models using API call data. These approaches rely on injecting data into the machine learning model rather than creating samples that will fool the model itself thus requiring a greater degree of access to the target system than the static attacks, which is less likely to be possible. In theory, an attacker could relatively easily manipulate the sequence of API calls to fool ML models though researchers have not demonstrated this yet and this should be taken into consideration in selecting a robust data source.

3.1.6 Malware Detection Robustness - Focus on Contextual Difference

This section has outlined three issues relating to the robustness of malware detection models: context, concept drift and adversarial samples. Whilst all are important considerations, this chapter focuses on contextual difference due to its relative lack of attention in previous literature: as this section and Chapter 2 have shown, a number of researchers are addressing the problems of concept drift and adversarial machine learning. McLaughlin et al. [125] are the only previous work to look at the change in performance when a model and this work uses static data. Furthermore, the contextual difference is an immediate problem for ML models once they are deployed and do not require the evolution of malware or an attacker to be sufficiently motivated to try and circumvent a model in order to see diminished detection accuracy.

3.2 Methodology

The experimental methodology employed in this chapter is to take the most commonly-used data sources and algorithms for malware detection research and compare the classification accuracy for each model configuration on an unseen test set drawn from the same source as the training data and then on a test set from a commercial organisation.

The reason for choosing the most commonly-used data types and algorithms rather than the configurations reporting the highest accuracy is that this repeated use and publication of certain algorithms and data types is itself an indication of the robustness of a particular approach.

This section outlines the experimental approach to testing model robustness across multiple contexts. The constituent parts of a machine learning model for malware detection (Figure 3.1) may take a wide variety of forms. Chapter 2 introduced malware

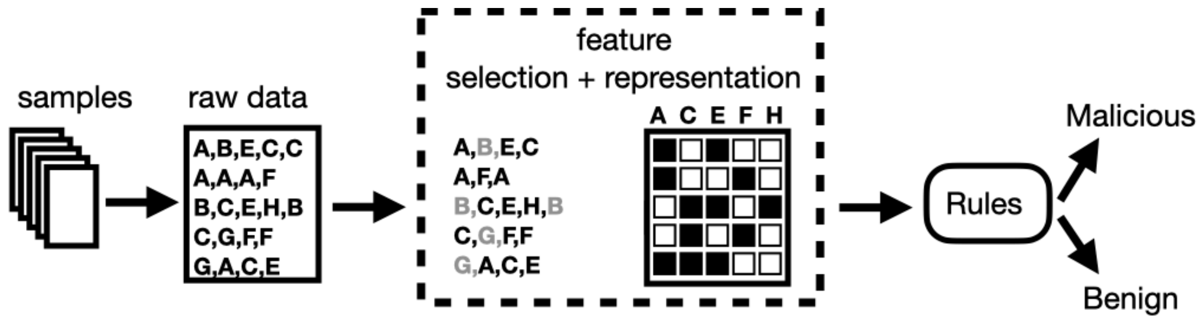


Figure 3.1: Representation of a machine learning malware detection classifier pipeline with intermediate steps between collecting raw data and parsing by a rule set to obtain a final decision. Feature selection excludes features that do not meet a certain criteria whereas feature representation describes the data structure used by the rule set.

detection dynamic data types and algorithms used by previous work but did not discuss feature selection or feature representation.

3.2.1 Feature Selection

Feature selection is a process for reducing the number of inputs to a model either using algorithmic or hand-picked methods. This is typically carried out to reduce data dimensionality [72, 87] which is a means to reduce the computational requirements for training models as well as the memory requirements for storing data. For example, Lungana et al. [120] reduce 1,143 features to 570 for a malware detection model using a feature selection process based on correlation with the classification labels and mutual information with other features i.e. features are omitted if they have a high correlation with another feature [212]. On the other hand, Wu et al. [214] use all 56,354 collected features with no feature selection for Android malware detection. The decision to use feature selection depends on the use case and the computational resources available.

3.2.2 Feature Representation

Feature representation refers to the way in which data is presented to a model, this could be the binary existence of a feature, the frequency with which a feature occurred, this frequency could, in turn, be represented as a histogram and the data fed to the model as a 2-dimensional matrix. Some malware detection research has borrowed ideas directly from computer vision papers, e.g. turning malware into images before classification [122, 219]

3.3 Choosing Data

This section describes the data types, feature selection methodologies and feature representations used for experimentation.

3.3.1 Features

Chapter 2 included a table (Table 2.1) of dynamic malware detection papers and the data used for detection. This data showed that API calls are the most popular by some margin, accounting for the data source in 67% of the papers surveyed. The (jointly) next most popular are expert-defined dynamic events, machine data, and network data (each 9%). After these three types, the next most popular achieve 6% or less of the total share.

Although the next most popular data types are around 60 percentage points less common, in order to have something to compare against API calls, machine metrics and network data are used in the experiments. Expert-defined dynamic events, which constitute things like ‘internet explorer is launched’ or a specific set of API calls in succession, are not included because the expert-defined features may change over time and cannot be automated.

3.3.2 API calls

	Representation												Domain knowledge		Additional data		
		N-grams															
Reference	Binary	2-grams	3-grams	4-grams	5-grams	7-grams	10-grams	Frequency	Feature-vectors	Image	Time-series	Call-graphs	Hand-chosen	Clustered by type	Arguments	Null-terminated	Return values
[205]											X						
[159]									X						X		
[21]												X	X				
[4]											X				X		
[15]												X		X	X		X
[144]												X					
[61]								X									
[194]	X																
[93]	X												X				
[160]	X	X	X	X											X		
[24]								X									
[10]												X					
[60]			X														
[45]		X	X					X							X	X	
[92]								X									
[166]	X												X				
[109]												X					
[28]		X						X					X				
[3]		X															
[164]	X												X		X		
[214]		X	X					X									
[178]			X	X													
[98]											X						
[90]											X						
[145]													X				
[105]											X						
[106]			X														
[80]								X							X		
[87]		X	X					X							X	X	
[219]			X							X							
[131]								X									
[161]											X						
[188]											X						
[47]											X						
[167]								X					X				
[191]					X	X											
[99]											X						
[85]											X						
[54]												X					
[88]								X									
[215]							X										
Total	5	6	8	2	1	1	1	11	1	1	10	6	7	1	8	2	1
% of 41 references	12%	15%	20%	5%	2%	2%	2%	27%	2%	2%	24%	15%	17%	2%	20%	5%	2%

Table 3.1: References using API calls as data, representation method and additional data collected. Where papers have used multiple representations (e.g. [61]) the best-performing representation is recorded)

Table 3.1 documents the *representations* of dynamic API calls from those papers identified in the background chapter. As the table shows, API calls have been represented in a number of different ways. One challenge in representing these data is that thousands are called in a short space of time and these long sequences must typically be condensed in some way in order to be digested by a model.

The most popular (28%) is a frequency-based representation (simply counting the occurrence of API calls) followed by time-series representations (24%). 3-grams are the third most popular (20%), which cluster API calls into sequential triples and use these as features. N-grams (sequences of length N) constitute 29% of API call representations but the N value is not agreed; Szeles et al. [191] noted that 5- and 7-grams gave the highest accuracy whereas Faruki et al. [60] found that 3-grams were best (20% of papers).

This chapter uses the two most popular representations: frequency and time-series representation.

3.3.3 Network data

Network data is the second most common feature used (see the previous chapter) but the data collected and representation varies across different research. The most common approach (3 of 6 papers) tracks numeric statistics e.g. the number of connection attempts, packets transmitted/received, bytes transmitted/received [14, 177, 211]. Less common is the tracking of IP address and port data (2 papers) [134, 208] or of the nodes in a local network hosting a particular file[193]. 2 papers count the protocols used to transmit packets [134, 177]. Due to the prevalence of numeric data, one network traffic feature set is used.

3.3.4 Machine Metrics

Machine metrics like network data are also represented in several ways. The most commonly monitored aspects are memory usage,[25, 109] CPU usage data [25, 175] and the number of processes, [25, 175]. Only one paper each looks at input/output read/writes [109] and battery level (for mobile devices) [175]. Researchers looking at Linux systems have used hardware performance counters [13, 170, 171]. This data is represented as a time series otherwise as summative data has not been used in previous work. The consensus on representation is low due to the small number of papers, but taking those metrics with at least two papers the machine activity feature set user here comprises user CPU usage, system CPU usage, memory use, swap use, the total number of processes, maximum process ID and time in seconds since execution began captured each second.

3.3.5 Summary of feature sets and representations

Feature selection is sometimes used to reduce the dimensionality of data but due to the numbers of distinct API calls observed during data collection being low and the representation not increasingly data dimensional (as an N-gram approach would), no feature selection methods are used.

Four file types are tested in these experiments: executables (EXE), Portable Document Format files (PDF), Microsoft Excel spreadsheets (XLS) and Microsoft Word documents (DOC). These four categories are chosen as these types constituted significant numbers of the malicious files detected by the organisation.

The description of features sets and number of features are listed below - note that the number of API calls varies for each filetype since different API calls are observed in the execution of each set of training samples, these are indicated by a set of values in curly braces:

- **API frequency [{62,117,165,277} features]:** total number of times each unique

API call is observed.

- **Sequential API calls [{62,117,165,277}$\times$time features]:** API calls observed in sequence, these are grouped into 1-second clusters to provide a time-series representation and to capture the temporal co-occurrence of calls, since the optimal N for N-grams

- **Network [4 features]:** total packets sent, total packets received, total bytes sent, total bytes received

- **Machine metrics [7 features$\times$time]:** user CPU usage, system CPU usage, memory use, swap use, the total number of processes, maximum process ID and time in seconds since execution began captured each second

Further to these feature sets models are trained using all features to see if this has any impact on accuracy across datasets.

- **All metrics [{73,128,176,288} features$\times$time]:** API call frequency and network data and machine metrics

3.4 Algorithms

As outlined in the previous chapter, the most popular algorithms for malware detection mirror the most popular used in other data science domains (in Kaggle competitions domains at least). As a reminder to the reader, the three most common were: neural networks, support vector machines and random forests. In order to reduce the number of possible models that can be tested, these three algorithms are used for analysis in this chapter. A brief description of the three algorithms and their learning mechanisms follows:

3.4.1 Neural network

Neural networks are brain-inspired general function approximators which map data from inputs to outputs through a network of neurons connected by weighted paths. A feed-forward neural network or multi-layer perceptron is the simplest form of neural network. Multi-layer perceptions were first proposed by Rosenblatt in 1958 [162] as a simplified model of information processing by neurons in the brain. Hardware acceleration has enabled multi-layer and very large perceptrons (deep neural networks) to be trained quickly.

The success of a neural network (NN) can be highly dependent on the hyperparameters used to configure it. The following experiments use the configuration proposed by Huang and Stokes [87] as their research achieves the highest dynamic malware detection rate reported at 99.69% accurately labelled samples. The NN is a feed-forward network with 2000 rectified linear unit neurons in a single hidden layer, trained with a dropout rate of 0.25. Dropout is a technique used in training NNs by which neurons are randomly switched 'off' (activation is set to 0) during training, which is widely found to help models generalise better. Although a more optimal hyperparameter configuration may exist for each dataset, previous works show borrowing configurations from other papers e.g. [108] therefore this practice is used in order to mimic existing research

In these experiments, the PyTorch [147] python library was used to implement a feed-forward neural network. Of the papers surveyed using neural networks, 5 use recurrent neural networks (RNNs) [85, 105, 145, 161, 215], 4 use convolutional neural networks (CNNs) [108, 125, 208, 219] 3 combine both [106, 188, 196], and 5 use feed-forward neural networks [48, 75, 87, 100, 120]; the others use a further type of network such as an auto-encoder [81]. The type of neural network depends on the data (representation) being used and the type of learning problem being solved. Autoencoders are typically used for preprocessing data or for unsupervised learning problems. This malware detection problem has ground truth labels and is therefore framed as a supervised learning problem. RNNs are used for time-series data and CNNs for image processing. Since the API call frequency data is not time-series, feed-forward networks are used.

Support Vector Machine

The support vector machine idea was first proposed by Vapnik and Lerner in 1963 [200].

The kernel of an SVM determines the possible geometries of the separating plane for SVMs. Some malware detection research does not report the kernel function used [113, 159, 178, 214] but those which do overwhelmingly used the radial basis function (RBF) kernel [5, 13, 21, 21, 206, 211, 211]. The RBF kernel is capable of creating non-linear separating planes. These experiments use the RBF kernel as it is the most commonly used in the research surveyed of those papers which declare the kernel type. This was implemented with the Scikit-learn python library [23, 148] implementation of a support vector machine with default parameters (apart from the kernel type) as parameter values were not reported for most papers, therefore, it is assumed that default configuration was used. Although this may seem counter-intuitive and that the best hyperparameters should be chosen for each of the $4x10$ cross validation folds datasets, again, previous works [13, 21, 211**?**] show borrowing configurations from other papers e.g. [108] therefore this practice is used in order to mimic existing research

Default configuration differs library-to-library ([214] and [13] use LibSVM [31], whilst [159] uses Shogun [184]) and also with library versions, these versions are not reported so it is difficult to precisely replicate previous configurations.

Random Forest

The random forest algorithm (RF) is an ensemble algorithm, meaning that the final decision is determined by combining the ‘votes’ of other sub-classifiers. Ho first proposed random decision forests in 1995 [83] to overcome the poor performance of decision tree classifiers in modelling complex functions.

In these experiments, the Scikit-learn python library [23, 148] is used to implement a random forest rather than XGBoost since random forests are more commonly used by malware detection researchers. The aim of these experiments is to recreate some of the common practices used for dynamic malware detection. XGBoost may yield better results but that is not the aim of this chapter (it is used in the next chapter).

3.5 A Machine Learning Model from Laboratory to Practice

It is difficult for security professionals to predict how well a model will perform in their own environment that was trained on external data. The aim of this chapter is to see how robust popular data sources and algorithms are when tested on data from two different provenances.

3.5.1 Samples

The samples collected are summarised in Table 3.2 across the four filetypes.

The training data is collected from publicly available sources. The malware is obtained from VirusShare [201] whilst the benign document files were collected from the GovDocs repository [41]. The benign executables are a combination of executables from a fresh Microsoft Windows install and from a free online software provider,

Filetype	Malicious		Benign	
	Provenance	Total Samples	Provenance	Total Samples
EXE	VirusShare	938	Win7 SysFiles and PortableApps	1047
PDF	VirusShare	1014	GovDocs	1012
DOC	VirusShare	1004	GovDocs	1000
XLS	VirusShare	805	GovDocs	999
EXE	Org.	845	Org.	872
PDF	Org.	2088	Org.	2011
DOC	Org.	92	Org.	52
XLS	Org.	23	Org.	330

Table 3.2: Dataset provenance by filetype. Org. = Commercial data provided by the organisation. Benign publicly available samples validated by checking for detections using VirusTotal [202]

PortableApps [78]. VirusTotal's online malware analysis platform was used as a proxy to check if any of the PortableApps were in fact malicious. Data collection was an iterative process until 1000 files of each type, benign and malicious were collected. It was not possible to request a specific number of samples from the organisation and so some datasets are smaller than others. An organisation-provided dataset was supplied by the Secure Operations Centre of a commercial entity and the imbalance and number of samples is representative of the file-formats seen during a fixed period. Henceforth the publicly available samples will be referred to as the 'public dataset' and the organisation-provided samples will be referred to as the 'commercial dataset'.

To improve the statistical significance of the results, 10-fold cross-validation is used. The public dataset is split into 10 subsets, with equal proportions of malicious and benign samples in each (stratified k-fold). 9 subsets are combined and used to train a model with one subset held out for testing, this is then repeated for the other 9 possible combinations of subsets. This results in 10 trained models, which are then tested with the respective held-out 10th of data from the public data and then with the entire organisation test set.

Cross-validation is not best practice for security applications due to the changing nature of software, especially malware, over time [150]. However, in these experiments, the accuracy of the model *over time* is not the central concern but rather the accuracy of the model on data collected in a different environment. Subsequent chapters in this thesis enforce date-splitting for evaluation.

3.5.2 Sparsity

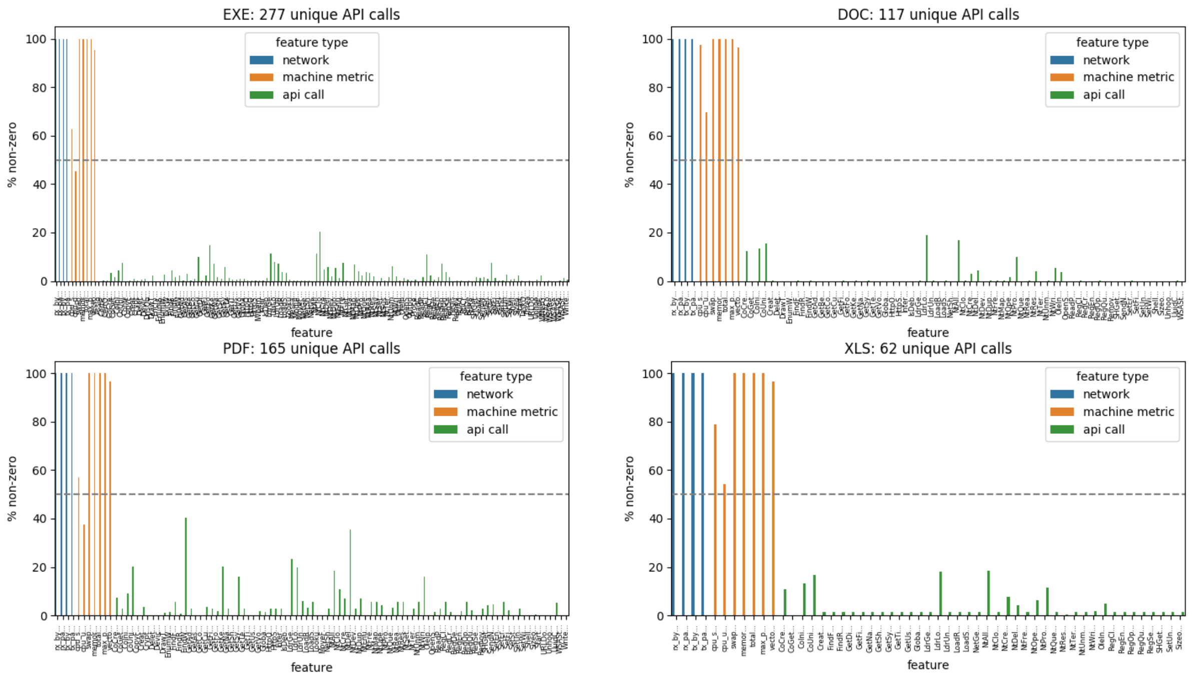


Figure 3.2: Percentage of instances with non-zero network data, machine metrics and API calls by filetypes in the public dataset. Grey dashed line denotes 50% mark, the threshold for sparsity.

There are between 62 and 277 unique API calls depending on the filetype (public dataset only[1]). A brief look at the three data categories across the four filetypes reveals that API calls are sparse features whereas network and machine metrics are not. A sparse feature vector is one for which most features occur less than half the time, a dense input vector is the inverse. Figure 3.2 shows the percentage of instances for which API calls hold a value of 0 (within a second-long snapshot for a fair comparison with machine metrics) vs machine and network activity metrics (excluding the time since execution started since this value is never 0). Every API call is sparse in the time series

[1]for security reasons the organisation data is not analysed at the request of the commercial entity providing the data

public training data, in that more than 50% of the time it is not being called. Any API call not seen during training will mean nothing to the model. None of the network metrics are sparse and for machine metrics, only the CPU user consumption percentage is sparse: 45.2% for EXE 37.6% for PDF.

Hypothesis on Sparsity and Robustness

At this point, it was possible to make a hypothesis that the API calls will not perform as robustly on the commercial test set as the numeric continuous data.

In 2011, Xu et al. [216] proved that there is a trade-off between algorithm stability and model sparsity for l1-regularised (lasso) and l2-regularised (ridge) regressions. The research defined sparse algorithms rather than sparse input feature sets and defines sparse algorithms as those able to ‘identify redundant features’ i.e. to replace two features with one where they are well-correlated. Stable algorithms are those which have a uniform loss in relation to the order of samples presented during learning, especially as the number of training samples increases. Though in this case, it is the input features that are sparse and not necessarily the model, extending Xu’s proof, it could be that the sparsity of input features decreases model robustness.

3.5.3 Data Collection

API calls, machine and network metrics are collected whilst software executes in a Windows 7 virtual machine. Cuckoo Sandbox [77] was used to manage restoring the VM to its original state between samples and to manage the manner in which samples are executed. The Cuckoo Sandbox API hooking module was used to collect API call features, and the Psutil [63] python library was used to collect machine metrics.

Microsoft Office (v.2010) and Adobe Acrobat (v.9) were installed for the XLS, DOC and PDF files to run. Older, more vulnerable, versions of Office and Acrobat are used so that the malicious behaviours which may rely on older vulnerabilities are more likely to be elicited[2].

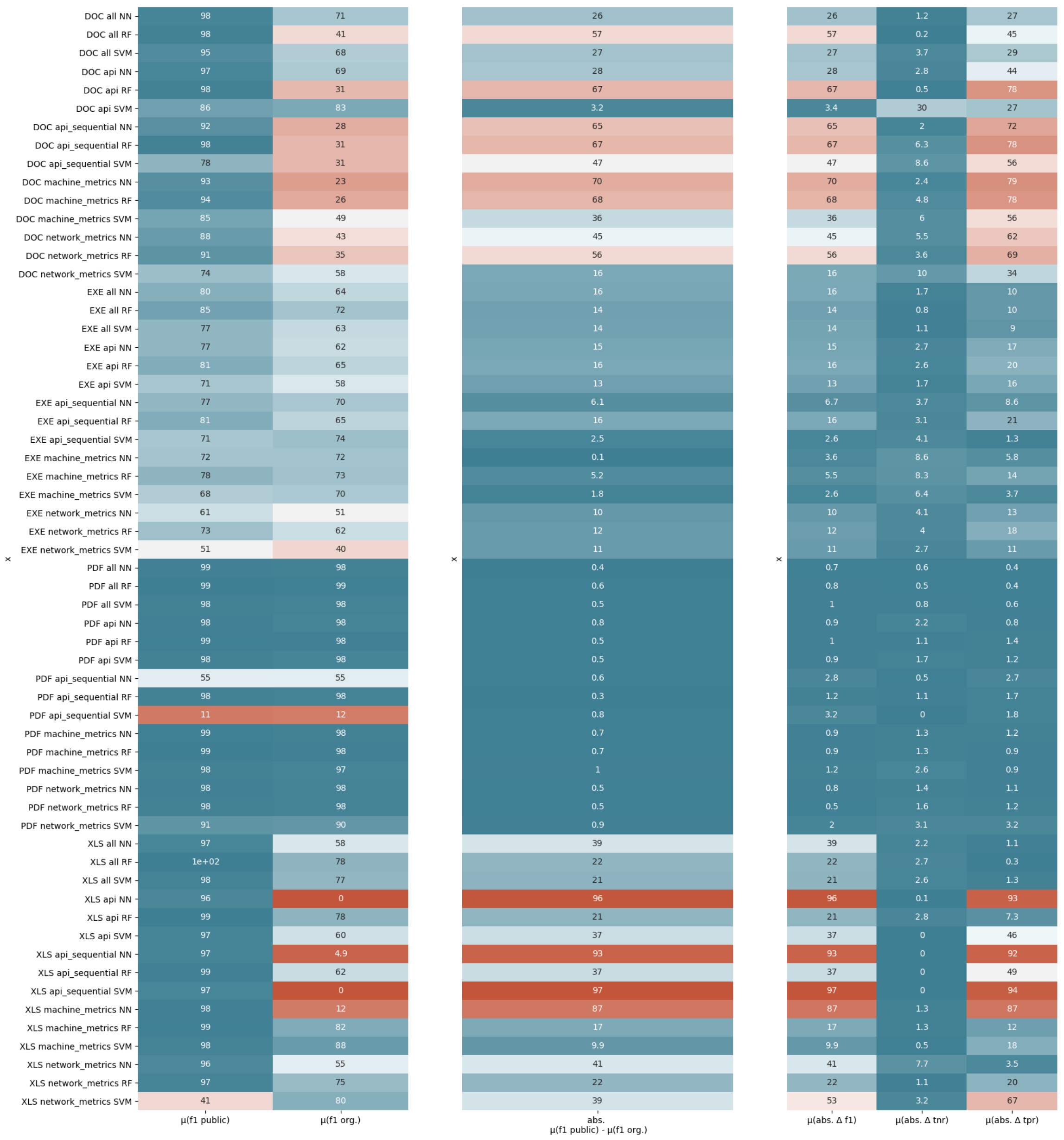
3.5.4 A Note on Time Series Data with Non-Time Series Algorithms

Note that none of Random Forests, SVMs or feed-forward Neural Networks (algorithms used in this chapter) are directly capable of processing time series. The time-series vectors could be flattened across time to create a vector that is *number of features*x*time steps* long but since the time series length is variable with this data (not all samples execute for the full time); for the time series API calls and machine metrics, the models are trained on the data collected in one-second intervals the mean prediction across all time points is used; this can be considered majority voting across all time points. Chapter 4 addresses time series models and dynamic data, but the goal of this chapter was to use the most prevalent algorithms from previous work.

3.6 Contextual Robustness Results

The heatmap in Figure 3.3 reports the mean F1-scores for the different file types, algorithms and input features on the public test sets generated through 10-fold cross-validation; this data is available in plain text in the appendix (Table 1). F1-scores are used due to the highly imbalanced malware/benignware ratios for some filetypes. The table also indicates the mean absolute change in F1-score, true positive rate (TPR) and true negative rate (TNR) for the 10 models in each category. These latter metrics are intended to give a sense of the change in metrics from one dataset to the other. In these experiments, the public test sets represent the unseen test sets used by researchers in the lab. An organisation adopting a research model that scored a particular F1-score in the lab may expect some small deviation in accuracy in practice, perhaps 2 to 3 percentage points deviance from the accuracy reported, the change ranges ($\triangle$ range)

[2]This is not to say that the number of vulnerabilities in software necessarily decreases as version numbers increase



x	μ(f1 public)	μ(f1 org.)	abs. μ(f1 public) - μ(f1 org.)	μ(abs. Δ f1)	μ(abs. Δ tnr)	μ(abs. Δ tpr)
DOC all NN	98	71	26	26	1.2	27
DOC all RF	98	41	57	57	0.2	45
DOC all SVM	95	68	27	27	3.7	29
DOC api NN	97	69	28	28	2.8	44
DOC api RF	98	31	67	67	0.5	78
DOC api SVM	86	83	3.2	3.4	30	27
DOC api_sequential NN	92	28	65	65	2	72
DOC api_sequential RF	98	31	67	67	6.3	78
DOC api_sequential SVM	78	31	47	47	8.6	56
DOC machine_metrics NN	93	23	70	70	2.4	79
DOC machine_metrics RF	94	26	68	68	4.8	78
DOC machine_metrics SVM	85	49	36	36	6	56
DOC network_metrics NN	88	43	45	45	5.5	62
DOC network_metrics RF	91	35	56	56	3.6	69
DOC network_metrics SVM	74	58	16	16	10	34
EXE all NN	80	64	16	16	1.7	10
EXE all RF	85	72	14	14	0.8	10
EXE all SVM	77	63	14	14	1.1	9
EXE api NN	77	62	15	15	2.7	17
EXE api RF	81	65	16	16	2.6	20
EXE api SVM	71	58	13	13	1.7	16
EXE api_sequential NN	77	70	6.1	6.7	3.7	8.6
EXE api_sequential RF	81	65	16	16	3.1	21
EXE api_sequential SVM	71	74	2.5	2.6	4.1	1.3
EXE machine_metrics NN	72	72	0.1	3.6	8.6	5.8
EXE machine_metrics RF	78	73	5.2	5.5	8.3	14
EXE machine_metrics SVM	68	70	1.8	2.6	6.4	3.7
EXE network_metrics NN	61	51	10	10	4.1	13
EXE network_metrics RF	73	62	12	12	4	18
EXE network_metrics SVM	51	40	11	11	2.7	11
PDF all NN	99	98	0.4	0.7	0.6	0.4
PDF all RF	99	99	0.6	0.8	0.5	0.4
PDF all SVM	98	98	0.5	1	0.8	0.6
PDF api NN	98	98	0.8	0.9	2.2	0.8
PDF api RF	99	98	0.5	1	1.1	1.4
PDF api SVM	98	98	0.5	0.9	1.7	1.2
PDF api_sequential NN	55	55	0.6	2.8	0.5	2.7
PDF api_sequential RF	98	98	0.3	1.2	1.1	1.7
PDF api_sequential SVM	11	12	0.8	3.2	0	1.8
PDF machine_metrics NN	99	98	0.7	0.9	1.3	1.2
PDF machine_metrics RF	99	98	0.7	0.9	1.3	0.9
PDF machine_metrics SVM	98	97	1	1.2	2.6	0.9
PDF network_metrics NN	98	98	0.5	0.8	1.4	1.1
PDF network_metrics RF	98	98	0.5	0.5	1.6	1.2
PDF network_metrics SVM	91	90	0.9	2	3.1	3.2
XLS all NN	97	58	39	39	2.2	1.1
XLS all RF	1e+02	78	22	22	2.7	0.3
XLS all SVM	98	77	21	21	2.6	1.3
XLS api NN	96	0	96	96	0.1	93
XLS api RF	99	78	21	21	2.8	7.3
XLS api SVM	97	60	37	37	0	46
XLS api_sequential NN	97	4.9	93	93	0	92
XLS api_sequential RF	99	62	37	37	0	49
XLS api_sequential SVM	97	0	97	97	0	94
XLS machine_metrics NN	98	12	87	87	1.3	87
XLS machine_metrics RF	99	82	17	17	1.3	12
XLS machine_metrics SVM	98	88	9.9	9.9	0.5	18
XLS network_metrics NN	96	55	41	41	7.7	3.5
XLS network_metrics RF	97	75	22	22	1.1	20
XLS network_metrics SVM	41	80	39	53	3.2	67

Figure 3.3: Mean (μ) F1-scores for withheld public test set, organisation test set, absolute difference between mean F1-scores, mean absolute difference between F1-scores per model, mean absolute difference in true positive rate (TPR) and true negative rate (TNR) per model. Evaluated using a public and commercial dataset for 4 filetypes, 3 algorithms, 6 input feature sets averaged over 10 models (10-fold cross-validation). **N.B.** Metrics have been multiplied by 100 for easier comparison with percentages

reported in the table indicate that in many cases the models fail to meet this criterion. The colour scale is inverted for the right-hand side so that dark green represents the most desirable scores across the heatmap.

Figures 3.4, 3.6, 3.7, and 3.8 display the EXE, PDF, DOC and XLS ‘lab’ F1-score, TPR and FNR along the x-axis and the corresponding metric for the organisation test set on the y-axis. A 45-degree line indicates where the data should fall if lab results were perfectly reflected in the organisation test set. Points below this line indicate where the lab results were better than the organisation results, it would be reasonable to hypothesis that the accuracy metrics will fall as the underlying distribution of the lab test data is likely to be more similar to the training data than the organisation data.

In the following sections let N be the total number of samples.

3.6.1 EXE

For the ‘lab’ dataset, executables (EXE) see the lowest F1-scores of all 4 filetypes. This is expected because the possible set of behaviours for EXEs is broader than for the other types, which are all channelled through the same application initially (e.g. Adobe Acrobat, Microsoft Word, Microsoft Excel).

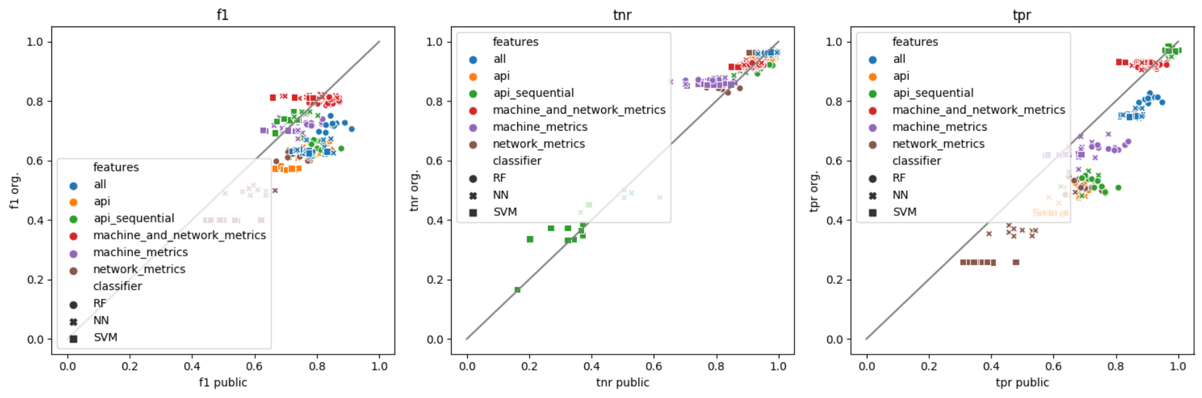


Figure 3.4: F1-score, TPR, FNR on public test sets and organisation test set for EXE files with different feature sets and algorithms

The best-performing algorithm and feature set in the lab for EXE files was a combination of all features and a random forest. On testing with the organisational dataset, however, there is a significant drop in F1-score from 0.85 to 0.73, the most stable on average being the sequential API calls. Figure 3.4 shows that network data (brown) and frequency counts of API calls (orange) drift furthest from the F1-scores, TPRs and TNRs seen in the lab. Machine metrics (purple) retain F1-scores better than aggregating all the data together (blue), this is likely because the models including all features are using the API frequency data (green) and so seeing worse results on the organisation test set again.

In the lab, frequency of API calls performed equally well (mean F1: NN=77, RF=81, SVM=71) to sequential APIs (mean F1: NN=77, RF=81, SVM=71) but the frequency representation sees a much sharper fall in mean F1-scores than the sequential representation. Machine metrics are most stable looking at the difference between means of F1-scores $\frac{1}{N}\sum_{i=0}^{N} F1_i^{lab} - \frac{1}{N}\sum_{i=0}^{N} F1_i^{org.}$ but API calls perform worse than sequential APIs for mean of the absolute changes per model: $\frac{1}{N}\sum_i = 0^N(F1_i^{lab} - F1_i^{org.})$. Although these non-time series algorithms do not know that the sequential API call data are related to one another, one may hypothesise that the separation of calls into small segments of time may give a better indication of precise malicious behaviour e.g. a particular sequence of APIs is linked to process injection or some other common malicious behaviour. This is a possible avenue for future work.

Of the feature sets, the machine metrics see the smallest divergence from the lab F1-scores, averaging across all algorithms; the absolute change in mean F1-score for the EXE models is lowest when trained using machine metrics (0.1 percentage points for NN, 5.2 percentage points for RF, 1.8 percentage points for SVM) (See Fig.3.3).

Though the machine metrics may be considered the most reliable data across these two sources, the mean F1 is only 78/100, compared to 85/100 when using all features. Though a researcher may choose robustness over accuracy, due to the low F1-score for this data, additional experiments were conducted combining feature sets for the executable files only. The feature sets are as follows:

- **sequential network data [4 features]:** network data may be polled at intervals rather than just counted at the end, similarly to how the API calls have been used sequentially above
- **sequential machine metrics and API calls [7 + 277 = 284 features]**
- **sequential API calls and network data [277 + 4 = 281 features]**
- **sequential machine metrics and network data [7 + 4 = 11 features]**

Figure 3.5 shows the same metrics as previously reported for EXE files using the old and new feature sets. The sequential network metrics perform a little better (with NN and RF) than taking the total metrics for the whole execution trace. Machine metrics with sequential API calls see a bigger drop in F1-scores than for either feature set used in isolation. Combining API calls with network data has a lower or equivalent mean F1 to sequential API calls alone on both the public and organisation test sets. Machine metrics coupled with network data however not only achieve higher F1-scores than most other models (mean F1s: 81, 84, 78) but these scores remain within 4 percentage points in testing on the organisational test set (mean F1s: 81, 80, 81).

The machine metrics and network data is plotted against the original feature sets in Figure 3.4 in red (other feature sets omitted to enable easier reading of the graph). This feature set produced the best performing and most robust model for EXE detection, scoring a mean F1-score 0.81 using a neural network in the lab and also a mean of 81 on the commercial dataset.

3.6.2 PDF

As is clear from the heatmap and Figure 3.6 almost all models achieved very high accuracy distinguishing malicious and benign PDFs. Those models which performed poorly performed equally poorly on the organisation dataset. Interestingly, for PDF files the sequential API data (green) is the worst-performing feature set though it was

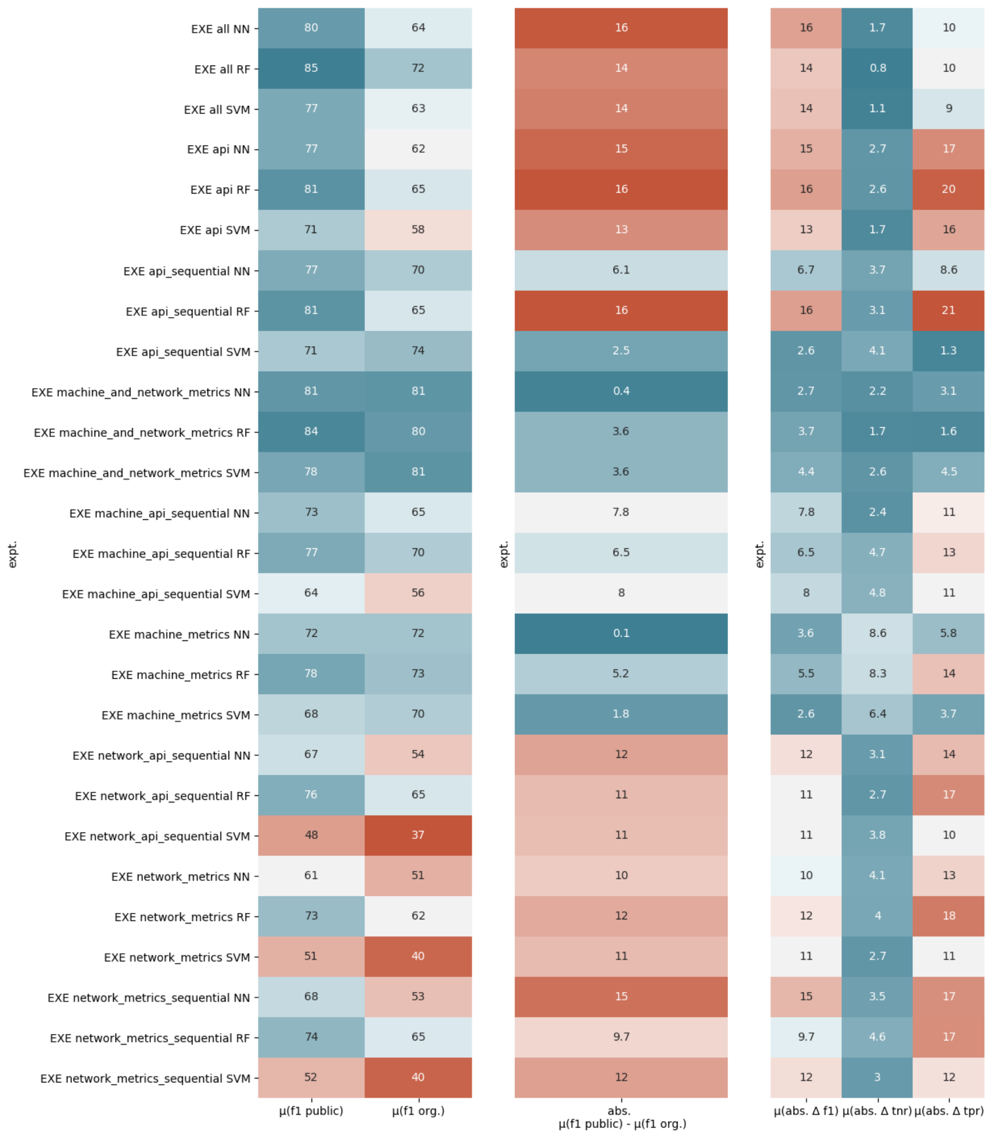


Figure 3.5: Mean μ Executable classification F1-scores for public and organisation test set as well as difference metrics for additional different feature sets.

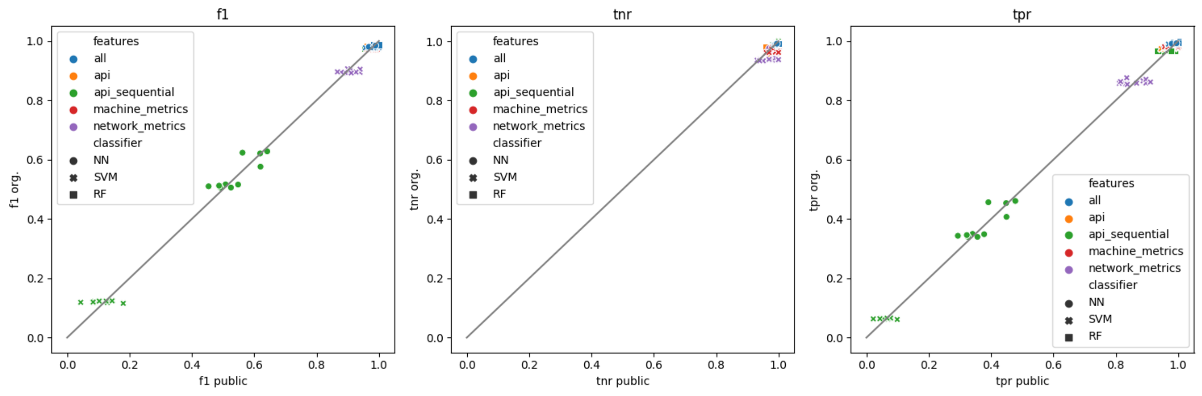


Figure 3.6: F1-score, TPR, FNR on public test sets and organisation test set for PDF files with different feature sets and algorithms

arguably the best for executables. When used to train a random forest these features seemed to perform well but failed to produce any reasonable models using an SVM or NN. Combining feature sets for PDF files did not improve classification metrics.

3.6.3 DOC

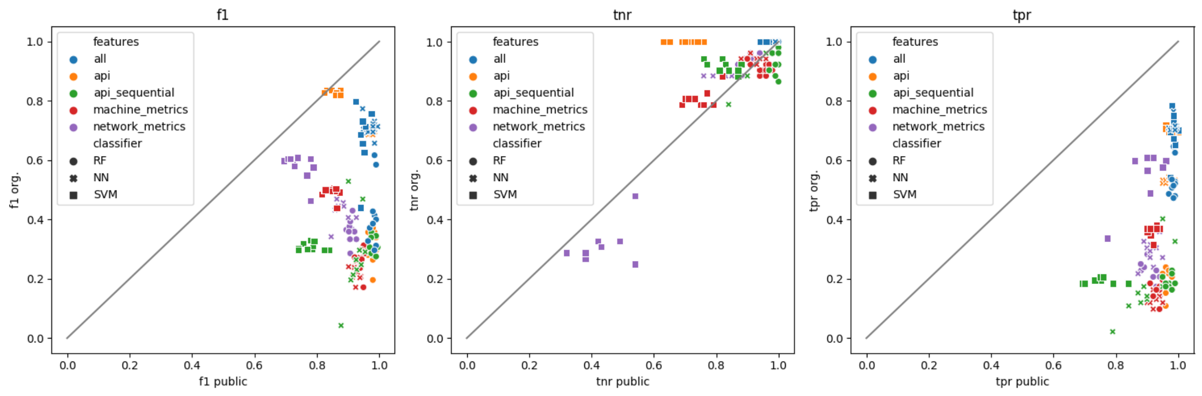


Figure 3.7: F1-score, TPR, FNR on public test sets and organisation test set for DOC files with different feature sets and algorithms

The organisation test set for Microsoft Word documents (DOC) was imbalanced. Although models using all features together were able to achieve high detection accuracy in the lab, it was not replicated for the organisation data. It may seem that the SVM performed well but from Figure 3.7 it is evident the F1-score only appears stable because the dramatic loss in TPR is balanced by a dramatic increase in TNR, which is not desirable for an end-user believing that the model they are using is prone to false positives when in fact in its new environment it is now prone to false negatives. It is difficult to draw any conclusions about the diversity of DOC malware or benignware due to the very small organisation test set. Combining feature sets for DOC files did not improve classification metrics.

3.6.4 XLS

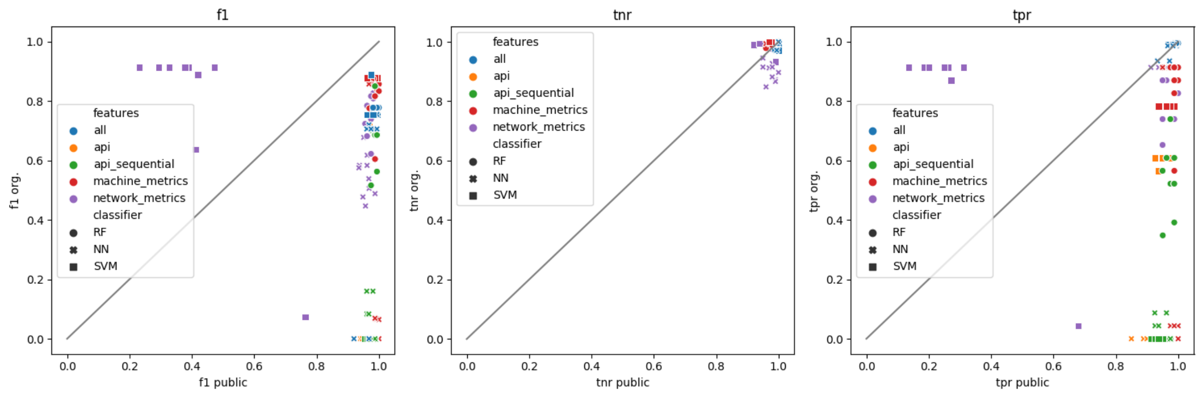


Figure 3.8: F1-score, TPR, FNR on public test sets and organisation test set for XLS files with different feature sets and algorithms

The Microsoft Excel spreadsheets (XLS) organisation-provided test set is also highly imbalanced. Again, Figure 3.7 shows that there were a few models able to achieve high F1-scores on both the lab and enterprise datasets but even the best model sees a nearly 10% reduction in the lab test results. For XLS there is no dominant feature set and neither are the results clustered by feature set (colour) as they are for EXEs and PDFs but neural networks performed particularly badly here. Combining feature sets for XLS files did not improve classification metrics.

3.6.5 Discussion: EXE Results

Given the small and imbalanced datasets of the DOC and XLS commercial test sets, it is difficult to conclude that one model or set of features has performed best. The PDF classifier achieves very high and consistent accuracy metrics on the test set irrespective of the algorithm or feature set. The EXE is more challenging to classify, even on the public test sets taken from the same distribution as the training data.

As reported in table 2.1 above, API calls are the most commonly used feature and the frequency of API calls are the most common representation of this data (Table 3.1 above). For security reasons, it is not possible to print the distribution of API calls from the commercial dataset but the high-level metrics show significant difference from the public dataset. The public data EXE files saw 277 unique API calls and the commercial dataset produced 263. 57 of the API calls in the commercial dataset did not appear anywhere in the public dataset, these API calls, therefore, failed to provide information to the models since the models had no context for their meaning. There were also 63 in the public dataset that did not appear in the commercial set, indicating that the software sets were drawn from significantly different distributions. More concretely, analysing the machine metrics, a 2-distribution Kolmogorov-Smirnov (KS) test was used to measure the likelihood that the data were drawn from the same distribution. The KS-test produces a p-value, which if higher than 0.05 indicates that the data are likely to have been drawn from the same distribution (at a 95% confidence interval). The highest p-value for any of the 10 machine metrics (time, since execution began, is excluded) is 1.6e-08. Such metrics may be used to indicate that the organisation data may see significantly different results to the evaluation data but this topic requires extensive additional research.

A further investigation could look at extending Xu et al.'s proof [216] to sparse input feature spaces rather than sparse models.

3.7 Limitations and Conclusions

The experiments in this chapter were intended as an analogy for the resilience of academic findings when tested in an operational context, addressing the following research question *RQ1: To what extent do dynamic data and algorithm choices impact the robustness of malware detection accuracy across different datasets?*.

Machine metrics were found to be more stable, especially when combined with network data, but it is only possible to conclude that this is the case for the organisation test set used. This may be indicative that as well as algorithmic sparsity, feature sparsity may lead to less stable models and machine metrics were a non-sparse feature that performed well (c.f. API calls and network traffic alone). However, these experiments *do* demonstrate the variance in model performance in different contexts when features are varied even when exactly the same samples are used and laboratory performance between two metrics seems comparable. Little correlation was found between algorithm types though random forests tended to see the larger than average divergence from lab metrics.

One limitation of machine metric models however is that they are dependent on the hardware upon which they run, unlike API calls. API call frequency may be impacted by hardware i.e. slower hardware = fewer calls are made. Machine metrics may be even more subject to change with different hardware configuration; CPU usage is expressed as a percentage for example and the percentage of total processing power used will depend on how many cores and how fast the processor is. However, this is not an insurmountable problem, a virtual machine is used to collect the data for which the resource capabilities can be specified, though the underlying bare metal must meet the minimum requirements of the virtual machine.

These experimental results can only provide limited conclusions. The experiments themselves, despite more than 600 models being trained, only touch a fraction of the space that may be explored but they highlight the need for practitioners to have a means of evaluating algorithms within their own environment or perhaps training the models on data from their own ecosystem, this will be limited by the ease with which labelled data can be collected.

These preliminary results indicated that some of the most commonly used dynamic data sources for classifying executables may outperform others in testing but may not stand up using another dataset. This result constitutes the following contribution: *C1: A demonstration that the most accurate type of dynamic data in laboratory testing may not be the best-performing or most robust data type when tested on data of a different provenance.*.

The next chapters will build on the tentative conclusion that machine metrics with network data are a robust feature to use for dynamic malware detection.

Chapter 4

Early Stage Prediction

This chapter addresses *RQ2: Is it possible to predict that software is malicious from its early stages of execution?* and presents results showing the trade-off between time into execution and classification accuracy. Dynamic data is favoured over static data due to the difficulty of obfuscation by comparison with static techniques [47, 75, 108, 220]. This is the first attempt to see if the maliciousness of malware can be predicted based on its early execution. The results indicate that it is possible to accurately classify 94% of samples within the first 5 seconds of execution; this presents a significant reduction in analysis time by comparison with existing works for a relatively small reduction in accuracy. This work was first published in [156].

The following section presents the case for early stage detection (4.1). Section 4.2 outlines the methodology, followed by experimental results (4.3. Finally future work (4.4 and conclusions(4.5) are discussed.

4.1 The Case for Early Stage Prediction

The previous chapter makes the case for automatic malware detection based on the rate of new malware. Automatic rule generation using machine learning is necessary as malware detection models require updating to capture information about new malicious behaviours. A key limitation of dynamic behavioural data being that it takes longer to collect data than static techniques, and therefore may not be preferred for endpoint detection.

In order to address this shortcoming of dynamic analysis, this chapter proposes exploring the relationship between classification accuracy and data collection time. If the data collection time can be sufficiently reduced, this may enable dynamic malware detection to be incorporated into an end-point solution without greatly compromising usability, and perhaps by extension adoption and use. Wider adoption of dynamic analysis for endpoint detection could help to mitigate some of the aforementioned weaknesses of static analysis, prevent more malware from successfully executing and thus reduce the negative impacts of cyber attacks.

Malicious and benign files comprise a wide range of software and potential behaviours, but this chapter hypothesises that malicious activity begins rapidly once a malicious file begins execution because this reduces the overall run-time of the file and thus the window of opportunity for being disrupted by a detection system, analyst, or technical failure.

The previous chapter argues that non-sparse representations of software behaviour may mitigate the reduction in model performance caused by changing distributions in that behaviour. The most popular behavioural features were compared with the most popular machine learning algorithms. Following this analysis machine activity data is used in this chapter which includes experiments with time-series models.

The results in this chapter find that an ensemble of recurrent neural networks is able to classify 94% of samples correctly within the first 5 seconds and 96% within the first 20 seconds. Previous dynamic analysis research collects data for around 5 minutes per sample. 5 seconds is not the goal to reach but rather the time at which accuracy made a significant increase and the differential of accuracy over data recording time begins to decrease and plateau. It is possible that additional data could be captured after the first 20 seconds and before 5 minutes but the detection accuracy has already appeared to plateau by this point in time.

Testing the robustness of this model on new malware families, further experiments omit entire malware families from the training set and test model accuracy with promising results. Notably a case study using 3,000 ransomware samples achieves 94% detection accuracy after just 1 second when ransomware is omitted from the training set and 99% when it is included in the training set. The motivation for removing ransomware samples, which is carried out using labels assigned by virus total is in order to mimic the advent of a new malware coming into existence, this experiment is imperfect however since it does not also split by time, meaning that some of the ransomware being tested may overlap with temporally popular malware techniques. Half of the ransomware is then included in a second training of the model in order to see if exposure to this type of malware helps the model, which is does with more than 99 in 100 samples being detected.

4.1.1 Dynamic Malware Detection: Time Series Algorithms and Time-saving Approaches

The previous chapter outlines the state of the art for malware detection using machine learning techniques. Whilst static models are capable of achieving high detection accuracy, these results may not be robust over time [168] and have been demonstrated to be easily fooled by small manipulations to code [75, 108].

Time Series Algorithms for Malware Detection

Methods using dynamic data assume that malware must enact the behaviours necessary to achieve their aims. Typically, these approaches capture behaviours such as API calls to the operating system kernel. The models used vary, but due to the time-series nature of dynamic data some researchers have used Hidden Markov Models which assign probabilities to sequences of discrete states [47, 90].

Some previous work has used recurrent neural networks (RNNs) for malware detection. Tobiyama et al.[196] use RNNs to extract features from 5 minutes of API call log sequences which are then fed into a convolutional neural network to obtain 0.96 AUC score with a dataset of 170 samples. Kolosnjaji et al [106] sought to detect malware families with deep neural networks, including recurrent networks, to classify malware into families using API call sequences. By combining a convolutional neural network with long-short-term memory (LSTM) cells, the authors were able to attain a recall of 89.4%, but do not address the binary classification problem of distinguishing malware from benignware. Pascanu et al. [145] did conduct experiments into whether files were malicious or benign using RNNs and Echo State Networks. The authors found that Echo State Networks performed better with an accuracy of around 95% (error rate of 5%) but did not attempt to predict malicious behaviour from initial execution.

Reducing Behavioural Analysis Time

Existing methods to reduce dynamic data recording time have focused on efficiency. The core concept is only to record dynamic data if it will improve accuracy, either by omitting some files from dynamic data collection or by stopping data collection early. Shibahara et al. [177] decide when to stop analysis for each sample based on changes in network communication, reducing the total time taken by 67% against a method that analyses samples for 15 minutes each - thus reducing analysis to an average of just under 5 minutes per sample. The goal of [177] is to optimise analysis time in order to collect malicious network behaviour and not waste time when no network behaviour is present; the authors do not test detection accuracy for malicious/benign software thus the goal is not comparable to the aim of this chapter; furthermore samples execute for up to 30 minutes which is not a competitive time frame with static analysis methods. Neugschwandtner et al. [136] used static data to determine dissimilarity to known malware variants using a clustering algorithm. If the sample is sufficiently unlike any seen before, dynamic analysis is carried out. This approach demonstrated an improvement in classification accuracy by comparison with randomly selecting which files to dynamically analyse, or selecting based on sample diversity. Similarly, Bayer et al. [16] create behavioural profiles to try and identify polymorphic variants of known malware, reducing the number of files undergoing full dynamic analysis by 25%. These approaches still allow some files to be run for a long dynamic execution time, whereas this chapter investigates a blanket cut-off of dynamic analysis for all samples, with a view to this analysis being run in an endpoint anti-virus engine.

4.2 Methodology: Recurrent Neural Networks for Early Stage Malware Prediction

In order to advance malware detection to a more predictive model that can respond in seconds this chapter proposes a model which uses only short sequences of the initial dynamic data to investigate whether this is sufficient to judge a file as malicious with a high degree of accuracy. As Burnap et al. [26] argue 'malware cannot avoid leaving a behavioural footprint' of machine activity.

4.2.1 Features

Following the previous chapter, this chapter uses 10 machine activity data metrics as feature inputs to the detection model. These metrics are captured in a snapshot every second for the first 20 seconds whilst the sample executes starting at 0s, such that at 1s, there are two feature sets or a sequence length of 2. The metrics captured were: system CPU usage, user CPU use, packets sent, packets received, bytes sent, bytes received, memory use, swap use, the total number of processes currently running and the maximum process ID assigned.

As illustrated in Figure 4.1, activity data are collected during the execution of Portable Executable (PE) samples using Cuckoo Sandbox [77], a virtualised sandboxing tool.

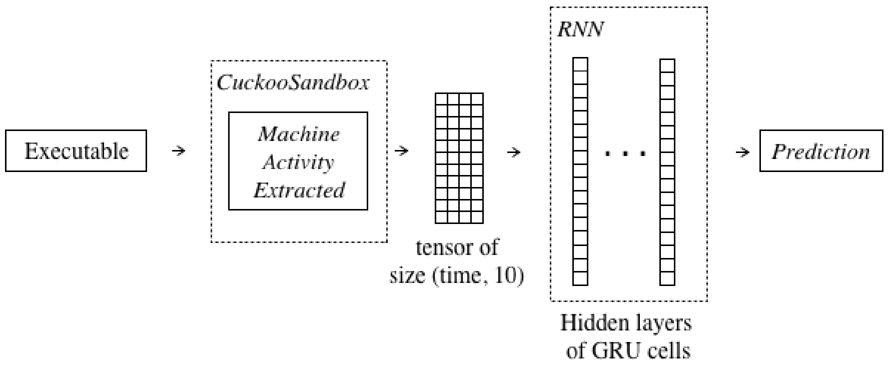

Executable
CuckooSandbox
Machine Activity Extracted
tensor of size (time, 10)
RNN
. . .
Hidden layers of GRU cells
Prediction

Figure 4.1: High-level model overview

While executing each sample the machine activity metrics are extracted using a custom auxiliary module reliant on the Python Psutil library [63].

4.2.2 Algorithm Choice

As the data are sequential, time series models can be used to extract information about the way in which data changes over time as well as the raw values. Making use of the time-series data means that the rate and direction of change in features as well as the raw values themselves are all inputs to the model. As noted in the previous chapter, neural networks are the most popular algorithm for machine learning tasks in malware detection and are the winning algorithm used in most machine learning competitions. This chapter proposes using RNNs for predicting malicious activity as as they are able to process time-series data. Moreover, they can process continuous numeric time series data such as machine metrics thus capturing information about change over time as well as the raw input feature values. Hidden Markov Models are not suited to highly variate numeric data due to the requirement for discrete state spaces [116], therefore some researchers have used RNNs which are able to model a large possible universe of states and memory over an extended chain of events [116].

RNNs can create temporal depth in the same way that neural networks are deep when multiple hidden layers are used. Until the development of the Long Short-Term

Memory Networks (LSTM) cell by Hochreiter and Schmidhuber in 1997, RNNs performed poorly in classifying long sequences, as the updates required to tune the weights between neurons would tend to vanish or explode [19].

In a feed forward neural network the activation of a given neuron is calculated using a weight matrix, W, denoting the relationship between the incoming neurons from the previous layer, the sum of the previous layer's activation functions, x a constant bias term, b, and an activation function g, often a nonlinear function such as a sigmoid or a rectified linear function. The activation a of a neuron i is therefore

$$a_i = g(W \cdot x + b)$$

. The weight matrix is updated using a backpropagation algorithm [97, 115], which requires g to be differentiable as the update vector is calculated using the chain rule to calculate the overall weight change using the partial derivatives of the model error with respect to the model weights in each layer.

Recurrent neural networks (RNNs) capture dependencies over T timesteps using historic inputs $X = x_0, x_1 ... x_T$. An additional weight matrix, U is trained to denote the relationship between timesteps such that the activation of a neuron at time t is

$$a_{i=t} = g(W \cdot x_t + U \cdot a_{i=t-1} + b)$$

The backpropagation algorithm must be amended in order to handle these temporal dependencies. The backpropagation through time (BPTT) algorithm, formulated by multiple researchers independently, must calculate the error derivative at each time step and then aggregate the errors; thus in BPTT, the network is often said to be 'unrolled' in time. If g is a 'squashing' function like a sigmoid, so-named because it rescales (squashes) any input to a range between 0 and 1. When the network is 'rolled' back up again the partial differentials are multiplied using the chain rule; T differentials are multiplied, each one reducing the differential further. In aggregation this can cause the weight update to be ‘vanishingly small’ or effectively zero. If using non-squashing functions the inverse problem can occur by which gradients become huge or ‘explode’. Several solutions have been proposed to this problem such as gradient clipping (for exploding gradients only) or regularisation of the weight values. LSTM networks address both the vanishing and exploding gradient problem in RNNs.

LSTM cells use gating mechanisms Γ to mitigate this problem to control the memory held within a cell. Each gate has its own weight matrices and biases W, U, b. The LSTM uses four gates with these respective roles: input, forget and output. These gates can all be represented using the generic RNN activation calculation where g is a sigmoid function, i.e.

$$\Gamma_{gate} = \sigma(W_{gate} \cdot x_t + U_{gate} \cdot a_{i=t-1} + b_{gate})$$

The gates’ outputs are combined to give the cell activation

$$a_{i=t} = \Gamma_{output} tanh(\Gamma_{forget} a_{i=t-1} + \Gamma_{input} c_{new})$$

where c_{new} is the new memory content of the cell, calculated by $tanh(W_{gate} \cdot x_t + U_{gate} \cdot a_{i=t-1})$. These gating mechanisms allow the LSTM to store historical inputs either by bypassing intermediate segments of the time series or adding the new memory data to existing data thus mitigating the problems surrounding weight updates. The success of LSTM has prompted a number of variants, though few of these have significantly improved on the classification abilities of the original model [74].

Gated Recurrent Units (GRUs) [35], however, have been shown to have comparable classification to LSTM cells, and in some instances can be faster to train [37]. For this potential training speed advantage, this chapter uses GRU units. GRU units are similar to LSTM units with two key differences, which are explained in detail by Chung et al. [37]. In brief, an update gate combines the input and forget gate such that the forget gate activation of the LSTM is $1 - \Gamma_{input}$ and there is no output gate to modulate the exposure of the hidden memory state of the cell. This reduces the number of parameters in the model and thus the computational complexity of the model and training time.

An appropriate architecture and learning procedure of a neural network is integral to a successful model. These attributes are captured by hyperparameter settings, which are often hand-crafted. Due to the rapid evolution of malware, it is anticipated that the RNN should be re-trained regularly with newly discovered samples, thus the architecture may need to change too. As it needs to be carried out multiple times, this process should be automated. A random search of the hyperparameter space is used as it can easily be parallelised , it is trivial to implement, and has been found to be more efficient at finding good configurations than grid search [20]. The best-performing configuration on a 10-fold cross-validation over the training set is used, the hyperparameter search space and final configuration is detailed in Table 1 for reproducibiltiy.

Hyperparameter	Possible values	Best configuration
Depth	1, 2, 3	3
Bidirectional	True, False	True
Hidden neurons	1 – 500	74
Epochs	1 – 500	53
Dropout rate	0 – 0.5 (0.1 increments)	0.3
Weight regularisation	None, $l1$, $l2$, $l1$ and $l2$	$l2$
Bias regularisation	None, $l1$, $l2$, $l1$ and $l2$	None
Batch size	32, 64, 128, 256	64

Table 4.1: Possible hyperparameter values and the hyperparameters of the best-performing configuration on the training set

4.2.3 Data Collection

4.2.4 Samples

Initially, 1,000 malicious and 600 'trusted' Windows 7 executables were obtained from VirusTotal [152] along with 800 trusted samples from the system files of a fresh Windows 7 64-bit installation. Followed by a further 4,000 Windows 7 applications downloaded from popular free software sources, such as Softonic [183], PortableApps [78] and SourceForge [186]. The online download files are included to give a a better representation the typical workload of an anti-virus system than purely using Windows system files.

The VirusTotal API [152] was used as a proxy to label the downloaded software as benign or malicious. VirusTotal runs files through around 60 anti-virus engines and reports the number of engines that detected the file as malicious. Similar to [168], for malicious samples, the dataset omitted any files that were deemed malicious by less than 5 engines in the VirusTotal API as the labelling of these files is contentious. Files not labelled as malicious by any of the anti-virus engines were deemed 'trusted' as there is no evidence to suggest they are malware, these were considered to be benign software samples. This has the limitation of not detecting previously unseen malware but our samples are selected from an extended time period historically so it is likely that it would be reported as malware at some point in this period if it were in fact malicious.

The final dataset comprised 2,345 benign and 2,286 malicious samples, which is consistent with dataset sizes in this field of research e.g. ([61], [195], [223], [4], [47], [196], [90]). A further 2,876 ransomware samples obtained from the VirusShare online malware repository [201] were used for the ransomware case study in Section 4.3.4.

It was possible at the time to extract the date that VirusTotal had first observed each file together with the families and variants that each anti-virus engine classified the malware samples as. The dates that the files were first seen ranged from 2006 to 2017. The test and training set files were split according to the date first seen to mimic the arrival of completely new software. The training set only comprised samples first seen by VirusTotal before 11:15 on 10th October 2017 and the test set only samples after this date, which produced a test set of 500 samples (206 trusted and 316 malicious). This timestamp was chosen because it gave a reasonable number of both malicious and benign samples in the training and testing datasets, in line with the sample size in the existing literature.

Family	Total	APT	Ransomware
Trojan	1,382	0	76
Virus	407	20	56
Adware	180	0	51
Backdoor	123	7	0
Bot	76	0	0
Worm	24	0	0
Rootkit	11	0	0
Disputed	83	-	-
Total	**2,286**		

Table 4.2: Number of instances of different malware families in dataset

The malware infection mechanism, referred to here as the family, total counts are documented in Table 4.2. The 'disputed' class represents those malware for which a family could not be determined because the anti-virus engines did not produce a majority vote in favour of one type.

In addition to these majority-vote types, a search for advanced persistent threat malware (APTs) and ransomware in each category was carried out. This was because APTs are notoriously difficult for static engines to detect and the ransomware case-study in Section 4.3.4 required removal of all ransomware from the training set.

4.2.5 Data Distributions

Table 4.3 outlines the minimum and maximum values of the 10 inputs that were collected for malware and benignware respectively. The inter-quartile ranges of values are similar for some features (see Figure 4.2), however, the benign data sees a far greater number of outliers in RAM use (memory and swap) and packets being received. The malicious data has a large number of outliers in total number of processes, but the benign samples have outliers in the maximum assigned process ID, indicating that malicious files in this dataset try to carry out lots of longer processes simultaneously, whereas benign files will carry out a number of quick actions in succession. These experiments did not use outlier detection and removal as part of preprocessing, though future work could explore this and whether it improved model accuracy.

	Benign		Malicious	
	Min.	Max.	Min.	Max.
Total Processes	43	57	44	137
Max. Process ID	3,020	26,924	3,020	5,084
CPU User (%)	0	100	0	100
CPU System (%)	0	100	0	100
Memory Use (MB)	941	8,387	939	1,957
Swap Use (MB)	941	14,040	941	1,956
Packets Sent (000s)	0.3	110	0.3	129
Packets Received (000s)	2.9	737	2.9	192
Bytes Received (MB)	4	1,116	4	266
Bytes Sent (MB)	0.4	1,434	0.4	1,188

Table 4.3: Minimum and maximum values of each input feature for benign and malicious samples

4.2.6 Data Preprocessing

Prior to training and classification, the data is normalised to improve model convergence speed in training. By keeping data between 1 and -1, the model is able to converge more quickly, as the neurons within the network operate within this numeric range [112]. This is achieved by normalising around the zero mean and unit variance of the training data. This z-score normalisation is used instead of minimum-maximum scaling (which makes the minimum value scale to -1 and the maximum scale to 1) because this latter approach can cause outliers to skew the scaled distribution and reduce granularity. Z-scores allow outliers to exist outside the [-1,1] range and prevent the distribution of more common values being squeezed into a very small range. For each feature, i , the mean, μ_i, and variance, σ_i, of the training data are calculated. These values are stored, after which every feature, x_i is scaled:

$$\frac{x_i - \mu_i}{\sigma_i}$$

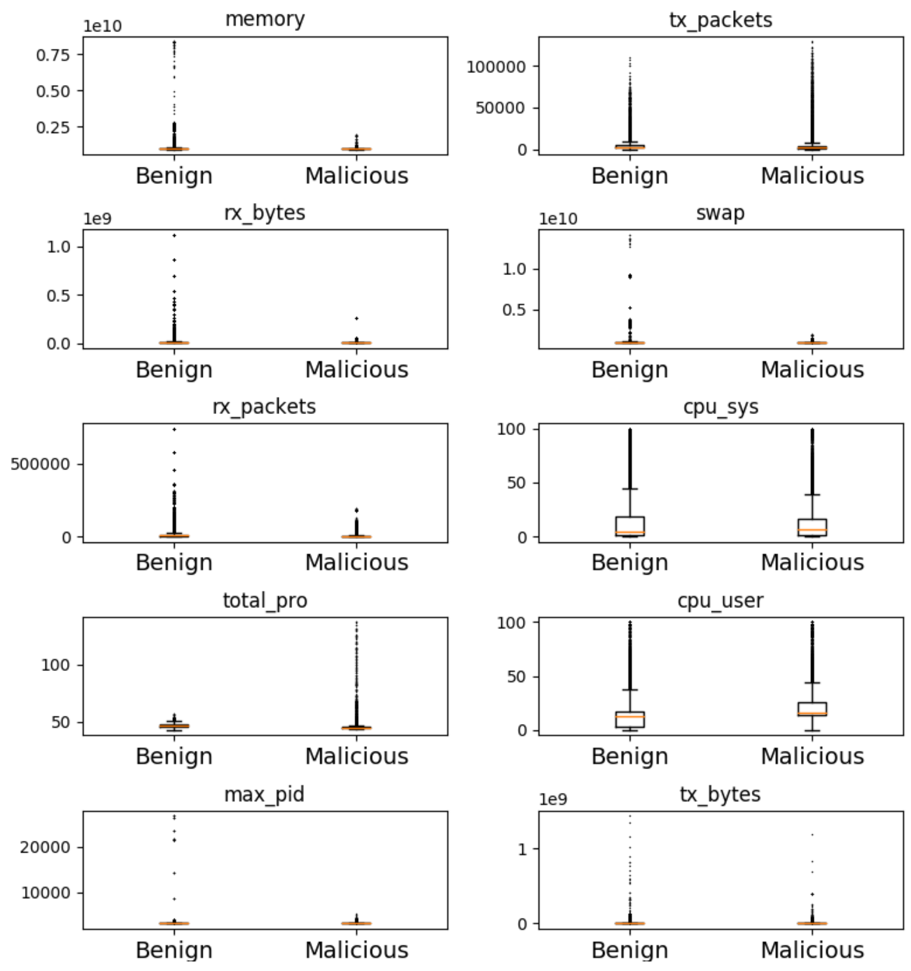


Figure 4.2: Box plots of input features for benign and malicious samples

4.3 Experimental Results

For reproducibility, the code used to implement the following experiments can be found at [155]. To implement the RNN experiments Keras [36] was used, whilst Scikit-Learn [148] was used to implement the other machine learning algorithms. The RNN training was accelerated using an Nvidia GTX1080 GPU. The Virtual Machine used for data collection used 8GB RAM, 25GB storage, and a single CPU core running 64-bit Windows 7. Python 2.7 was installed on the machine along with a free office software suite (LibreOffice), browser (Google Chrome) and PDF reader (Adobe Acrobat) in case of dropped files being in a different format. The virtual machine was restarted between each sample execution to ensure that malicious and benign files alike began from the same machine set-up.

4.3.1 Hyperparameter Configuration

Each layer of a neural network learns an abstracted representation of the data fed in from the previous layer. There must be a sufficient number of neurons in each layer and a sufficient number of layers to represent the distinctions between the output classes. The network can also learn to represent the training data too closely, causing the model to overfit. Choosing hyperparameters is about finding a nuanced, but generalisable representation of the data. Table 1 details the search space and final hyperparameters selected for the models in the later experiments. A maximum depth of 3 hidden layers was explored because previous work [87, 107] did not see significant increases in accuracy for malware detection with deeper models and an increased depth has a significant impact on training time. Although there are only 8 parameters to tune, there are 478 million different possible configurations[1]. As well as the hyperparameters above, the amount of data execution is considered a hyperparameter and is randomly selected. This is to allow for interaction between hyperparameters and the volume of data col-

[1]From Table multiply all number of options together: $3x2x499x499x5x4x4x4 = 478,081,920$

lected.

Although the goal is to find the best classifier for the shortest amount of time, selecting an arbitrary time such as 5 or 10 seconds into file execution may only produce models capable of high accuracy at that sequence length. It is not known whether a model will increase monotonically in accuracy with more data or peak at a particular time into the file execution. Randomising the time into execution used for training and classification reduces the chances of having a blinkered view of model capabilities.

Without regularisation measures, the representations learned by a neural network can fail to generalise well. Several techniques for regularisation are trialled: dropout as well as $l1$ and $l2$ regularisation on the weight and bias terms in the network in our search space. Dropout [187] randomly omits a pre-defined percentage of nodes each training epoch, which commonly limits overfitting. $l1$ regularisation penalises weights growing to large values whilst $l2$ regularisation allows a limited number of weights to grow to large values. The random search indicated that a dropout rate of 0.1-0.3 produced the best results on the training set, but weight regularisation was also prevalent in the best-performing configurations.

Bidirectional RNNs use two layers in every hidden layer, one processing the time series progressively, and the second processing it regressively. Pasacnu et al. [145] found good results using a bidirectional RNN, as the authors were concerned that the start of a file's processes may be forgotten by a progressive sequence as if the LSTM cell forgets it in favour of new data, the regressive sequence ensures that the initial data remains prevalent in decision-making. In the experiments reported in this chapter, many of the best-scoring configurations used a bidirectional architecture.

A model depth of 2 or 3 gave the best results. The number of hidden neurons was 50 or more in each layer to give any accuracy above 60%. All configurations used the 'Adam' weight updating rule [102] as it learns to adjust the rate at which weights are updated during training.

4.3.2 Predicting Malware Using Early-Stage Data

Our goal is examine the extent to which dynamic data collection time can be reduced whilst maintaining a reasonable level of detection accuracy. If the model is accurate within a short time, this sandbox-based analysis could be integrated into an endpoint antivirus system.

RNNs were tested against other machine learning algorithms used for behavioural malware classification: Random Forest, J48 Decision Tree, Gradient Boosted Decision Trees, Support Vector Machine (SVM), Naive Bayes, K-Nearest Neighbour and Multi-Layer Perceptron algorithms (as in [195], [61], [214], [59]). Previous research indicates that Gradient Boosted Decision Trees, Random Forest, Decision Tree or SVM are likely to perform the best of those considered.

To mimic the challenge of analysing new incoming samples, the train and test set are split using only the samples that were first seen by VirusTotal after 11:15 on 10th October 2017. This does not account for variants of the same family being present in both the test and training set, but this question is explored in Section 4.3.3.

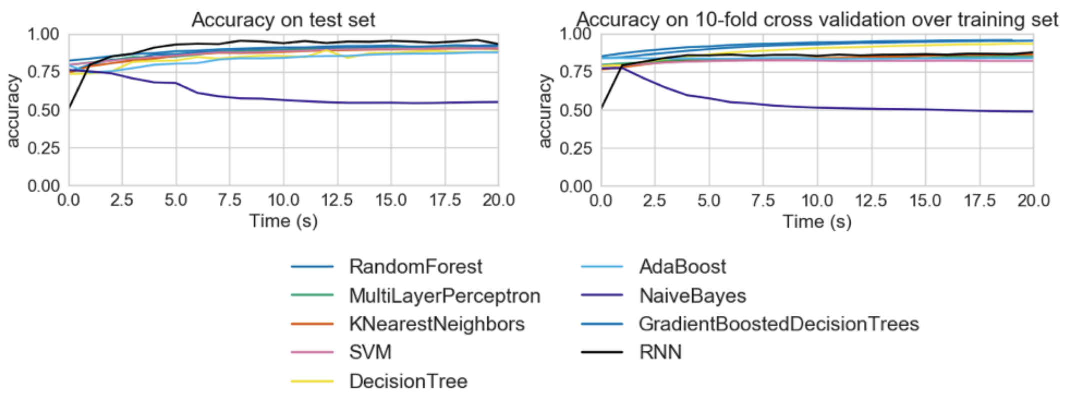


Figure 4.3: Classification accuracy (percentage with correct labels) for various machine learning algorithms and a recurrent neural network as time into file execution increases

Figure 4.3 shows the accuracy trend as execution time progresses for the 10-fold cross validation on the training set and on the test set. Gradient boosted decision tree

Classifier	Accuracy (%)	Time (s)	FP (%)	FN (%)
RandomForest	95.29	19	**5.03**	4.5
MultiLayerPerceptron	85.01	20	21.3	9.83
KNearestNeighbors	86.3	20	17.53	10.96
SVM	82.39	10	24.5	10.62
DecisionTree	93.41	20	7.87	5.72
AdaBoost	83.94	**2**	19.78	12.03
NaiveBayes	77.44	**2**	29.78	10.7
GBDT	**95.81**	19	5.44	**3.32**
RNN	87.75	20	10.93	15.15

Table 4.4: Highest average accuracy (percentage with correct labels) over 10-fold cross validation on training set during first 20 seconds of execution with corresponding false positive rate (FP) and false negative rate (FN)

Classifier	Accuracy (%)	Time (s)	FP (%)	FN (%)
RandomForest	92.05	20	4.29	12.29
MultiLayerPerceptron	91.07	18	5.53	12.98
KNearestNeighbors	90.38	18	4.66	15.12
SVM	90.57	20	5.13	14.39
DecisionTree	89.17	12	5.22	17.22
AdaBoost	87.82	19	7.24	17.72
NaiveBayes	76.25	**0**	24.74	21.13
GBDT	92.62	20	4.33	11.08
RNN	**96.01**	19	**3.17**	**4.72**

Table 4.5: Highest accuracy (percentage with correct labels) on unseen test set during first 20 seconds of execution with corresponding false positive rate (FP) and false negative rate (FN)

(GBDT) achieves the highest 10-fold mean accuracy over the 20 seconds of execution on the training set (see Table 4.4), but the RNN achieves the highest accuracy on the unseen test set (see Table 4.5) and outperforms all other algorithms on the unseen test set after 2 seconds of execution (see lower graph in Figure 4.3). This could be because the training set is relatively homogeneous and so relatively easy for the GBDT to learn, but it is unable to generalise as well as the RNN to the completely new files in the test set. The RNN cannot usefully learn from 0 seconds as there is no sequence to analyse so accuracy is equivalent to random guess. Using just 1 snapshot (at 1 second) of machine activity data, the SVM performs best on the test set and is able to classify 80% of unseen samples correctly. But after 2 seconds the RNN performs consistently better than all other algorithms. Using 4 seconds of data the RNN correctly classifies 91% of unseen samples, and achieves 96% accuracy at 19 seconds into execution, whereas the highest accuracy at any time predicted by any other algorithm is 93% (see GBDT in Table 4.5). The RNN improves in accuracy as the amount of sequential data (lags) used for training and classification increases. Although peak accuracy occurs at 19 seconds, the predictive accuracy gains per second begin to diminish after 4 seconds. From 0 to 4 seconds accuracy improves by 41 percentage points (11 percentage points from 1 second to 4 seconds) but only by 5 points from 4 to 19 seconds. Our results indicate that dynamic data from just a few seconds of execution can be used to predict whether or not a file is malicious. At 4 seconds it is possible to accurately classify 91% of samples, which constitutes an 8 percentage point loss from the state of the art dynamic detection accuracy[87] in exchange for a 04:56 minutes time saved from the typically documented data recording time per sample (see Table 4.6), making the model a plausible addition to endpoint anti-virus detection systems. One point to note about the accuracy at 20 seconds, there is a decrease, this is likely because the timing of the virtual machine shutting down was not precise as 20 seconds so may have impacted the number of samples executing at this time therefore creating a significant change in the number of samples that were executing at 19 seconds. The next-best performing algorithm on the test set is the GBDT closely followed by the random forest, these two algorithms show similar percentage accuracy of labels, false positive and false negative rates which is perhaps unsurprising since both are ensemble tree-based methods.

Time (s)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Acc. (%)	80	85	87	91	93	94	93	95	95	94	95	94	95	95	95	95	94	95	96	93
FN (%)	12	14	16	14	10	9	10	5	7	9	6	9	6	7	7	6	9	7	5	7
FP (%)	33	17	9	2	2	3	2	3	2	2	2	2	4	3	2	4	3	3	3	5

Table 4.6: RNN prediction Accuracy (Acc.) (percentage with correct labels), false negative rate (FN) and false positive rate (FP) on test set from 1 to 20 seconds into file execution time

4.3.3 Simulation of Zero-day Malware Detection

Dividing the test and training set by date ensures that the two groups are distinct sets of files. However, a slight variant on a known strain is technically a new file. Using the AV labels from virus total, it was possible to extract approximate labels for the samples malware families and variants and to test how well the model performs when confronted with a completely new family or variant.

Table 4.7 gives the numbers in the test set for the families and those variants for which there were more than 100 instances in the dataset. Dinwod, Eldorado, Zusy and Wisdomeyes are Trojans; Kazy and Scar are Viruses. Those variants listed as advanced persistent threats (APTs) are also counted since signature-based systems struggle to detect these if previously unseen. The APTs and some of the high-level families have less than 100 samples and as such the results are unlikely to be indicative for the general population of that family but are tested anyway for comparison.

To avoid contamination from those samples that were disputed, these are removed from the dataset for the following experiments. For each family in Table 4.7, a completely new model was trained without any samples from the family of interest.

The test set is entirely malicious, which means accuracy is an appropriate metric as it is just the rate of correct detection from the only class of interest. Table 4.8 gives the predictive accuracy over time for different families and for APTs, and Table 4.9 gives

Family	Total
Trojan	1,382
Virus	407
Adware	180
Backdoor	123
Bot	76
Worm	24
Rootkit	11
APTs	27
Variant	**Total**
Dinwod	265
Artemis	228
Eldorado	209
Zusy	135
Wisdomeyes	132
Kazy	116
Scar	101

Table 4.7: Number of samples belonging to different families and variants with over 100 samples in the dataset

the predictive accuracies for the five variants for which more than 100 instances were observed in the dataset. Perhaps surprisingly, there are high classification accuracies across these two sets of results.

The families are detected with lower accuracy in general than the variants. For the Trojans particularly, during the first few seconds, accuracy is worse than random chance. Because so much of the dataset set is comprised of Trojans, removing these from training halves the number of malware samples, so this may account for the particularly poor performance. The accuracy does increase significantly between 1 and 3 seconds of execution. This is possibly because Trojans are defined by their delivery mechanism, and the model has not been trained on any examples of this form of malware delivery. The model has, however, seen malicious behaviour from other families, which may be similar to some of the later behaviours by the Trojans, accounting for the significant rise in accuracy. Though the Worms are detected with a 100% accuracy at each second, there were only 24 Worm samples in the dataset.

The high detection accuracy for some families does not mean that the model will always be able to distinguish future malware families but rather indicates some underlying data similarity of the different samples' behavioural data despite belonging to different strains. Figure 4.4 illustrates the *mean* features over time comparing these families against benignware. It appears that there is almost linear separability between benignware and malware families regarding memory and rx_bytes but it is not so simple since these average figures mask the full range of values. It does, however show that the *mean* behaviour is quite well divided between malware and benignware some these two features, together with swap after some initial seconds. This helps to illustrate that despite the families being different, there are common underlying behaviours between them. It is also possible to pick out that adware appears to use far more CPU on average than other types.

Family	Time(s)									
	1	2	3	4	5	6	7	8	9	10
Trojan	11.16	49.67	70.23	68.07	73.86	69.33	55.63	57.75	60.18	56.24
Virus	91.26	89.58	82.7	83.0	83.54	88.89	84.56	86.31	84.38	82.26
Adware	90.68	90.0	83.33	84.11	59.59	85.71	87.22	66.41	77.31	73.5
Backdoor	91.3	91.21	80.0	83.53	82.28	79.73	87.32	82.61	79.69	80.7
Bot	93.06	91.55	92.86	84.85	90.16	85.71	80.0	86.36	88.1	87.5
Worm	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Rootkit	100.0	75.0	75.0	75.0	100.0	75.0	100.0	100.0	66.67	100.0
APT	96.3	96.3	88.46	92.0	100.0	94.74	94.74	100.0	94.74	89.47

Table 4.8: Classification accuracy on different malware families with all instances of that family removed from training set

Variant	Time(s)									
	1	2	3	4	5	6	7	8	9	10
Dinwod	90.57	89.43	78.11	91.32	93.96	98.87	99.25	98.11	98.08	97.31
Eldorado	94.3	93.3	92.0	86.42	90.07	82.01	74.81	81.75	85.48	83.61
Wisdomeyes	92.59	90.91	83.72	91.34	89.83	92.63	94.44	84.52	90.36	87.34
Zusy	91.18	89.63	85.94	82.11	81.74	85.19	85.29	88.66	90.43	85.56
Kazy	89.74	82.76	85.22	86.49	87.88	94.94	87.5	88.89	91.43	89.71
Scar	92.08	92.08	75.25	78.22	62.63	81.82	89.69	81.44	86.46	88.42

Table 4.9: Classification accuracy on different malware variants with all instances of that variant removed from training set

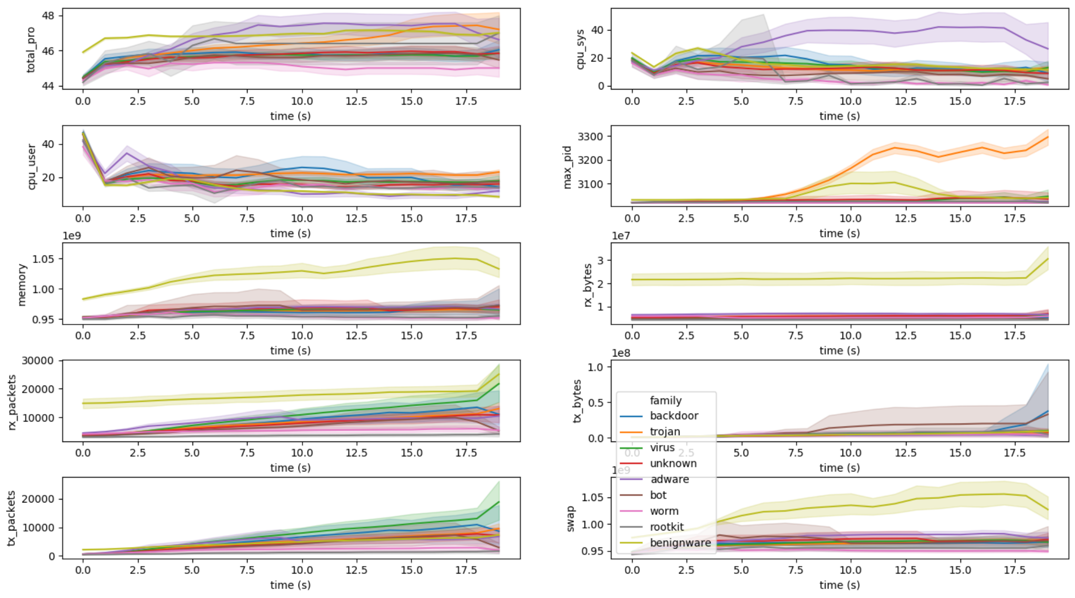


Figure 4.4: Features over time for each malware family and benignware

The variants tend to achieve a higher predictive accuracy than the families. Other than Dinwod, all families score lower at 10 seconds than at 1 second. Each variant is a kind of Trojan or Virus, but the model was trained on other types of Trojan and Virus. This can help explain the slight drop in accuracy over the first 10 seconds. It is the delivery mechanism which the variants have in common with samples in the training set, so the period over which this occurs (the first few seconds) gives the best predictive accuracy. Every variant was detected with over 89% accuracy during the first second of execution, despite the model having no exposure to that variant previously.

If the model is able to score well on a family without ever having seen a sample from that family, the model may hold a robustness against zero days, and support our hypothesis that malware do not exhibit highly different behavioural activity from one another as their goals are not highly divergent, even if the attack vector mechanisms are.

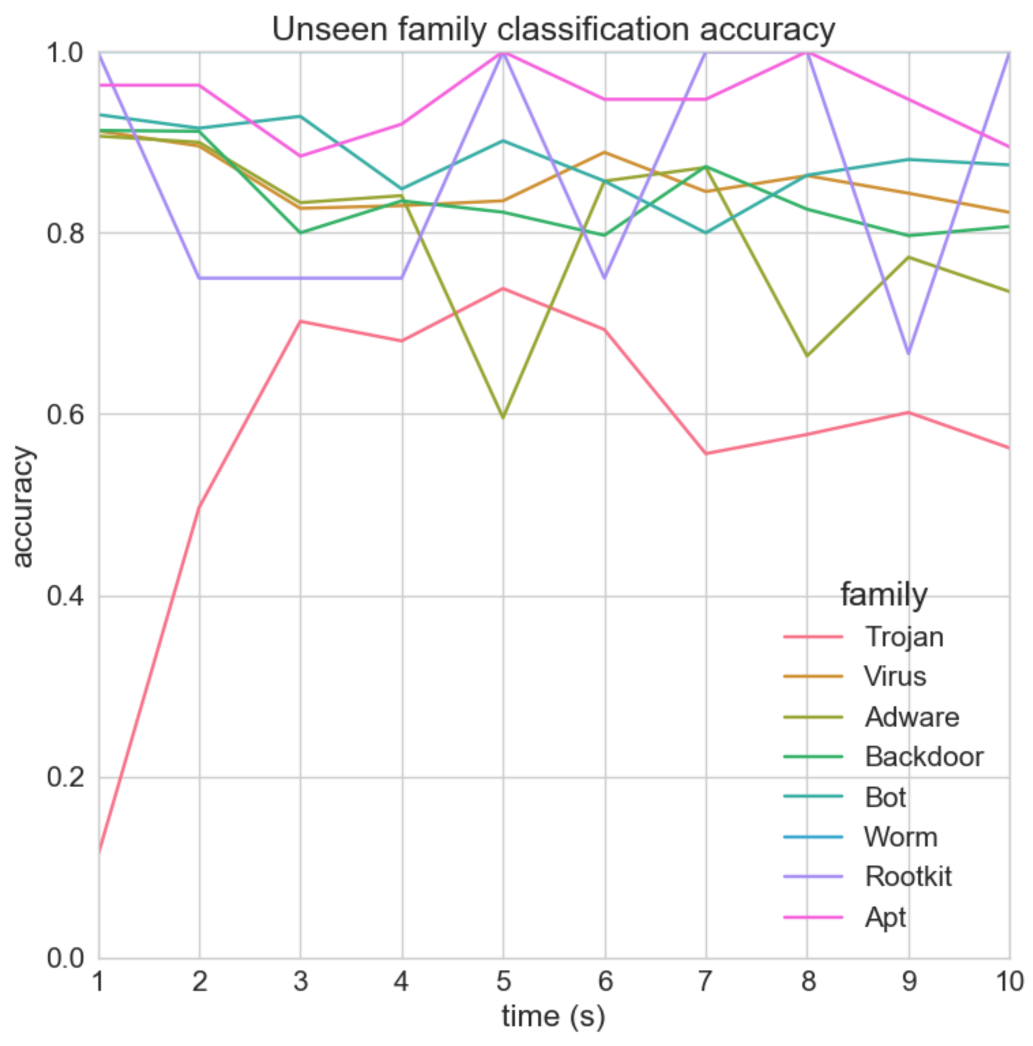


Figure 4.5: Comparative detection accuracy on various malware families with examples of the family omitted from the training set

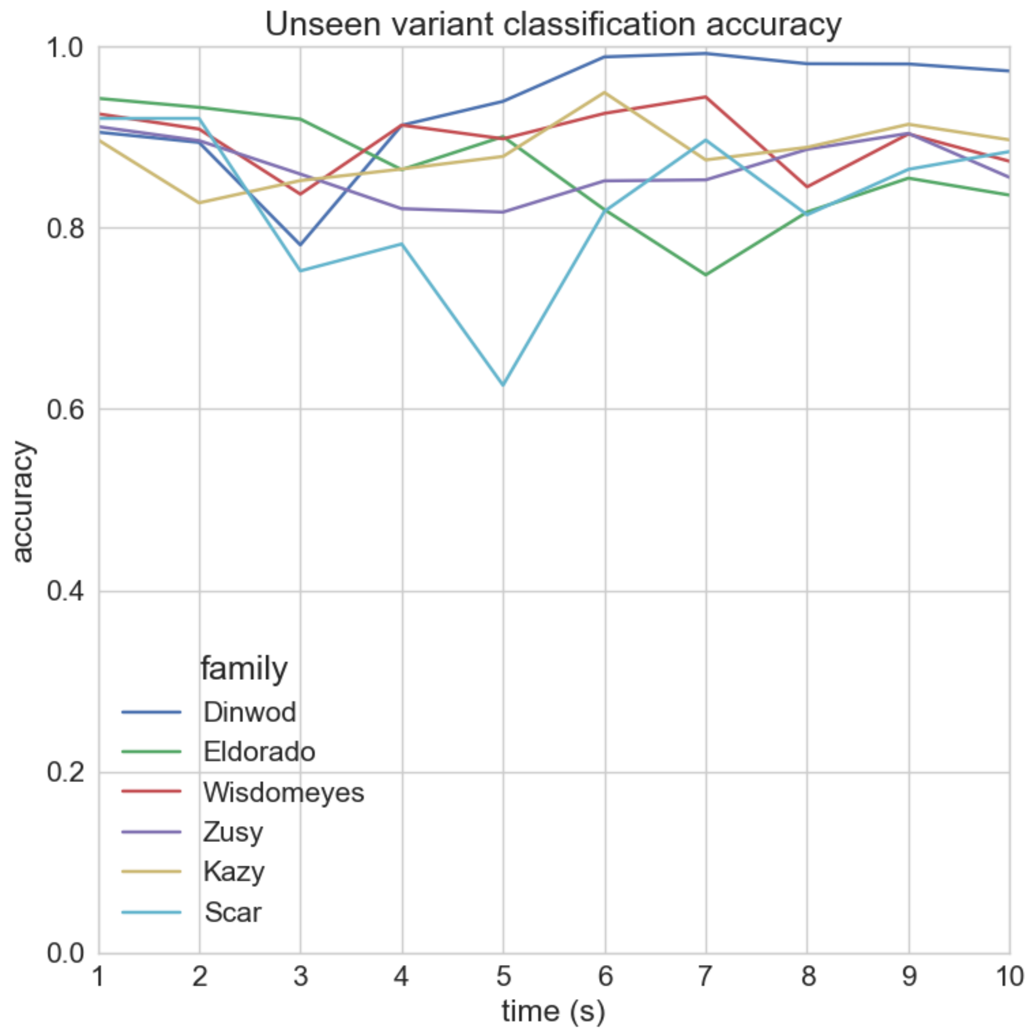


Figure 4.6: Comparative detection accuracy on various malware variants with examples of the variant omitted from the training set

4.3.4 Ransomware Case Study

Early prediction that a sample is malicious enables defensive techniques to move from recovery to prevention. This is particularly desirable for malware such as ransomware, from which data recovery is only possible by paying a ransom if a backup does not exist. An additional 2,788 ransomware samples from the VirusShare website [201] were collected to test the predictive capability of our model.

Reports in the wake of the high profile ransomware attacks, e.g. WannaCry/WannaDecryptor worm in May 2017, were reported to be preventable if a patch released two months earlier had been installed [198]. Endpoint users cannot be relied on to carry out security updates as the primary defence against new malware and it is non-trivial challenge for system administrators to oversee and patch large, legacy and heterogeneous infrastructure as often exists in healthcare systems, governments and businesses. The model is tested in it's ability to detect ransomware as a new threat by removing the 183 ransomware samples and the disputed-family samples from our original dataset and train a new model on the remaining samples, this model is then used to classify the VirusShare samples and the removed 183 samples.

The model is able to detect 94% of samples at 1 second into execution without having seen any ransomware previously. When ransomware is included in the training set this rises to 99.86% (see Table 4.10). Further research could explore the minimum number of samples and variance of samples required to boost accuracy, here, as a first experiment, randomly, half of the samples are included in training and half used for testing.

In Figure 4.7 there is a clear distinction in the accuracy trend over execution time between the model which has been trained on some of the relevant family. The model which has never seen ransomware before starts to drop in accuracy after the initial few seconds. Again this could be because the model is recognising the delivery mechanism at the start of execution, which will be common to other types of malware in the

Samples in Training Set	Time(s)									
	1	2	3	4	5	6	7	8	9	10
Omitted	94.19	93.72	90.94	92.02	86.77	92.46	89.55	87.62	77.88	87.52
Half included	99.86	99.1	97.96	98.83	98.29	97.89	98.78	99.29	97.96	96.46

Table 4.10: Classification accuracy (percentage with correct labels) on ransomware for one model which has not been trained on ransomware (omitted), and for one which has (half included)

training set, though the later malicious behaviour is less recognisable to the model by comparison with the later behaviour of the other types of malware it has seen. The model trained with half of the samples knows how ransomware behaves after a few seconds and so maintains a high detection accuracy.

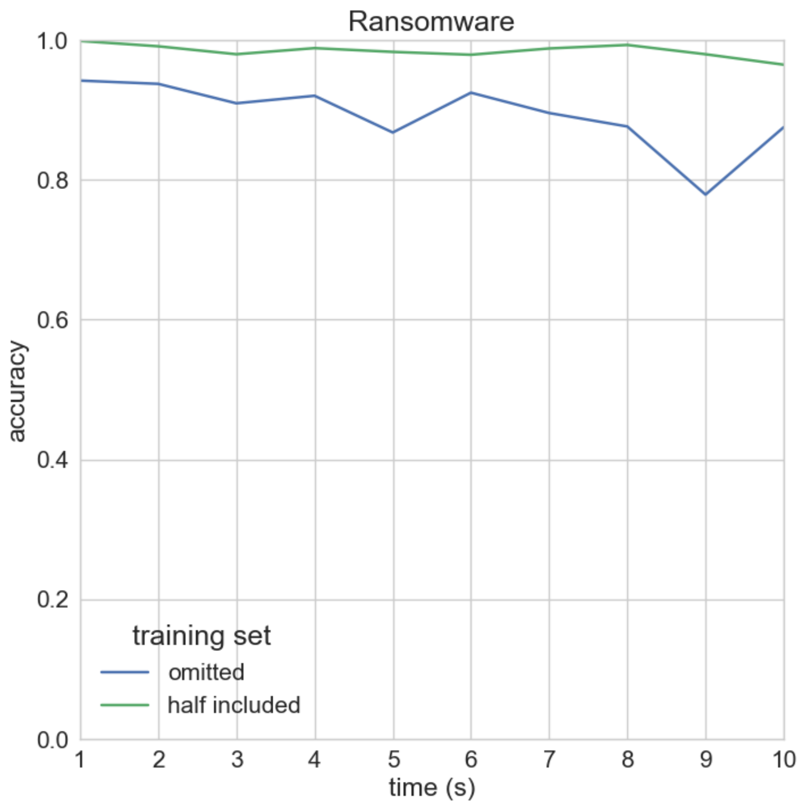


Figure 4.7: Classification accuracy on ransomware for one model which has not been trained on ransomware (omitted), and for one which has (half included)

It would be interesting to see if the model at 1 second and the model at 5 seconds rely on different input features to reach accurate predictions. It is difficult to penetrate the decision making process of a neural network; the architecture presented here has 1,344 neurons almost 4 million trainable parameters, but it is possible to turn the input features on and off and see the effect of combinations of features on classification accuracy. By setting the inputs to zero, which is the normalised mean of the training data, a feature can be turned 'off'. By turning off all the features and then turning them back on sequentially, it is possible to see which features are needed to gain a certain level of accuracy.

	Ransomware omitted from training set				Ransomware in training set			
# Features on	1 second model		5 second model		1 second model		5 second model	
	Max. Acc.	Features on	Max. Acc.	Features on	Max. Acc.	Features on	Max. Acc.	Features on
1	00.03	tx bytes	40.82	memory	89.36	rx packets	14.95	total processes
2	98.92	memory and rx bytes	97.54	rx bytes and rx packets	99.80	tx packets and {rx packets, rx bytes}	71.15	rx bytes and tx bytes

Table 4.11: Maximum accuracy scores in predicting ransomware with only one and two features turned on for a model not trained on ransomware and for a model trained on ransomware

Table 4.11, shows that with just two features, both the 1 second and the 5 second models trained with and without ransomware are able to beat 50% accuracy (random chance). The model trained using ransomware is able to correctly detect more than 99% of ransomware samples as malicious using just the number of packets sent and either the number of packets or number of bytes received. Unlike the model trained with ransomware, which draws accurate conclusions from packet data and total processes, when no ransomware is included in the training set, memory usage is also a prominent feature in accurate detection. Comparing to the broader families, in classifying Adware, Trojans and Viruses, memory and packets a single input feature allowed the model to achieve more than 50% accuracy, Trojans are the only family for which memory contributes to scoring above 50% at the one-second model, when combined with packets sent and swap. As Trojans comprise the majority of the dataset it makes sense that the most relevant features for classifying them help to define what constitutes malware to the model.

The accuracy in identifying unseen families highlights the presence of shared dynamic characteristics between different malware types. The broad families, which detail the malware infection mechanism particularly help to identify malware early on. Whilst new malware variants are likely to appear, new delivery mechanisms are far less common and help to distinguish unseen families from benignware.

4.3.5 Improving Prediction Accuracy With an Ensemble Classifier

As well as accuracy, the values of the model predictions increase with time into file execution. Therefore an ensemble method is presented, using the top three best performing configurations found in the hyperparameter search space during the previous experiments, to try and improve the classification confidence earlier in the file execution. Accuracy does not increase monotonically in our first configuration, and of the best three configurations on the 10-fold cross-validation, no single configuration consistently achieved the highest accuracy at each second, the configuration used in the previous sections was the configuration that scored the highest accuracy at 1 second.

This is comprised of the best-scoring configurations on the training set across the first 5 seconds, which are 3 distinct hyperparameter sets (one model was the best at 1 and 2 seconds, one at 3 and 5 seconds) and take the maximum of the predictions of these three RNNs before thresholding at 0.5 to give a final malicious/benign label. The configuration details are in Table 4.12, configuration ‘A’ is the same as has been used in the previous experiments.

Hyperparameter	A	B	C
Depth	3	1	2
Bidirectional	True	True	False
Hidden neurons	74	358	195
Epochs	53	112	39
Dropout rate	0.3	0.1	0.1
Weight regularisation	$l2$	$l2$	$l1$
Bias regularisation	None	None	None
Batch size	64	64	64

Table 4.12: Highest accuracy-scoring configurations during first 5 seconds in 10-fold cross validation on training set

Time (s)	Highest accuracy of configurations A, B and C	Ensemble acc. (%)	Ensemble FP (%)	Ensemble FN (%)
1	**79.69** (C)	79.5	33.5	12.03
2	**85.6** (A)	83.69*	25.73	10.16
3	87.52 (A, C)	**88.48***	15.05	9.21
4	91.54 (A)	**91.92***	8.74	7.64
5	92.38 (B)	**93.95***	3.4	7.84
6	94.09 (A)	**95.28***	4.37	4.97
7	94.92 (A)	**95.12***	4.85	4.9
8	94.25 (A)	**95.48***	4.88	4.26
9	94.97 (A)	**96.02***	4.39	3.68
10	**95.53** (C)	95.11*	5.45	4.48
11	95.91 (C)	**96.13***	4.95	3.04
12	**95.46** (C)	**95.46***	5.47	3.82
13	95.16 (A)	**95.6***	5.97	3.15
14	**95.93** (C)	**95.93***	5.03	3.29
15	**96.1** (C)	95.87*	4.57	3.77
16	95.62 (C)	**96.54***	4.08	2.94
17	95.34 (C)	**96.5***	3.06	3.86
18	**96.67** (C)	96.43*	4.12	3.1
19	**96.51** (C)	96.26*	4.23	3.3
20	93.81 (A)	**94.85***	8.22	3.31

Table 4.13: Ensemble accuracy (acc.), false positive rate (FP) and false negative rate (FN) compared with highest accuracy of configurations A, B and C. Those marked with a '*' signify predictions that were statistically significantly more confident by at the confidence level of 0.01

To combine the predictions of configurations A, B and C the ensemble uses the maximum value of the three to bias the predictions in favour of detecting malware (labelled as 1) over benignware (labelled as 0). An ensemble of models does tend to boost accuracy, increasing detection from 92% to 94% at 5 seconds, and the maximum accuracy from configuration A alone, 96%, is reached at 9 seconds instead of at 19 seconds (see Table 4.13). The results in Table 4.13 show that the accuracy score improves or matches the highest scoring model of configurations A, B and C for 12 of the first 20 seconds. Model A, the original configuration, only bests the ensemble accuracy once. In order to determine whether the ensemble scores improved predictive confidence on the individual samples compared with the predictions of the best-scoring model. The predictive confidence is measured by by rewarding those correct predictions closer to 1 or 0 more highly, i.e. a prediction of 0.9 is better than 0.8 when the sample is malicious. The equation for predictive confidence is as follows:

$$confidence = 1 - |b - p|$$

where b is the true label and p is the predicted label.

A one-sided T-test showed that the confidence of predictions from the ensemble method were significantly higher (at 0.01 confidence level) after 2 seconds. Malicious predictions are likely be more confident as the ensemble uses the maximum value of the three models, but it is interesting that taking the maximum of the benign samples does not out weigh the increase in confidence. This indicates that three models are more confident about benign samples than malicious ones, which is reflected in the high true negative rate reported above for the individual model. A further benefit of the ensemble approach is the reduction in the FNR during the first 10 seconds. Model A does not achieve 5% FNR until 8 seconds but the ensemble does this at 6 seconds. The FNR is also less turbulent, remaining lower than Model A's global minimum (at 19 seconds) from 6 seconds onwards.

These tables highlight that there may be some trade-off between accuracy and time but that after a point, more accuracy is not gained with more time during the first 20 seconds. These experiments may serve as useful information for creators of dynamic detection systems by which to determine data collection time.

4.3.6 Feature Importance

The components of the ensemble must have learned different decision boundaries in order to improve on the individual accuracies of the components. Neural networks are often considered black-boxes due to the difficulty in explaining how inputs map to outputs through multiple hidden layers.

This section attempts to interpret what configurations A, B, and C are using to distinguish malware and benignware. These preliminary tests seek to gauge the relative importance of features to classification accuracy.

By setting the test data for a feature (or set of features) to zero, which is the mean of the data thanks to normalisation, it is possible to approximate the impact of reducing or increasing values to the mean. With categorical features, it would be possible to simply set them to zero, but for numeric continuous values it is difficult to nullify the impact of the data, therefore this is just an approximation. The overall impact of turning features ‘off” is judged by observing the fall in accuracy and dividing it by the number of features turned off. A single feature incurring a 5 percentage point loss attains an impact factor of -5, but two features creating the same loss would be awarded -2.5 each. The average across impact scores is used to approximate the importance of each feature when a given number of features are switched off.

Figure 4.8 gives the impact factors for each feature at 4 seconds into file execution. Intuitively, the more features omitted, the higher the impact factors become. Interestingly, there are some very small gains in accuracy for configurations A and B when only one feature is missing but no more than 0.2 percentage points. For each of the configurations, CPU use on the system has the highest impact factor. It is most integral for configuration A, which is also the best-scoring model. The CPU use in configuration A does not see a significant increase in its impact factor as more input features are removed, but for configuration B, all features attain higher impact factors the more are removed. Configuration B is more reliant on the interplay between features than the other two models.

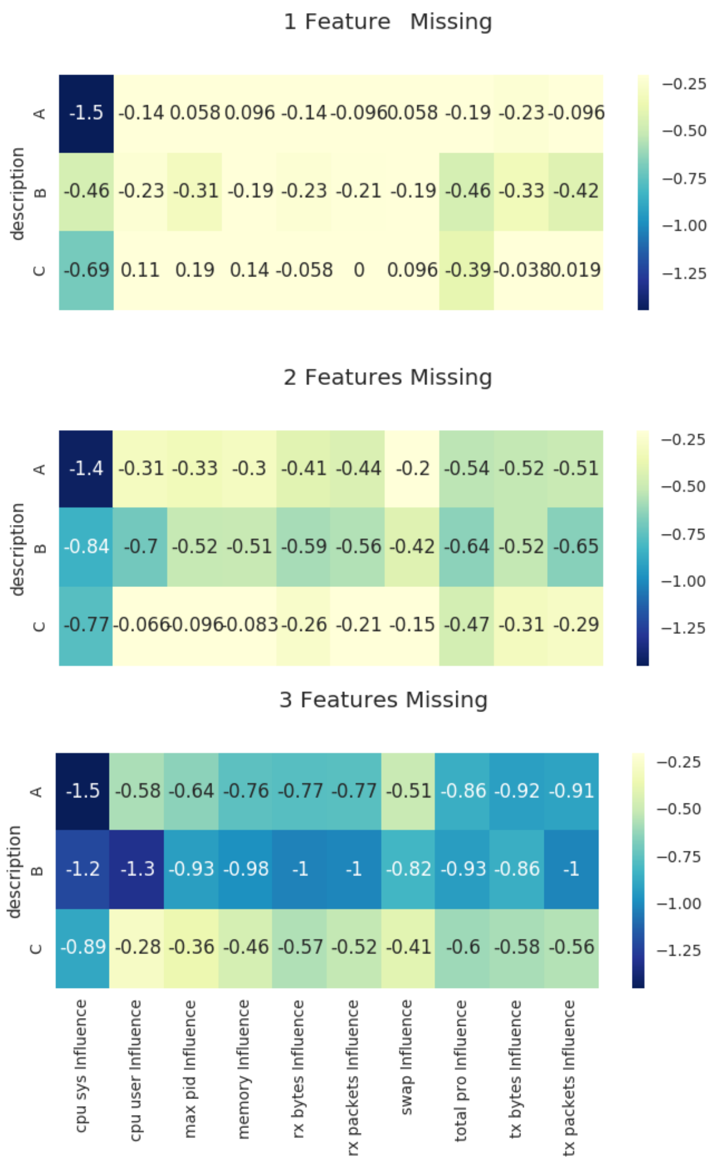


Figure 4.8: Impact scores for features with 1, 2 and 3 features turned off 4 seconds into file execution

The difference between the impact scores and their emphasis can help us to see which features are most predictive at different time steps (at 4 seconds this is CPU system usage) and to understand how an ensemble classifier is able to outperform the predictions of its components. As all three models suffer the biggest loss from CPU usage, if an adversary knew this she might be able to manipulate CPU system use to avoid detection. Future work should examine the decision processes of networks to detect potential weaknesses that could be exploited to evade detection. The ensemble offers a small increase in accuracy but more importantly, this analysis can help to understand ways in which the models may be manipulated, by biasing results towards malicious predictions (taking the maximum prediction) all models must be successfully manipulated in order to impact the overall results, this analysis indicates that this may be possible by altering CPU system usage and that the use of more diverse models may improve robustness against adversarial attacks.

4.4 Limitations and Future work

These results indicate that behavioural data can provide a good indication of whether or not a file is malicious based only on its initial behaviours, even when the model has not been exposed to a particular malware variant before. Dynamic analysis could reasonably be incorporated into endpoint antivirus systems if the analysis only takes a few seconds per file. Further challenges which must be addressed before this is possible include:

4.4.1 Robustness to Adversarial Samples

The robustness of this approach is limited if adversaries know that the first 5 seconds are being used to determine whether a file will run in the network. By planting long sleeps or benign behaviour at the start of a malicious file, adversaries could avoid detection in the virtual machine. This chapter hypothesised that malicious executables begin attempting their objectives as soon as possible to mitigate the chances of being interrupted, but this would be likely to change if malware authors knew that only subsections of activity were the basis of anti-virus system decisions. The next chapter looks at real-time detection as a mitigation against this vulnerability.

4.4.2 Process Blocking

If a live monitoring system is implemented, processes predicted to be malicious will need to be terminated. Future work should examine the ability of the model to block once the classifier anticipates malicious activity, and to investigate whether the malicious payload has been executed.

4.4.3 Portability to Other Machines and Operating Systems

The machine activity metrics are specific to the context of the virtual machine used in this experiment. To move towards adoption in an endpoint anti-virus system, the RNN should be retrained on the input data generated by a set of samples on the target machine. Though this recalibration will take a few hours at the start of the security system installation, it will only need to be performed when hardware is upgraded (once per machine for most users) and opens the possibility of porting the model to other operating systems, including other versions of Windows.

Though this chapter has not tested the portability of the data between machines, i.e. training with data recorded on one machine and testing with data recorded on another, it is easy to see cases in which this will not work. Some metrics, such as CPU usage are relative (measured as a percentage of total available processing power) and so will change dramatically with hardware capacities, for example a file requiring 100% of CPU capacity on one machine may use just 30% on another with more cores. Non-relative metrics such as memory usage (measured in bytes) may also be limited by the hardware capabilities of a machine. However, there is no clear reason why the model cannot be re-calibrated to a new machine. There would be cause for concern if the data fed into the model did not allow it to learn a representation of the data that distinguishes malicious and benign files. This loss of data could occur due to hardware limitations; for example a very small amount of RAM could limit the memory usage such that the useful information that one sample uses 1.1MB and another 1.2MB are both capped at 1MB, thus appearing the same to the model. In practice, this model could be used with live data from real machines, taking a real-time approach. The different results that would be observed using a dual-core processor offer a potential advantage in training but also necessitate re-calibration on a per-machine basis. Since this is a one-off time cost, it is not a major limitation of the proposed solution.

4.5 Conclusions

Dynamic malware detection methods are often preferred to static detection as the latter are particularly susceptible to obfuscation and evasion when attackers manipulate the code of an executable file. However, dynamic methods previously incurred a time penalty due to the need to execute the file and collect its activity footprint before making a decision on its malicious status. This meant the malicious payload had likely already been executed before the attack was detected. This chapter has presented a novel malware prediction model based on recurrent neural networks (RNNs) that significantly reduces dynamic detection time, to less than 5 seconds per file, whilst retaining the advantages of a dynamic model. This offers the new ability to develop methods that can predict and block malicious files before they execute their payload completely, preventing attacks rather than having to remedy them.

The experimental results demonstrate that it is possible to achieve a detection accuracy of 94% with just 5 seconds of dynamic data using an ensemble of RNNs and an accuracy of 96% in less than 10 seconds, whilst typical file execution time for dynamic analysis is around 5 minutes.

The best RNN network configurations discovered through random search each employed bidirectional hidden layers, indicating that making use of the input features progressing as well as regressing in time aided distinction between malicious and benign behavioural data.

A single RNN was capable of detecting completely unseen malware variants with over 89% accuracy for the 6 different variants tested at just 1 second into file execution. The accuracy tended to fall a little after the first 2 seconds, implying that the model was best able to recognise the infection mechanism at a family level (e.g. Trojan, Virus) given that this would be the first activity to occur. The RNN was less accurate at detecting malware at a family level when that family had been omitted from the training data (11% accuracy at 1 second detecting Trojans), further indicating that the model was easily able to detect new variants, provided it had been exposed to examples of that family of infection mechanisms. Our ransomware use case experiment supported this theory further, as the RNN was able to detect ransomware, which shares common infection mechanisms with other types of attack such as Trojans, with 94% accuracy, without being exposed to any ransomware previously. However, this accuracy fell as time into file execution increased, again implying that the model was easily able to detect a malicious delivery mechanism, better than the activity itself. After exposure to ransomware, the model accuracy remained above 96% for the first 10 seconds.

The RNN models outperformed other machine learning classifiers in analysing the unseen test set, though the other algorithms performed competitively on the training set, this used 10-fold cross validation, which does not reflect real-world use in which the samples using during inference would have been created later than the training samples. This indicates that the RNN was more robust against overfitting to the training set than the other algorithms and had learnt a more generalisable representation of the difference between malicious and benign files. This is particularly important in malware detection as adversaries are constantly developing new malware strains and variants in an attempt to evade automatic detection. The ensemble RNNs outperformed the individual components of the standalone network. This is not surprising as ensemble methods are widely used (the Random Forest, AdaBoost and GBDT algorithms can all be considered ensemble methods), however this does have implications for the computational overhead required to run such a model. The next chapter begins to address this consideration for power and memory resources.

These results contribute to the field of malware detection by answering *RQ2: Is it possible to predict that software is malicious from its early stages of execution?*. Whilst initial performance is promising, the longevity of this method would easily be challenged by malware authors triggering benign execution traces prior to the malicious activity. The next chapter therefore examines the possibility of dynamic run-time detection, which will eventually force the malware to reveal itself as it must do so in order to achieve its goals.

Chapter 5

Real-time Malware Detection and Process Killing

The previous chapter showed that early-stage malware detection using short behavioural traces can achieve high detection accuracy. This chapter imagines a future in which dynamic detection is widely adopted for endpoint protection by just using short behavioural traces. The early detection approach is limited since an attacker might simply inject some benign process behaviour at the start of execution. This limitation can be extended to virtual machine (VM)-based analysis that lasts for any period of time. Real-time analysis uses a sliding window of data such that, even if there is benign activity or a program sleep injected at the start of malware execution, the malicious activity should still be observed. Instead of looking at just the start of an application's activity, each individual process is monitored for its entire lifetime.

This chapter builds on the existing work in real-time malware detection to address the following unanswered research questions:

RQ3: Does the use of lower latency algorithms impact upon the speed of accurate malware detection?

and

RQ4: Is it possible to separate the signals of tens of simultaneous benign and ma-

licious processes?

and

RQ5: Is it possible to detect and kill malicious processes early enough to prevent malicious damage?

This chapter will make the case for real-time detection being a necessary component of endpoint defence in the next section (5.1). It will then discuss the key research challenges that this chapter will seek to address in 5.2. The subsequent two sections will outline the methodologies trialled to address these challenges (5.3). Section 5.5 will then report the experimental design followed by the general malware and benignware results (5.6) and prevented damage results (5.7). The chapter ends with discussion (5.8), future work (5.9) and conclusions (5.10).

5.1 The case for real-time detection with an automated response

When dynamic malware detection relies on the use of VMs there is always a risk that the data observed during analysis is not what would be observed on the endpoint. The malware author may have explicitly programmed conditional execution into the sample to evade virtual environments, or the version of software that will be exploited may simply not be installed on the VM. In the latter case the VM can be made to mirror the image of most machines on an IT estate, but what if the IT estate is very large and diverse? What if some people have carried out the latest security updates and others are still using an older version of Windows? This chapter argues that real-time detection is therefore a necessary part of endpoint defence-in-depth, since it eliminates any disparity between the observed activity on the victim machine and observed activity in the analysis environment.

An important point to note is that this chapter does use virtual machines for the purposes of experiments as a convenient way to infect and restart machines. The VMs used are designed to mimic use by a real user, in that benign applications are being launched and interacted with and the file system is populated with documents and multimedia. The details are elaborated later in this chapter (section 5.5.2). This limits the training data collected since the malicious activity of each application is not verified due to the large number of samples. However, this work is still novel and future work could take this improved labelling into consideration to improve the model. As highlighted in the background chapter (2), Sun et al.'s work [188] has looked at running benign and malicious applications simultaneously to mimic real-use but has only run a maximum of 5 applications at once. This chapter asks what happens to detection accuracy when this number is increased to 35.

Real-time detection introduces new risks, namely that the malware could damage a real endpoint or network. To mitigate against the likelihood of damage, this chapter focuses on very early detection of malicious processes (using the methods developed in the previous chapter) - and builds on this to include an automatic immediate response. The underlying assumption here is that less execution time reduces the opportunity for the malware to cause damage, this assumption is tested in section 5.7. A human response would both require round-the-clock monitoring and the emergency support required for mitigation, which is expensive, and would also be too slow to mitigate against many types of malware damage, as Correa noted in a 2016 article, some ransomware can encrypt an entire hard drive in 17 seconds [43].

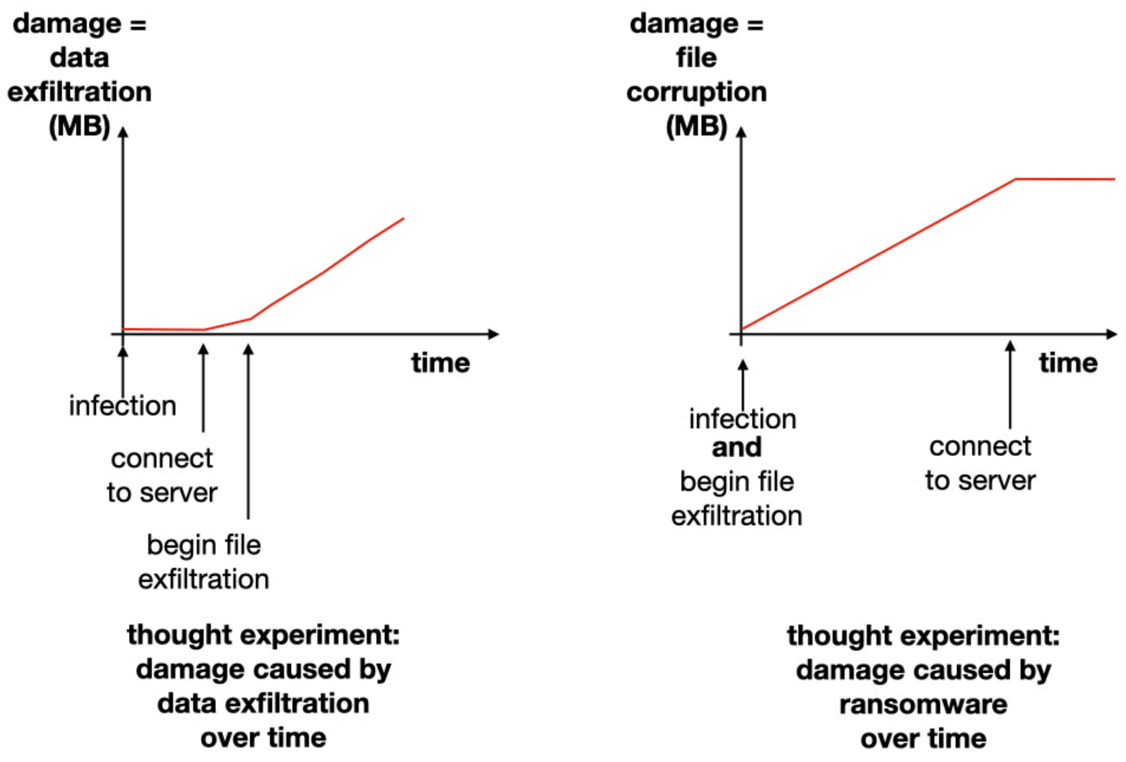


Figure 5.1: The damage caused over time by two imaginary malware, one which contacts a server before inflicting damage (left) and one which does so afterwards (right).

This raises a question around the data used to detect malware - is there is enough non-damaging malicious activity to allow the model to detect and kill the attack before the payload fully executes or is the damage itself the signal that the model requires in order to act? Figure 5.1 draws the damage over time of two imaginary malware samples: one conducting data exfiltration and one a ransomware attack. The data exfiltration sample contacts a server first and begins causing damage a little later, this could allow the detection model both more data and more time in order to detect and block the malware, on the other hand, is the malicious behaviour required in order to detect the malware in the first instance? This is not a question that is directly addressed in this chapter since it would require a definition of when 'damage' begins and thus reverse engineering all of the samples in the test dataset and in some cases the damage caused will be subjective or context dependent. However, a proxy experiment is carried out to measure the damage inflicted by ransomware with and without automated detection and process killing (section 5.7).

Implementing real-time detection on an endpoint that is being used by a real user with real assets that could be compromised introduces several new challenges beyond those of dynamic detection in a sandbox. These are outlined below:

5.2 Research Challenges

Moving from supervised learning models to automated agents which detect and respond to malware presents some significant challenges. During supervised learning the model continues to update its parameters with respect to some loss function; often this means minimising the number of wrong predictions *on average* over the entire trace of each sample and over the entire training set with misclassifications early and late in the trace being treated equally. Unlike post-trace detection, a model which integrates an automated agent is able to impact its world, thus changing the data that is available for analysis.

Consider a scenario in which a benign sample runs for 5 minutes (300 seconds), the supervised model correctly classifies this sample for 299 of the 300 seconds, averaging out to a correct classification. A real-time model, however, provides classifications constantly whilst the sample executes. If the misclassification (as malware) occurs at 10 seconds into execution but correctly throughout the rest of the program, this is likely to satisfy the supervised learning algorithm since the average accuracy is very high (99.67% = 299*100/300). But the automated agent would causes the program to be killed at 10 seconds, thus the 290 remaining opportunities for the model to correctly classify the sample correctly after the 10 second mark do not exist.

For real-time detection with an automated action attached, the impact of a misclassification has greater potential to disrupt the user than a misclassification which simply produces a prediction label for later consideration by a human.

The likelihood of false positives is higher than for post-trace detection models since the model makes a new decision each second and if the process is automatically killed, there is no opportunity to reclassify it later. A single false positive detection during several hours of true negative detections will terminate that process (and its child processes); this can mean work being lost, interruption to business processes and other significant consequences for the user.

As outlined in chapter 2, there are 4 key problems that require solving in the jump from whole-application analysis in a sandbox to real-time detection with multiple other applications running and being analysed simultaneously, these are summarised below:

- **Signal Separation:** Detection in real time requires that the malicious and benign activity are separated in order that automated actions can be taken on only the malicious processes.

- **Use of Partial Traces:** In order to try and mitigate damage, malware needs to be detected as early as possible but, as shown in the previous chapter, there is a trade-off between the amount of data collected and classification accuracy in the first few seconds of an application launching and the same may be true for individual processes.

- **Quick Classification:** The inference itself should be as fast as possible in order to further limit the change of malicious damage once the proces is deemed malicious.
- **Impact of Automated Killing in Supervised Learning:** Supervised learning averages the error rate across the entire training set but when the classification results in an action, this smoothing out of errors across the temporal dataset is not possible.

As outlined in the background chapter (2), these challenges have not been addressed by previous works with the exception of signal separation and the use of partial traces which have been partially tackled by Sun et al. [188] and Das et al. [48].

The next section describes the approaches trialled in order to address these challenges.

5.3 Methodology: Supervised Learning to Automated Agent

As explained above, supervised learning models average errors across the training set but in the case of real-time detection and process killing, a *single* false positive on a benign process amongst 300 true-negatives would cause disruption to the user. The time at which an malware is detected is also important, the earlier the better. Therefore the supervised learning model needs to be adapted to take account of these new requirements.

This chapter hypothesises that accurate models which consume less computational resources will perform better at blocking malicious activity since they will execute more quickly and thus reduce the opportunity for damage to take place. The computational resources and speed of execution are evaluated alongside detection accuracy.

In this chapter automated malicious process detection and killing is tackled using three methods listed here and explained and justified in detail below. These three methods were all tested and therefore all three are reported in the interests of future work.

1. Statistical methods to smooth the alert surface and filter out single false-positives
2. Reinforcement learning, which is capable of incorporating the consequences of model actions into learning
3. A regression model based on the feedback of a reinforcement learning model made possible by having the ground-truth labels

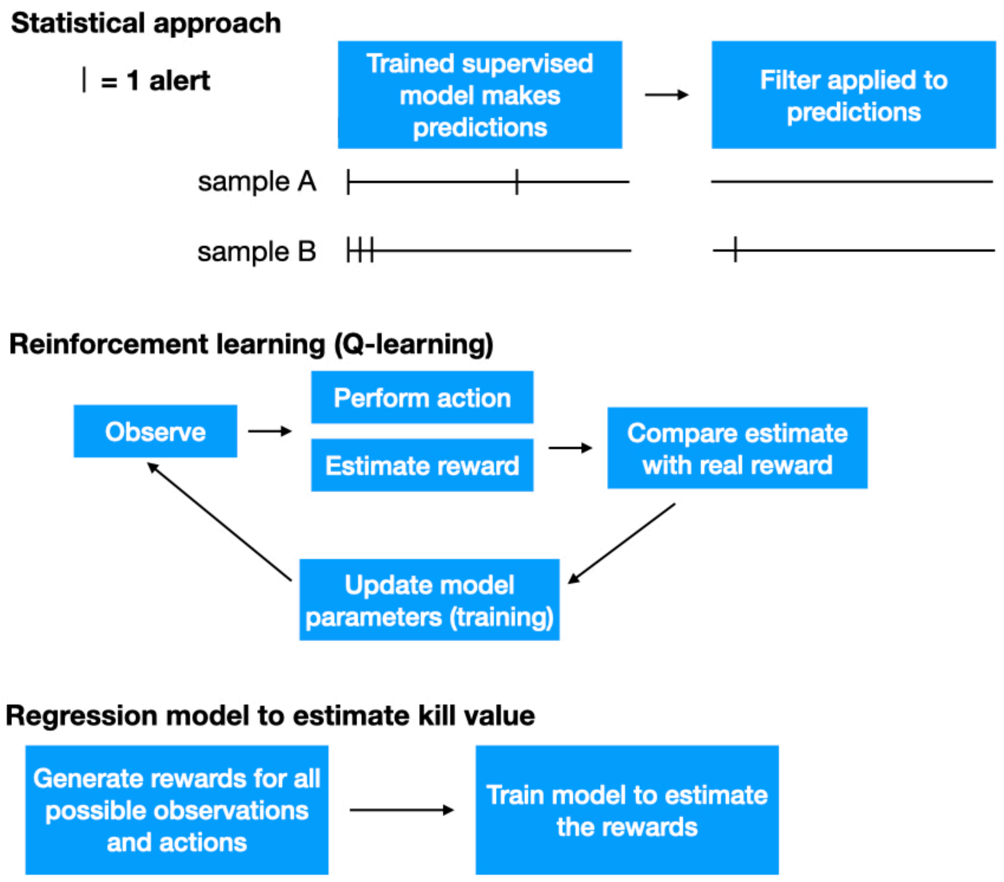

Statistical approach
| = 1 alert
Trained supervised model makes predictions
Filter applied to predictions
sample A
sample B
Reinforcement learning (Q-learning)
Observe
Perform action
Estimate reward
Compare estimate with real reward
Update model parameters (training)
Regression model to estimate kill value
Generate rewards for all possible observations and actions
Train model to estimate the rewards

Figure 5.2: High-level depiction of three approaches taken

Figure 5.2 gives a high-level depiction of the three approaches tested in this chapter.

5.3.1 Statistical Approach: Alert Filtering

It is expected that transitioning from a supervised learning model to a real-time model will see a rise in false-positives since one single alert means benign processes (and all child processes) are terminated, which effectively renders all future data points as false positives. Filtering the output of the models, just as the human brain filters out transient electrical impulses in order to separate background noise from relevant data [18], may be sufficient to make supervised models into suitable agents. This is attractive because supervised learning models are already known to perform well for malware detection, as confirmed by the previous chapter and other related work [48, 86, 99, 156]. A disadvantage of this approach is that it introduces additional memory and computational requirements both in order to calculate the filtered results and to track processes current and historic scores, therefore a model which integrates the expected consequences of an action into learning is also tested: reinforcement learning.

5.3.2 Reinforcement Learning: Q-learning with Deep Q Networks

The proposed automated killing model may be better suited to a reinforcement learning strategy than to supervised learning. Reinforcement learning uses rewards and penalties from the model's environment. The problem that this chapter is seeking to solve is a supervised learning problem, but one for which it is not possible to average predictions. There are no opportunities to classify the latter stages of a process if the agent kills the process, and this can be reflected by the reward mechanism of the reinforcement learning model (see Figure 5.2 above). Therefore reinforcement learning seems like a good candidate for this problem space.

Two limitations of this approach are that *(1)* reinforcement learning models can struggle to converge on a balanced solution, the models must learn to balance the exploration of new actions with the re-use of known high-reward actions; commonly known as the exploration-exploitation trade-off [190] *(2)* in these experiments, the reward is based on the malware/benignware label at the application level rather than being linked to the actual damage being caused, therefore the signal is a proxy for what the model should be learning. This is used because, as discussed above, the damage caused by different malware is subjective.

For reinforcement learning, loss functions are replaced by reward functions which update the neural network weights to reinforce actions (in context) that lead to higher rewards and discourage actions (in context) that lead to lower rewards; these contexts and actions are known as state-action pairs. The reward is calculated from the perceived value of the new state that the action leads to e.g. points scored in a game. Often this cannot be pre-labelled by a researcher since there are so many (maybe infinite) state-action pairs. However in this case, all possible state-action pairs can be enumerated, which is the third approach tested (regression model - outlined in the next section).

The reinforcement model was still tested. Here the reward is $+N$ for a correct prediction, $-N$ for an incorrect prediction where N is the total number of processes impacted by the prediction e.g. if there is only one process in a process tree but 5 more will appear over the course of execution, a correct prediction gives a reward of $+6$, and incorrect prediction gives a reward of -6.

There are a number of reinforcement learning algorithms to choose from. This chapter explores q-learning [114, 189, 209, 210] to approximate the value or 'quality' (q) of a given action in a given situation. Q-learning approximates q-tables, which are look-up tables of every state-action pair and their associated rewards. A state-action pair is a particular state in the environment coupled with a particular action i.e. the machine metrics of the process at a given point in time with the action to leave the process running. When the number of state-action pairs becomes large, it is easier to approximate the value using an algorithm. Deep Q networks (DQN) are neural networks that implement q-learning and have been used in state-of-the-art reinforcement learning arcade game playing, see Mnih et al. [128]. A DQN was the reinforcement algorithm trialled here, though it did not perform well by comparison with the other methods, a different RL algorithm may perform better [129], but the results are still included in the interests of future work. The following paragraphs will explain some of the key features of the DQN.

The DQN tries out some actions, stores the states, actions, resulting states and rewards in a memory and uses these to learn the expected rewards of each available action; with the highest expected reward being the one that is chosen. Neural networks are well-suited to this problem since their parameters can easily be updated, tree-based algorithms like random forests and decision trees can be adapted to this end but not as easily. Future rewards can be built into the reward function and are discounted according to a tuned parameter signified by γ.

In Mnih et al's [128] formulation, in order to address the exploration-exploitation trade off, DQNs either exploit a known action or explore a new one, with the chance of choosing exploration falling over time. When retraining the model based on new experiences, there is a risk that previous useful learnt behaviours are lost, this problem is known as catastrophic forgetting [103]. Mnih et al's [128] DQNs use two tools to combat this problem. First, experience replay by which past state-action pairs are shuffled before being used for retraining so that the model does not catastrophically forget. Second, DQNs utilise a second network, which updates at infrequent intervals in order to stabilise the learning.

This chapter hypothesises that q-learning may enable a model to learn when it is confident enough to kill a process, using the discounted future rewards. For example, choosing not to kill some malware at time t may have some benefit as it allows the model to see more behaviour at t+1 which gives the model greater confidence that the process is in fact malicious. Whilst there are other reinforcement learning algorithms.

Q-learning approximates rewards from experience, but in this case, all rewards from state-action pairs can be pre-calculated. Since one of the actions will kill the process and thus end the ‘experience’ of the DQN, it could be difficult for this model to gain enough experience. Thus pre-calculation of rewards may improve the breadth of experience of the model, for this reason a regression model is proposed to predict the Q-value of a given action.

5.3.3 Regression using Q-Values

Unlike classification problems, regression problems can predict a continuous value rather than discrete (or probabilitic) values relating to a set of output classes. Regression algorithms are proposed here to predict the q-value of killing a process. If this value is positive, the process is killed.

Q-values estimate the value of a particular action based on the ‘experience’ of the agent. Since the optimal action for the agent is always known, it is possible to pre-compute the ‘(q-)value’ of killing a process and train various ML models to learn this value. For a complex problem space, for which a neural network is required, many resources and iterations are required, this approach investigates training a regression model, which is quicker and less resource intensive and mimics the DQN by trying to learn the value of killing a process than to train a DQN which explores the state-action space and calculates rewards between learning, since the interaction and calculation of rewards is no longer necessary. The regression approach can be used with any machine learning algorithm capable of learning a regression problem, regardless of whether it is capable of partial training.

There are two primary differences between this regression approach and the reinforcement learning DQN approach detailed in the previous section. Firstly, the datasets are likely to be difference. Since the DQN generates training data through interacting with its environment it may never see certain parts of the state-action space e.g. if a particular process A is always killed during training before time $t*$, the model is not able to learn from the process A data after $t*$.

Secondly, *only* the expected value of killing is modelled by the regressor, whereas the DQN tries to predict the value of both killing and of not killing the process. This means that the equation used to model the value of process killing is only an approximation of the reward function used by the DQN.

The equation used to calculate the value of killing is positive for malware and negative for benignware, in both cases it is scaled by the number of child processes impacted and in the case of malware, early detection increases the value of process killing (with an exponential decay). Let y be the true label of the process (0=benign, 1=malicious), N is the number of child processes and t is the time in seconds at which the process is killed then the value of killing a process is:

$$(y * 2 - 1) * (1 + N) * (1 + (y * (e^{-t})))$$

The equation above negatively scores the killing of benignware in proportion to the number of subprocesses and scores the killing of malware positively in proportion to the number of subprocesses. A bonus reward is scored for killing malware early, with an exponential decay over time. Killing malware may be said to be beneficial to the extent that it prevents damage but killing benignware (false positives) is highly dependent on context, different applications will have different value to different people at different times of day. Since it is difficult to model this without making some significant assumptions and likely biasing the model towards or away from killing, the malware and benignware are weighted equally with additional value assigned for killing malware early in accordance with the aforementioned assumptions.

5.4 Evaluation Methodology: Ransomware detection

As noted in the background chapter, to date research has not addressed the extent to which damage is mitigated by process killing, since Sun et al. [188] presented the only previous work to test process killing and damage with and without process killing is not assessed. To this end, this chapter uses ransomware as a proxy to detect malicious damage, inspired by Scaife et. al's approach [172]. A brief overview of Scaife et al.'s damage measurement is outlined below:

Early detection is particularly useful for types of malware from which recovery is difficult and/or costly. Cryptographic ransomware encrypts user files and withholds the decryption key until a ransom is paid to the attackers. This type of attack is costly to remedy, even if the victim is able to carry out data recovery [185]. Scaife et al.'s work [172] on ransomware detection uses features from file system data, such as whether the contents appears to have been encrypted, and number of changes made to the file type. The authors were able to detect and block all of the 492 ransomware samples tested with less than 33% of user data being lost in each instance. Continella et al. [40] propose a self-healing system, which detects malware using file system machine activity (such as read/write file counts), the authors were able to detect all 305 ransomware samples tested, with a very low false-positive rate. These two approaches use features selected specifically for their ability to detect ransomware, but this requires knowledge of how the malware operates. Whereas the approach taken here seeks to use features which can be used to detect malware *in general*. The key purpose of this final experiment (section 5.7) is to show that our general model of malware detection is able to detect general types of malware as well as time-critical samples such as ransomware.

5.5 Experimental design

This section outlines the data collection process and experimental setup.

5.5.1 Features

The same features as were used in Chapter 4 are used here for process detection, with some additional features to measure process-specific data. Despite the popularity of API calls noted in chapter 2, due to the findings in Chapter 3 and Sun et al.'s [188] difficulties hooking this data in real-time, these were not considered as features to train the model.

Category			
CPU use (%)	system level	user level	
Memory use (bytes)	total	physical (non-swapped)	swap
Child processes	count	maximum process ID	number of threads
I/O operation bytes on disk (bytes)	read	write	non-read-write I/O operations
I/O operation count on disk	read	write	non-read-write I/O operations
Priority	process priority	I/O process priority	
Network # Packets	TCP packet count	UDP packet count	
Network # Bytes	# bytes sent	# bytes received	
Network Other	number of connections currently open	statuses of the ports opened by the process (4 statuses)	
Miscellaneous	number of command line arguments passed to process	number of handles being used by process	

Table 5.1: 26 process-level features: 22 features + 4 port status values

At the process-level, 26 machine metric features are collected, these were dictated by the attributes available using the Psutil [63] python library. It is also possible to include the 'global' machine learning metrics that were used in the previous chapters. Though global metrics will not provide process-level granularity, they may give muffled indications of the activity of a wider process tree. The 9 global metrics are: system level CPU use, user level CPU use, memory use, swap memory use, number of packets received and sent, number of bytes received and sent and the total number of processes running.

The process-level machine activity metrics collected are: CPU use at the user level, CPU use at the system level, physical memory use, swap memory use, total memory use, number of child process, number of threads, maximum process ID from a child process, disk read, write and other I/O count, bytes read, written and used in other I/O processes, process priority, I/O process priority, number of command line arguments passed to process, number of handles being used by process, time since the process began, TCP packet count, UDP packet count, number of connections currently open, 4 port statuses of those opened by the process (see Table 5.1).

Preprocessing

Feature normalisation is used to aid convergence of NNs. The test, train and validation sets (x) are all normalised by subtracting the mean (μ) and dividing by the standard deviation (σ) of each feature in the training set: $\frac{x-\mu}{\sigma}$. This sets the mean of the features to 0 and the standard deviation to 1 for all input features, avoiding the potential for some features to be weighted more important than others during training purely due to the scalar values of those features. This requires additional computational resources but is not necessary for all ML algorithms; this is another reason why the supervised RNN used in chapter 4 may not be well-suited for real-time detection.

5.5.2 Data Capture

During data capture, this research sought to improve upon previous work and emulate real machine use to a greater extent than has previously been trialled. The implementation details of the VM, simultaneous process execution and RL simulation are outlined below:

Environment: Machine setup

The following experiments were conducted using a virtual machine (VM) running with Cuckoo Sandbox [77] for ease of collecting data and restarting between experiments and because the Cuckoo Sandbox emulates human interaction with programs to some extent to promote software activity. In order to emulate the capabilities of a typical machine, the modal hardware attributes of the top 10 'best seller' laptops according to a popular internet vendor [9] were used, and these attributes were the basis of the VM configuration. This resulted in a VM with 4GB RAM, 128GB storage and dual-core processing running Windows 7 64-bit. Windows 7 was the most prevalent computer operating system (OS) globally at the time of designing the experiment [68], though Windows 10 is now the most popular OS, the findings in this research should still be relevant.

Simultaneous applications

In typical machine use, multiple applications run simultaneously. This is not reflected by behavioural malware analysis research in which samples are injected individually to a virtual machine for observation. The environment used for the following experiments launches multiple applications on the same machine at slightly staggered intervals as if a user were opening them. Each malware is launched with a small number (1-3) and a larger number (3-35) of applications. It was not possible to find find up-to-date user data on the number of simultaneous applications running on a typical desktop, so here it was elected to launch up to 36 applications (35 benign + 1 malicious) at once, which is the largest number of simultaneous apps for real-time data collection to date. From the existing real-time analysis literature only Sun et al. [188] run multiple applications at the same time, with a maximum of 5 running simultaneously.

Each application may in turn launch multiple processes, causing more than 35 processes to run at once; 95 is the largest number of simultaneous processes recorded, this excludes background OS processes.

The results later in the chapter reveal a non-linear time impact on the number of processes for analysis; probably to to the varied resource consumption of the programs but illustrate a linear increase in monitoring and killing delay, which is to be expected since both of these programs have $O(n)$ complexity

Reinforcement Learning Simulation

For reinforcement learning, the DQN requires an observation of the resulting state following an action. To train the model, a simulated environment is created from the pre-collected training data whereby the impact of killing or not killing a process is returned as the next state. For process-level elements this reduces all features to zero. A caveat here is that in reality killing the process may not occur immediately and therefore memory, processing power etc. may still be being consumed at the next data observation. For global metrics, the process-level values for the killed processes (includes child processes of the killed process) are subtracted from the global metrics. There is a risk again that this calculation may not correlate perfectly with what would be observed in a live machine environment.

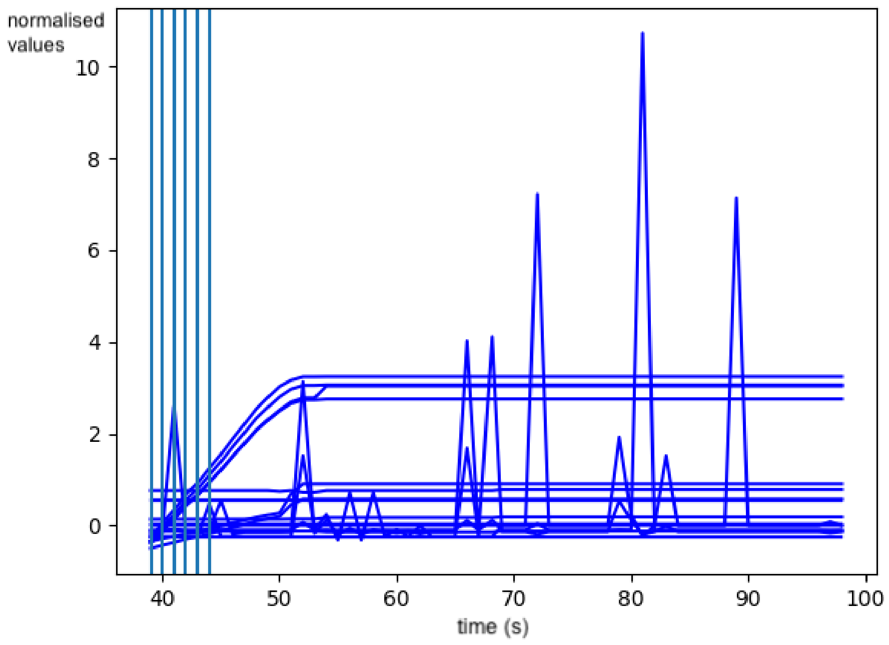


Figure 5.3: Benignware sample, normalised process-level metrics 6 observations made without process being killed

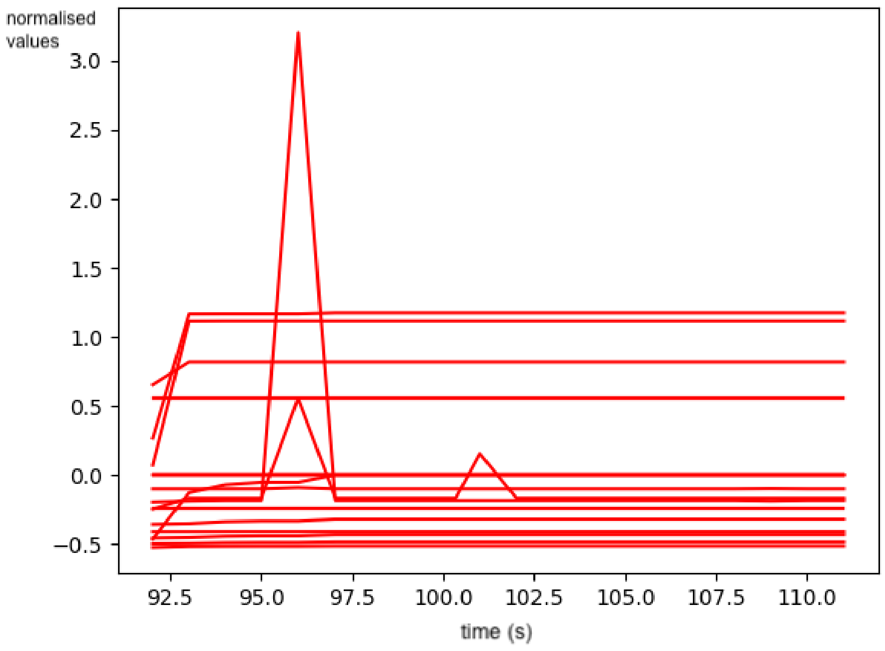


Figure 5.4: Malware sample, no observations made yet

In order to observe the model performance, a visualisation was developed to accompany the simulated environment. Figure 5.4 shows a screenshot of the environment visualisation for one malicious and one benign process.

5.6 Experimental Results

5.6.1 Dataset

The dataset is comprised of 3,604 benign executables and 2,792 malicious applications (each containing at least one executable), with 2,877 for training and validation and 3,519 for testing. These dataset sizes are consistent with previous real-time detection dataset sizes e.g. (Das et al. [48] use 168 malicious, 370 benign; Sayadi et al. [171] use over 100 each benign and malicious; Ozsoy et al. [139] use 1,087 malicious and 467 benign; Sun et al. [188] use 9,115 malicious, 877 benign). With multiple samples running concurrently to simulate real endpoint use, there are 24K processes in the training set and 34K in the test set. Overall there are 58K behavioural traces of processes in the training and testing datasets. The benign samples comprise files from VirusTotal [152], from free software websites (later verified as benign with VirusTotal), and from a fresh Microsoft Windows 7 installation. The malicious samples were collected from two different VirusShare [201] repositories.

In Pendelbury et al's analysis [150], the authors estimate that in the wild between 6% and 22% of applications are malicious, normalising to 10% for their experiments. Using this estimation of Android malware, a similar ratio was used in the test set in which 13.5% were malicious.

Malware families

This chapter is not concerned with distinguishing particular malware families, but rather with identifying malware in general. However, a dataset consisting of just one malware

Malware family	# Train set	# Test set	Total	Description
startsurf	66	273	339	adware
fareit	33	222	255	spyware
vigram	23	212	235	adware
winwrapper	78	8	86	PUA
downloadguide	15	59	74	adware
gandcrab	5	54	59	ransomware
emotet	12	46	58	credstealer
chapak	4	37	41	installer
virut	30	2	32	backdoor
installmonster	12	18	30	installer
noon	8	22	30	spyware
gamarue	11	18	29	backdoor
razy	7	16	23	crypto stealer
zeroaccess	23	0	23	rootkit
soft32downloader	5	22	23	installer
appster	7	15	22	PUA
prepscram	1	20	21	installer
zusy	2	19	21	spyware
darkkomet	17	1	18	RAT
adposhel	4	14	16	adware
swrort	13	0	13	backdoor
slugin	13	0	13	installer
vobfus	11	2	13	installer
speedingupmypc	1	11	12	adware
relevantknowledge	5	6	11	adware
kuaizip	4	7	11	PUA
bladabindi	7	4	11	backdoor
Other ($\leq$ 10 instances)	377	260	602	-
# Other families ($\leq$ 10 instances)	184	154	288	-
Unknown	333	291	671	-
Total	**1,137**	**1,655**	**2,792**	-

Table 5.2: Malware families with more than 10 samples in the dataset. 315 families were represented in the dataset, with 27 having being represented more than 10 times. Basic description provided which does not cover the wide range of behaviours carried out by some malware families but is intended to indicate the range of behaviours in the top 27 families included in the dataset. PUA = potentially unwanted application, RAT = remote access trojan

family would present an unrealistic and easier problem than is found in the real-world. The malware families included in this dataset are reported in Table 5.2. The malware family labels are derived from the output of around 60 antivirus engines used by VirusTotal [152].

Ascribing family labels to malware is non-trivial since antivirus vendors do not follow standardised naming conventions and many malware families have multiple aliases. Sebastián et al. [174] have developed an open source tool, AVClass, to extract meaningful labels and correlate aliases between different antivirus outputs. AVClass was used to label the malware in this dataset. Sometimes there is no consensus amongst the antivirus' output or the sample is not recognised as a member of an existing family. AVClass also excludes malware that belongs to very broad classes of malware (e.g. “agent”, “eldorado”, “artemis”) as these are likely to comprise a wide range of behaviours and so may be applied as a default label in cases for which antivirus engines are unsure. In the dataset established in this research, 2,121 of the 2,792 samples were assigned to a malware family. Table 5.2 gives the number of samples in each family for which there were more than 10 instances found in the dataset. 315 families were detected overall, with 27 families being represented more than 10 times. These better-represented families persist in the train and test sets, but the other families have little overlap. 104 of the 154 other families seen in the test set are not identified by AVClass as being in the training set.

Malicious vs. Benign Behaviour

Statistical inspection of the training set reveals that benign applications have fewer subprocesses than malicious processes, with 1.17 processes in the average benign process tree and 2.33 processes in the average malicious process tree. Malware was also more likely to spawn processes outside of the process tree of the root process, often using the names of legitimate Windows processes. In some cases malware launches legitimate applications, such as Microsoft Excel in order to carry out a macro-based exploit. Although Excel is not a malicious application in itself, it is malicious in this context, which is why malicious labels are assigned if a malware sample has caused that process to come into being. It is therefore possible to argue that some processes launched by malware are not malicious, because they do not individually cause harm to the endpoint or user, but without the malware they would not be running and so can be considered at least undesirable even if only in the interests of conserving computational resources.

Train-Test Split

The dataset is split in half with the malicious samples in the test set coming from the more recent VirusShare repository, and those in the training set from the earlier repository. This is to increase the chances of simulating a real deployment scenario in which the malware tested contain new functionality by comparison with those in the training set. Ideally, the benignware should also be split by date across the training and test set, however it is not a trivial task to calculate the date at which benignware was compiled. It is possible to extract the compile time from PE header, but it is possible for the PE author to manually input this date which had clearly happened in some instances where the compile date was 1970-01-01 or in one instance 1970-01-16. In the latter case (1970-01-16), the file is first mentioned online in 2016, perhaps indicating a typographic error [182]. Using internet sources such as VirusTotal [152] can give an indication when software was first seen but if the file is not very suspicious i.e. from a reputable source, it may not have been uploaded until years after it was first seen “in the wild’. Due to the difficulty in dating benignware in the dataset collected for this research, samples were assigned to the training or test set randomly.

For training, an equal number of benign and malicious processes are selected, so that the model does not bias towards one class. 10% of these are held out for validation. In most ML model evaluations, the validation set would be drawn from the same distribution as the test set. However, because it is important not to leak any information about the malware in the test set, since it is split by date, the validation set here is drawn from the training distribution.

Implementation Tools

Data collection used the Psutil [63] Python library to collect machine activity data for running processes and to kill those processes deemed malicious. The RNN and Random Forests were implemented using the Pytorch [146] and Scikit-Learn [148] python libraries respectively. The model runs with high priority and administrator rights to make sure the polling is maintained when compute resources are scarce.

5.6.2 Hyperparameter Search for Process-Level Supervised RNN

In this first set of experiments benchmarks the supervised learning models and demonstrate their unsuitability to process killing.

In the previous chapter, an RNN was used to distinguish malicious and benign software in the early stages of execution with 96% accuracy in the first 20 seconds and 94% in the first 5 seconds. Real-time detection necessitates detection at the process-level with multiple applications, benign and malicious, running simultaneously.

A random search of the hyperparameter space explores the best-performing architecture on 2 feature sets: process-only data (26 features) and process-level + global metrics (37 features).

The hyperparameters (see Table 5.3) are as follows: hidden neurons refers to the number of GRU cells in each hidden layer; the depth refers to the number of hidden layers in the neural network; the batch size indicates the number of samples predicted by the model between weight updates and the epochs designate the number of times the model will be exposed to the entire dataset; the dropout rate is a regularisation technique first proposed by Srivastava et al. [187] to reduce the chance of model overfitting, the rate is the proportion of neurons randomly turned ‘off’ (set to an activation of 0) during training; the window size gives the length of historical data for each process visible to the model. For example a window size of 5 will allow the model to see the current set of features and the features at the 4 previous time-steps. The data used

	possible values	process level data	process level data + global metrics
Hyperparamter			
hidden neurons	8 - 1024	253	193
depth	[1, 2, 3]	2	2
batch size	[64, 128, 256]	128	256
epochs	1 - 200	89	161
dropout rate	[0, 0.1, 0.2, 0.3, 0.4, 0.5]	0.1	0.1
window size	2 - 30	16	6
loss function	binary cross-entropy		
weight update rule	Adam [102]		
recurrent unit	GRU cell		
validation F1-score (*100)	-	**97.43**	94.61

Table 5.3: Hyperparameter search space and the hyperparameters of the model giving the lowest mean false-positive and false-negative rates

here is captured every second such that a window size of 5 corresponds to the model being able to see the machine activity metrics from 4, 3, 2, and 1 seconds ago as well as the current metrics. For reproducibility, the loss function used to train the model was binary-cross-entropy and the weight update algorithm used was Adam [102].

The winning hyperparameter configurations and search space are detailed in table 5.3 and the search space is illustrated with respect to validation set F1-score in Figure 5.5. From the graphs in Figure 5.5, it is clear that many models were able to accurately predict the validation set. There is evidence of a slight positive but plateauing correlation between sequence length and F1-score, as in the previous chapter when looking at global machine metrics. It is also evident from the right-most graph that adding the global metrics tends to lead to worse models (lower F1-scores) but adding just the global number of processes together with the process-level data gives a slightly higher average F1-score. It should be noted that these plots do not show the marginal contribution of each hyperparameter such that the interplay between epochs and batch size cannot be seen. There are fewer models with 2 or 3 hidden layers as these configurations were more likely to incur an out of memory error.

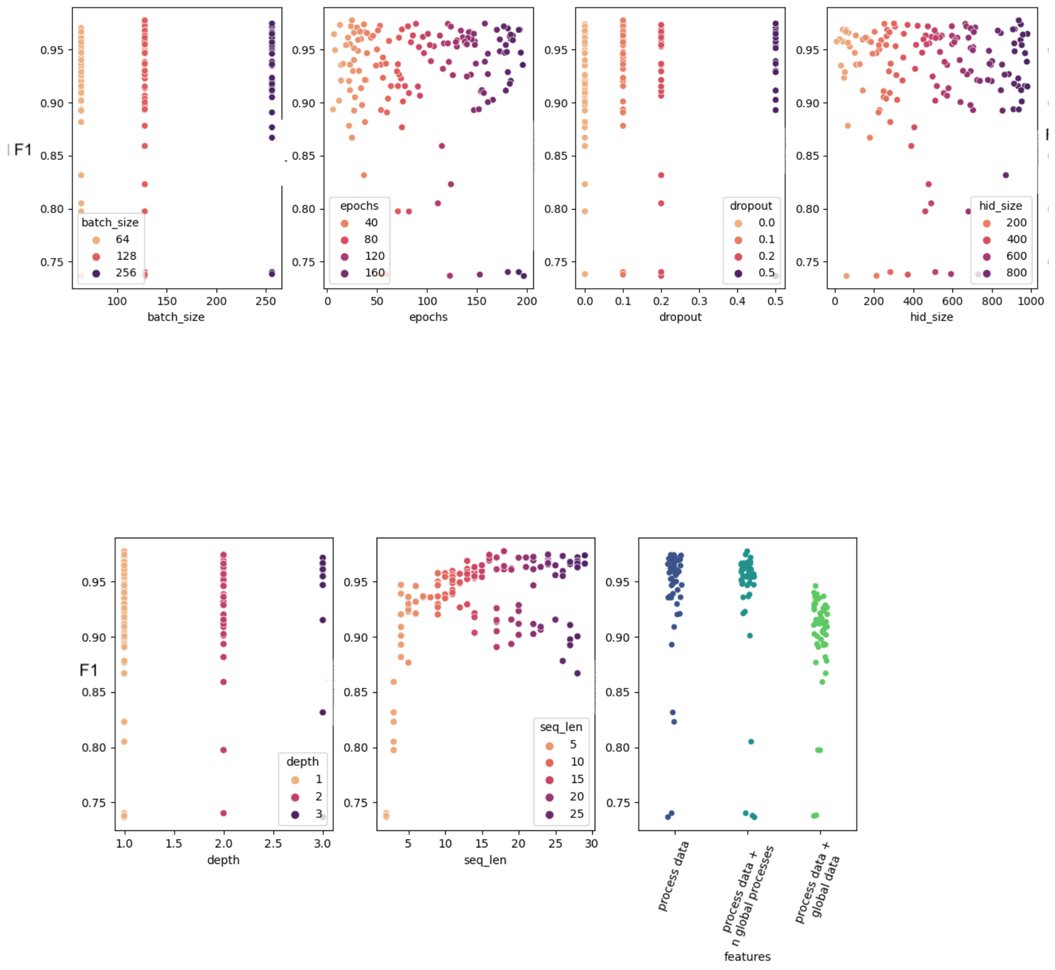


Figure 5.5: Validation set F1-score variance with hyperparameter values and feature sets

5.6.3 Supervised Learning for Process Killing

It is expected that supervised malware detection models will not adapt well to process-killing due to the averaging of loss metrics as described earlier. Initially this is verified by using supervised learning models to kill processes that are deemed malicious. For supervised classification, the model makes a prediction every time a data measurement is taken from a process. This approach is compared with one taking average predictions across all measurements for a process and for a process tree as well as the result of process killing. The models with the highest validation accuracy for classification and killing are compared.

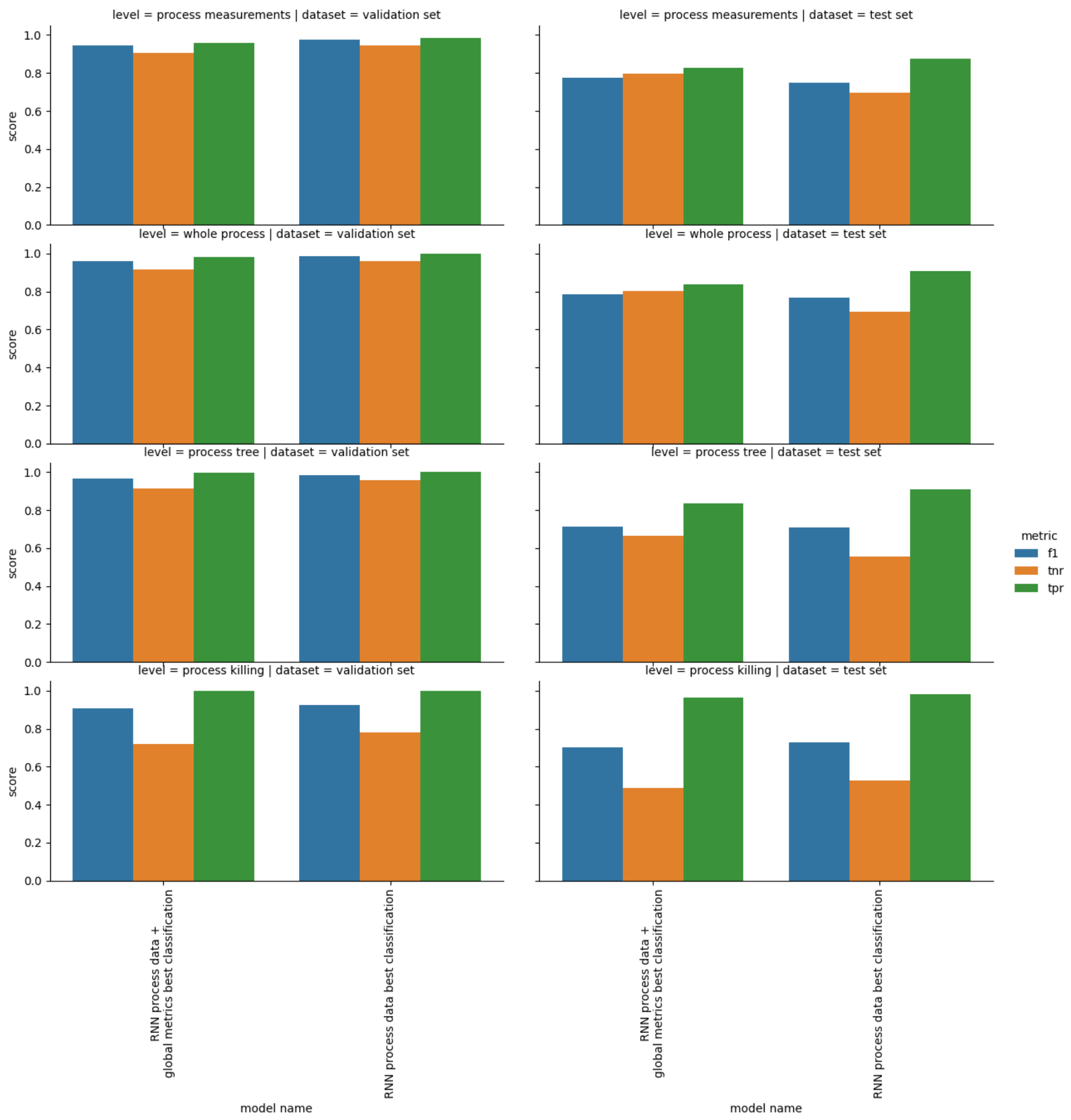


Figure 5.6: F1 scores, true positive rates (TPR) and true negative rates (TNR) for partial-trace detection (process measurements), full-trace detection (whole process), whole application (process tree) and with process level measurements + process killing (process killing) for validation set (left column) and test set (right column)

features	metric	classify	dataset	kill
proc. data	F1	**97.44**	validation set	**91.20**
proc. data	tnr	94.72	validation set	85.71
proc. data	tpr	98.64	validation set	95.80
proc. data + glob.	F1	94.61	validation set	87.69
proc. data + glob.	tnr	90.57	validation set	77.31
proc. data + glob.	tpr	95.93	validation set	95.80
proc. data	F1	74.91	test set	**72.63**
proc. data	tnr	69.41	test set	59.63
proc. data	tpr	87.52	test set	91.82
proc. data + glob.	F1	**77.66**	test set	71.83
proc. data + glob.	tnr	79.70	test set	59.63
proc. data + glob.	tpr	82.91	test set	90.24

Table 5.4: F1-score, true positive rate (TPR) and true negative rates (TNR) (all * 100) on test and validation sets for classification and process killing

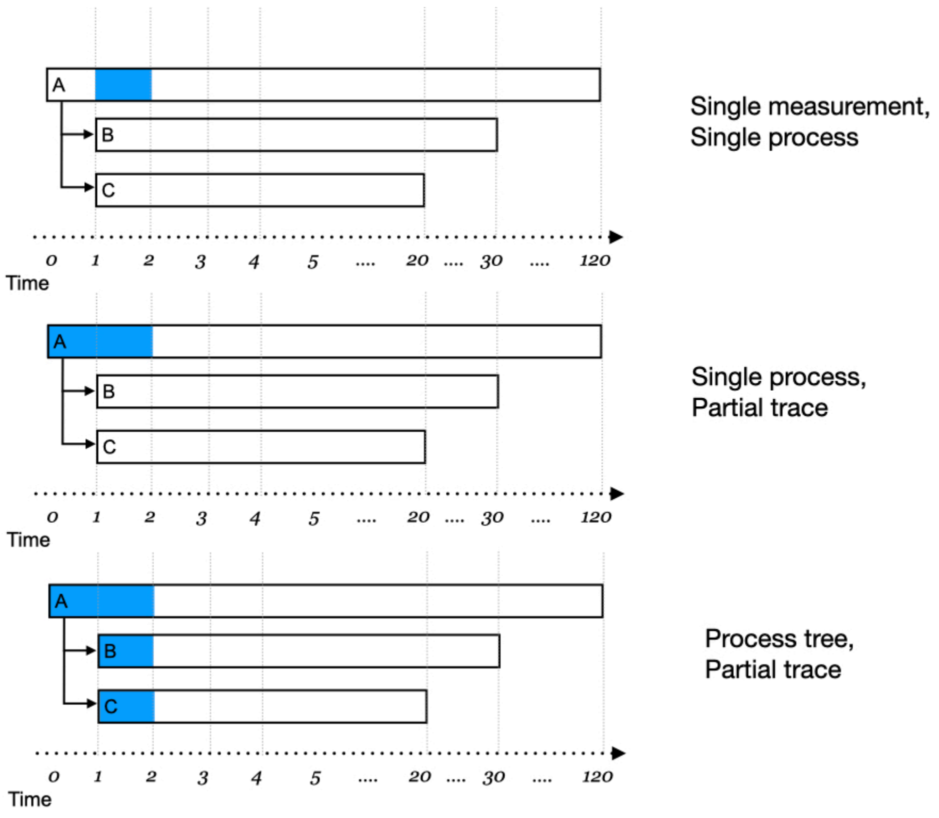


Figure 5.7: 3 levels of data collection: Each measurement, each process, each process tree

Figure 5.6 illustrates the difference in validation set and test set F1-score, true positive rate and false positive rate for these 4 levels of classification: each measurement, each process, each process tree, and finally showing process killing; see Figures 5.7 for diagrammatic representation of these first 3 levels. Table 5.4 reports the F1, TPR and TNR for classification (each measurement of each process) and for process killing.

The highest F1-score on the validation set is achieved by an RNN using process data only. When process killing is applied there is a drop of less than 5 percentage points in the F1-score but more than 15 percentage points are lost from the TNR.

On the unseen test set the highest F1-score is achieved by an RNN using process data + global metrics, but the improvement over the process data + total number of processes is negligible. Overall is there is a reduction in F1-score from (97.44, 94.61) to (74.91, 77.66), highlighting the initial challenge of learning to classifying individual processes rather than entire applications, especially when accounting for concept drift. Despite the low accuracy, these initial results indicate that the model is discriminating some of the samples correctly and may form a baseline from which to improve.

The test set TNR and TPR for classification on the best-performing model (process data only) are 79.70 and 82.91 respectively, but when process killing is applied, though the F1-score drops by 10 percentage points, the TNR and TPR move in opposite directions with the TNR falling to 59.63 and TPR increasing to 90.24. This is not surprising since a single malicious classification results in a process being classed as malicious. This is true for the best-performing models using either of the two feature sets (see Fig.5.6 above).

5.6.4 Accuracy vs. Resource consumption

Previous work on real-time detection has highlighted the requirement for a lightweight model (speed and computational resources). The previous chapter, RNNs were the best performing algorithm in classifying malware/benignware but RNNs have many parameters and therefore may consume significant RAM and/or CPU, they also require preprocessing of the data to scale the values, which other ML algorithms such as tree-based algorithms do not. Whilst RAM and CPU should be minimised, taking model accuracy into account, inference duration is also an important metric.

Though the models in this chapter have not been coded for performance and use common python libraries, comparing these metrics helps to decide whether certain models are vastly preferable to others with respect to computational resource consumption. The PyRAPL library [17] is used measure the CPU, RAM and duration used by each model. This library uses Intel processor ‘Running Average Power Limit’ (RAPL) metrics. Only the data pre-processing and inference is measured as training may be conducted centrally in a resource-rich environment. Batch sizes of 1, 10, 100 and 1000 samples are tested with 26 and 37 features respectively since there are 26 process-level features and 37 when global metrics are included. Each model is run 100 times for each of the different batch sizes.

For the RNN a ‘large’ and a ‘small’ model are included. The large models have the highest number of parameters tested in the random search (981 hidden neurons, 3 hidden layers, sequence length of 17) and the smallest (41 neurons, 1 hidden layer, sequence length of 13). These two RNN configurations are compared against other machine learning models which have been used for malware detection: Multi-Layer Perceptron (feed-forward neural network), Support Vector Machine, Naive Bayes Classifier, Decision Tree Classifier, Gradient Boosted Decision Tree Classifier (GBDTs), Random Forest and AdaBoost.

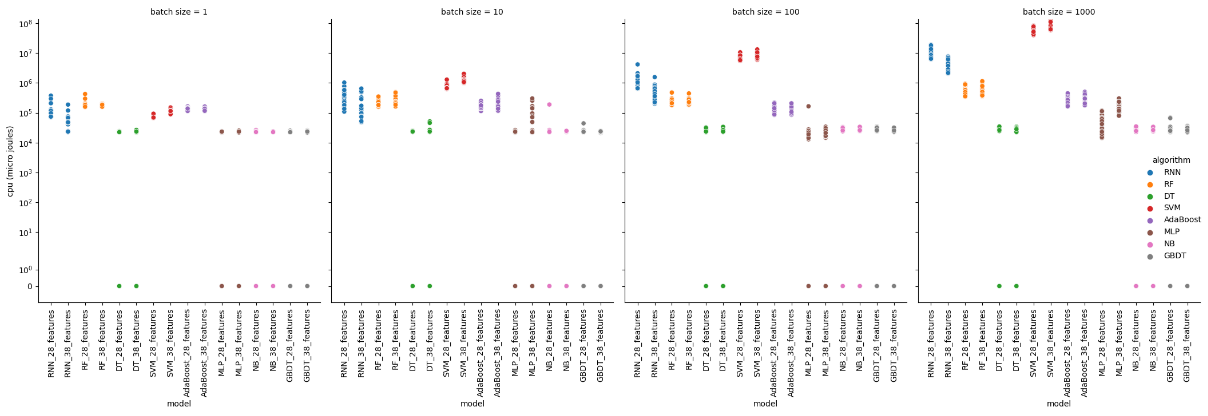


Figure 5.8: CPU use of different trained models

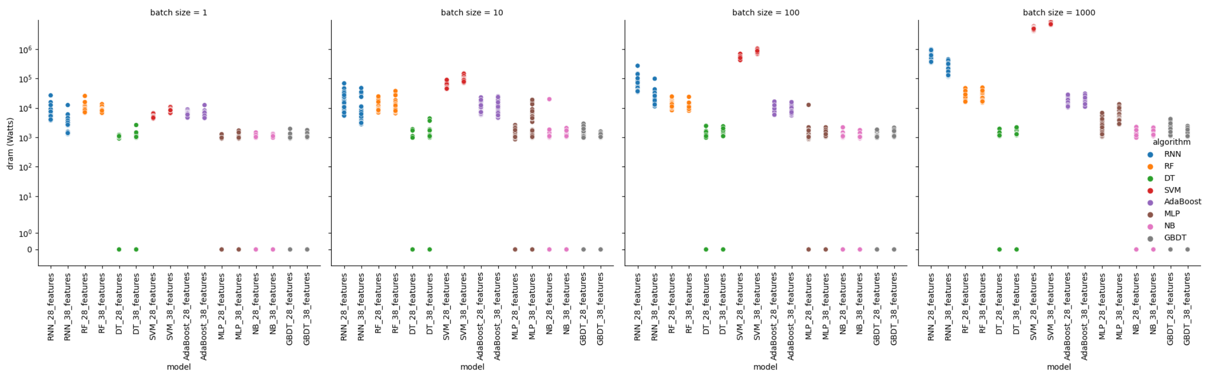


Figure 5.9: RAM use of different trained models

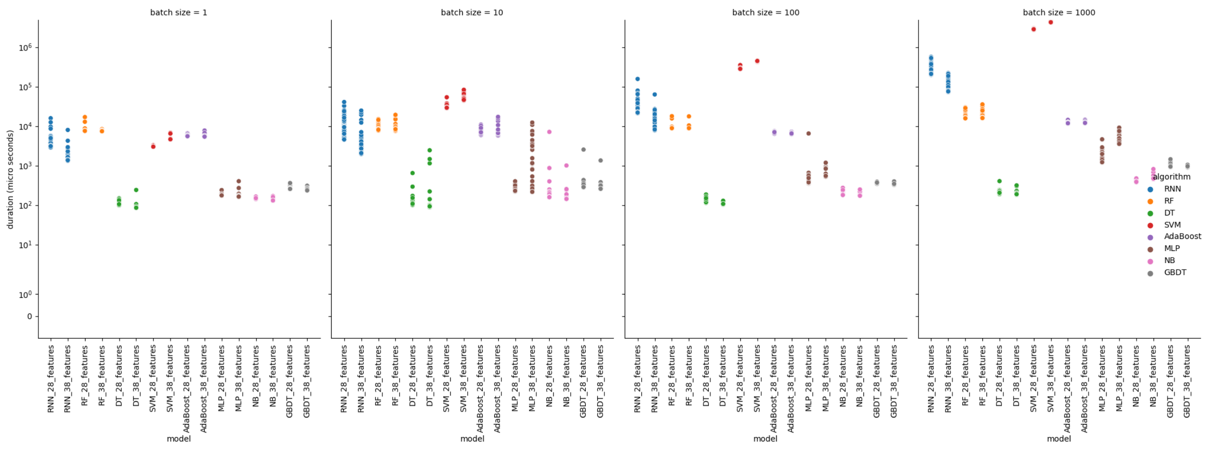


Figure 5.10: Inference duration of different trained models

Figures 5.8, 5.9, 5.10 show the varied CPU, RAM and inference time respectively for 7 algorithms with different numbers of features. The RNNs, random forests and SVMs perform worst across all three metrics. SVM is the only algorithm to breach the 1-second mark for inference duration (10e6 micro seconds). Decision Trees, GBDTs and Naive Bayes perform best. AdaBoost and MLP have middling performance. The RNN sees a marked difference in performance between the 'large' and 'small' networks but the number of features has a greater impact that the size of the network (although number of features impacts the network size too). Only MLP (also a neural network) and SVM show a consistently observable observable difference between using 26 and 37 features, but for most algorithms, the number of features does not significantly impact upon performance metrics. Sun et al. [188] used an RF as a lightweight initial processor whereas these experiments do not find it particularly lightweight, however it may be by comparison with the NN 'heavyweight' model used.

Low resource consumption is of little consequence if the model is not able to learn to distinguish benign and malicious processes. Focusing on batch-sizes of 100 (since 95 is the largest number of simultaneous processes observed in this dataset), the detection accuracy, process killing accuracy and performance metrics can be compared.

Table 5.5 reports the computational resource consumption and accuracy metrics together. Decision tree with 38 features is the lowest cost to run, RNN performs best at supervised learning classification on the validation set but only just outperforms the decision tree with 26 features, which is the best performing model at process killing on the validation set at 92.97 F1-score. The highest F1-score for process killing uses a Random Forest with 37 features, scoring 77.85 F1, which is 2 percentage points higher than the RF with 26 features (75.97). The models all perform at least 10 percentage points better on the validation set indicating the importance of taking concept drift into account when validating models.

model	n features	avg. cpu (μJ)	avg. dram (W)	avg. duration (μs)	val F1	kill val F1	test F1	kill test F1
AdaBoost	26	127967.84	7981.51	6595.37	88.35	74.36	77.19	60.09
AdaBoost	37	125041.20	7142.93	6469.16	89.63	76.07	80.10	60.14
DT	26	3905.63	202.65	128.02	97.39	88.48	66.44	62.95
DT	37	**2113.67**	**134.29**	**106.65**	96.32	83.57	79.61	62.50
GBDT	26	8788.41	338.78	349.31	92.27	78.26	82.47	63.33
GBDT	37	11005.80	486.46	329.45	93.13	80.26	84.94	63.46
MLP	26	11044.88	645.14	461.04	82.84	70.18	41.62	57.65
MLP	37	12932.09	628.64	555.42	73.00	67.63	57.66	57.26
NB	26	6947.67	297.87	185.73	75.80	67.42	62.90	56.11
NB	37	5187.96	258.80	177.37	75.58	67.61	61.88	55.33
RF	26	238621.20	11052.84	8997.31	97.12	**92.97**	71.58	75.97
RF	37	236598.44	9967.63	8879.97	96.57	91.05	**85.55**	**77.85**
RNN	26	887664.31	48885.96	27869.30	**97.44**	90.70	74.91	73.08
RNN	37	312108.07	17120.90	10414.58	94.61	87.31	77.66	71.95
SVM	26	6630490.84	464082.07	282026.57	78.34	67.04	68.16	56.91
SVM	37	7792179.78	730786.06	429081.31	64.89	65.68	61.39	56.25

Table 5.5: Average resource consumption over 100 iterations for a batch size of 100 vs. F1-scores on validation and test set for classification and process killing across 14 models. 26 features = process-level only, 37 features = machine and process level features.

5.6.5 How to solve a problem like process killing?

From the results above, it is clear that supervised learning models see a significant drop in classification accuracy when processes are killed as the result of a malicious label. This confirmation of the initial hypothesis presented here justifies the need to examine alternative methods. In the interests of future work and negative result reporting this chapter reports all of the methods attempted and finds that simple statistical manipulations on the supervised learning models perform better than using alternative training methods. This section briefly describes the logic of each method and provides a textual summary of the results with a formulae where appropriate. This is followed by a table of the numerical results for each method. In the following section let P be a set of processes $\{p_0, p_1...p_P\}$ in a process tree, let $t*$ be the time at which a prediction is made and let $\hat{y}_i$ be the prediction for process i at time t* where a prediction equal to or greater than 1 classifies malware.

a) Mean predictions

Reasoning: Taking the average prediction across the whole process will smooth out those process killing results

Not tested This was not attempted for two reasons: (1) Taking the mean at the end of the process means the damage is done (2) This method can easily be manipulated by an attacker: 50 seconds of injected benign activity required 50 seconds of malicious activity to achieve a true positive

$$\hat{y}_i = \frac{1}{t*} \sum_{t=0}^{t*} \hat{y_i^t}$$

b) Rolling mean predictions

Reasoning: Taking the average over a few measurements will eliminate those false positives that are caused by a single false positive over a subset of the execution trace. Window sizes of 2 to 5 are tested. Let w be the window size:

$$\hat{y}_i = \frac{1}{w} \sum_{t=t*-w}^{t*} \hat{y_i^t}$$

Summary of results: A small but unilateral increase in F1-Score using a moving window over 2 measurements on the validation set. Using a moving window of size 2 on the test-set saw a 10 to 20 percentage point increase in true negative rate (to a maximum of 80.77) with 3 percentage points lost from the true positive rate. This was one of the most promising approaches.

c) Alert threshold

Reasoning: Like the rolling mean, single false positives will be eliminated but unlike the rolling mean, the alerts are cumulative over the entire trace such that a single alert at the start and 30 seconds into the process will cause the process to be killed rather than requiring that both alerts are within a window of time. Between 2 and 5 minimum alerts are tested

$$\hat{y}_i = w - \sum_{t=0}^{t*} \hat{y_i^t}$$

Summary of results: Again a small increase across all models, with an optimal minimum number of alerts being 2 for maximum F1-score, competitive with the rolling mean approach.

d) Process-tree averaging

Reasoning: the data are labelled at the application level, therefore the average predictions across the process tree should be considered for classification

$$\hat{y}_i = \frac{1}{P} \sum_{p=0}^{P} \hat{y_i^{t*}}$$

Summary of results: Negligible performance increase on validation and test set data (<1 percentage point). This is likely because few samples have more than one process executing simultaneously.

e) Process-tree training

Reasoning: the data are labelled at the application level, therefore the sum of resources of each process tree should be classified at each measurement, not the individual processes

Summary of results: Somewhat surprisingly there was a slight reduction in classification accuracy when using process tree data. One explanation for this may be that the process tree creates noise around the differentiating characteristics that are visible at the process level.

DQN

Reinforcement learning is designed for state-action space learning. Both pre-training the model with a supervised learning approach and not pre-training the model were *tested.*

Summary of results: Poor performance, typically converging to either kill or not kill everything, of the few models which did not converge to a single dominant action, it does not distinguish malware or benignware well, indicating that it may not have learned anything. Reinforcement learning may help the problem of real-time malware detection and process killing but this initial implementation of a DQN did not converge to a better or even competitive solution to supervised learning. Perhaps better formulation of rewards (e.g. damage prevented) would help the agent learn.

Regression on predicted kill value

Reasoning: Though the DQN explores and exploits different state-action pairs and their associated rewards, when the reward from each action is known in the first place and the training set is limited, as it is here, Q-learning can be framed as a regression problem in which the model tries to learn the return (rewards + future rewards), the training is faster and can be used by any regression-capable algorithm. Let N be the number of current and future child processes for p_i at $t*$

$$(y * 2 - 1) * (1 + N) * (1 + (y * (e^{-t*})))$$

Summary of results: Improved performance on true negative rate, though not perceptible for the highest-scoring F1 models since F1-scores reward true positives more than true negatives, this metric can struggle to reflect a balance between the true positive and true negative rates. The highest true negative rate models are all regression models.

Table 5.6 lists the F1, TPR and TNR on the validation and test set for each of the methods described above. The best-performing model on the test and validation sets are reported and the full results can be found in Appendix Table 2. Small improvements are made by some models on the validation F1-score but the test set F1-score improves by 4 percentage points in the best instance.

				val			test		
methodology	best dataset	model	n features	F1	tnr	tpr	F1	tnr	tpr
Supervised learning	val	RF	26	92.37	87.39	96.64	74.57	62.71	92.95
	test	RF	37	89.68	83.19	94.96	76.43	67.19	92.52
Rolling mean	val	RF (min: 2)	26	**93.22**	94.12	92.44	78.26	73.83	89.76
	test	RF (min: 2)	37	92.70	94.96	90.76	80.77	78.88	89.38
Alert threshold	val	DT (min: 2)	26	92.17	95.80	89.08	73.43	67.44	86.56
	test	RF (min: 2)	37	91.30	94.96	88.24	**81.50**	81.53	87.97
Process tree averaging	val	RF	26	92.74	88.24	96.64	74.79	64.04	92.20
	test	RF	37	90.48	84.03	95.80	76.34	67.66	91.92
Process tree training	val	RF	26	90.35	82.58	98.32	74.20	52.44	92.74
	test	RF	26	90.35	82.58	98.32	74.20	52.44	92.74
Q-learning	val	DQN	26	51.71	72.27	44.54	27.74	55.50	26.94
	test	DQN	26	51.71	72.27	44.54	27.74	55.50	26.94
Regression	val	RF	26	91.94	87.39	95.80	74.77	66.05	90.35
	test	RF	26	91.94	87.39	95.80	74.77	66.05	90.35

Table 5.6: Summary of best process killing models by model training methodology. F1, TNR and TPR for validation and test datasets. Full results in Appendix Table 2.

In most cases, the models with the highest F1-score on the validation and test sets are not the same. The highest F1-score is 81.50 from an RF using a minimum alert threshold of 2 and both process-level and global process metrics.

5.6.6 Further experiment: Favouring high TNR

Though the proposed model is motivated by the desire to prevent malware from executing, the best TNR reported amongst the models above is 81.50%. 20% of benign processes being killed would not be acceptable to a user. Whilst this research is a novel attempt at very early-stage real-time malware detection and process killing, one might consider the usability and prefer a model with a very high TNR, even if this is at the expense of the TPR.

Considering this, the AdaBoost regression algorithm achieves a 100% TNR with a 39.50% TPR on the validation set. The high FNR is retained in the test set standing at

			val			test		
methodology	model	n features	F1	tnr	tpr	F1	tnr	tpr
Regression	AdaBoost	26	56.63	100.00	39.50	15.06	97.92	8.40
Regression + 4 alerts	GBDT	26	85.91	95.80	77.31	68.50	94.98	56.04

Table 5.7: Two models' F1-score, TNR, TPR for the validation and test set scoring the highest TNR on the validation and test sets.

97.92% but the TPR drops even further to just 8.40%. The GBDT also using regression to estimate the value of process killing, and coupled with a minimum of 4 alerts performs well on the test set but does not stand out in the validation set see Table 5.7.

Though less than 10% of the test set malicious processes are killed by the AdaBoost regressor, this model may be the most viable despite the low TPR. Future work may examine the precise behaviour and harm caused by malware that is/is not detected. To summarise results, the most-detected families were Ekstak (180), Mikey (80), Prepscram (53 processes) and Zusy (49 processes) of 745 total samples.

5.7 Measuring damage prevention in real time

Though a high percentage of processes are correctly identified as malicious by the best performing model (RF with 2 alerts and 37 features); it may be that the model detects the malware after it has already caused damage to the endpoint. Therefore, instead of looking at the time at which the malware is correctly detected, a live test was carried out with ransomware to measure the percentage of files corrupted with and without the process killing model working. This real time test also assesses whether malware can indeed be detected in the early stages of execution or whether the data recording, model inference and process killing is too slow in practice to prevent damage.

It is possible to quantify the damage caused by ransomware using the proportion of modified files as Scaife et al. [172] have done in developing a real-time ransomware (only) detection system. The damage of some malware types are more difficult to quantify owing to their dependence on factors outside the control of the malware. For example the damage caused by spyware will depend on what information it is able to obtain so it is difficult to quantify the benefit of killing spyware 5 seconds after execution compared with 5 minutes into execution. Ransomware offers a clear metric for the benefits of early detection and process killing.

		val			test		
model	n features	F1	tnr	tpr	F1	tnr	tpr
RF (alerts: 2)	37	91.30	94.96	88.24	**81.50**	81.53	87.97
DT (rolling mean: 2)	26	**93.16**	94.96	91.60	73.82	66.19	88.40

Table 5.8: Random Forest and Decision Tree each with a minimum requirement of two alerts ('malicious classifications') to kill a process. F1, TNR and TPR reported on validation and test set.

Although the RF with with a minimum of 2 alerts using both process and global data gave the highest F1-score on the test set (81.50), earlier experiments showed that RFs are not one of the most computationally efficient models by comparison with those tested. Therefore a decision tree is trained on process-only data (26 features) in case the time-to-classification is important for damage reduction despite the lower F1-score. For this reason the decision tree model is used in this test. The DT also has a very slightly higher TPR (see Table 5.8) so a higher damage prevention rate may be partially due to the model itself rather than just the fewer features being collected and model classification speed. Neither the TNR nor TPR on the test set surpass 90% on either model indicates that there may be significant overlap in the behavioural traces of malicious and benign software when considered at the process level and in the short-term.

22 fast-acting ransomware files were identified from a separate VirusShare [201] repository which *(i)* do not require internet connection and *(ii)* begin encrypting files within the first few second of execution. The former condition is set because the malicious server may no longer exist and for safety it is not desirable to connect to it if it does. These are the types of malware that if the proposed model could block, would save significant damage to the user in a time-frame that it would be difficult for a human

to react to.

The 22 samples were executed for 30 seconds each without the process killing model and the number of files modified was recorded. The process was repeated with 4 process killing models: DT with min. 2 alerts and 26 features, RF with min. 2 alerts and 37 features, AdaBoost regressor with 26 features and GDBT regressor with min. 4 alerts and 26 features.

It was necessary to run the killing model with administrator privileges and to write an exception for the Cuckoo sandbox agent process which enables the host machine to read data from the guest machine since the models killed this process. The need for this exception highlights that there are benign applications with malicious-like behaviours perhaps especially those used for networking and security.

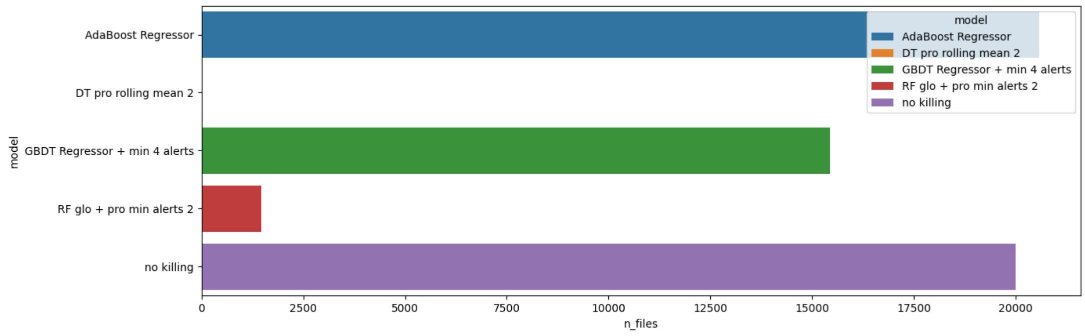


Figure 5.11: Total number of files corrupted by ransomware with no process killing and with three process killing models within the first 30 seconds of execution.

Figure 5.11 and Table 5.9 give the total number of corrupted files across the 22 samples. The damage prevention column is a proxy metric denoting how many files were not corrupted using a given process killing model by comparison with no model being in place. The 22 samples on average each corrupt 910 files within 30 seconds.

The DT model almost entirely eliminates any file corruption with only three being corrupted. The RF saves 92.68% of files. The ordinal ranking of ‘damage prevention’ is the same as the TPR on the test set but the relationship is not proportional. The same ordinal relationship indicates that the simulated impact of process killing on the

Model	Files damaged	Damage reduction	Detection rate (Ransomware TPR)	Test set TPR
no killing	19,997	-	-	-
DT pro rolling mean 2	3	99.98%	100.00	88.40
RF glo + pro min alerts 2	1,464	92.68%	100.00	87.97
GBDT regressor + min 4 alerts	15,432	22.83%	22.07	56.04
AdaBoost regressor	20,578	0.00%	9.09	8.83

Table 5.9: Total number of files corrupted by ransomware with no process killing and with three process killing models within the first 30 seconds of execution. Damage reduction is the percentage of files spared when no killing is implemented

collected test set was perhaps a reasonable approximation of measuring at least fast-acting ransomware damage, despite the TPR test set metrics being based on other malware families too.

The DT demonstrates that this architecture is capable of preventing damage, but the TNR on the test set of the DT model is so low (66.19) that this model cannot be preferred to the RF (81.53 TNR), which still prevents over 90% of file damage.

The GBDT prevents some damage, and detects a comparable number of ransomware samples (1 in 5). The AdaBoost regressor detected 2 ransomware samples of the 22, and in these two cases more than 64% and 45% of files were saved respectively; perhaps with more execution time the files would be detected but the key benefit of process killing is to stop damaging software like these ransomware samples and this algorithm saw more files encrypted than when no killing model was used; this is because there will be a slight variance in the ransomware behaviour and execution time each time it runs. It may appear surprising that more files were corrupted with the Adaboost algorithm running, but the process launching and shutdown of the virtual machine is not precise to the millisecond, thus extra execution time may have allowed additional files to be corrupted. The Random Forest is the most plausible model, balancing damage prevention and TNR, however the delay in classification may be a result of the requirement to collect more features and/or the real-time of the model itself.

5.8 Discussion: Measuring execution time in a live environment

Though algorithm execution duration was measured above, due to batch processing used by the models, the number of processes being classified can be increased by an order of magnitude with a negligible impact on execution time. The data collection and process killing both have $O(n)$ complexity; where n is the number of processes therefore it is expected that the number of processes to impact classification time. So a further experiment was carried out with the RF to measure in a live environment how long the data collection, model inference, and process killing takes as the number of processes increases. This was tested by executing more than 1000 processes in the virtual machine whilst the process killing RF runs.

Some processes demand more computational resources than others, and some malware in our test set locked pages in memory[42] which prevented the model from having sufficient resources to collect data, leading to tens of seconds during which no data was captured and many processes were launched, with better software engineering practices the model may be more robust against this kind of malicious activity.

These differences in behaviour can cause the evaluation time to lag as demonstrated by the outlier points visible in Figure 5.12. The data shows a broadly linear positive correlation between the number of processes (being monitored or killed) and the time taken for the data collection and process killing; this confirms the hypothesis that more processes equates to slower processing time. The slowest total processing time was 0.81 seconds (seen with both 17 and 40 simultaneous processes running) but the mean processing time is just under 0.3 seconds with 65 simultaneous processes, fitting comfortably within the 1-second goal time. Additional code optimisation could greatly improve on these initial results which indicate that the processing, even using standard libraries and a high level programming language, can execute reasonably quickly.

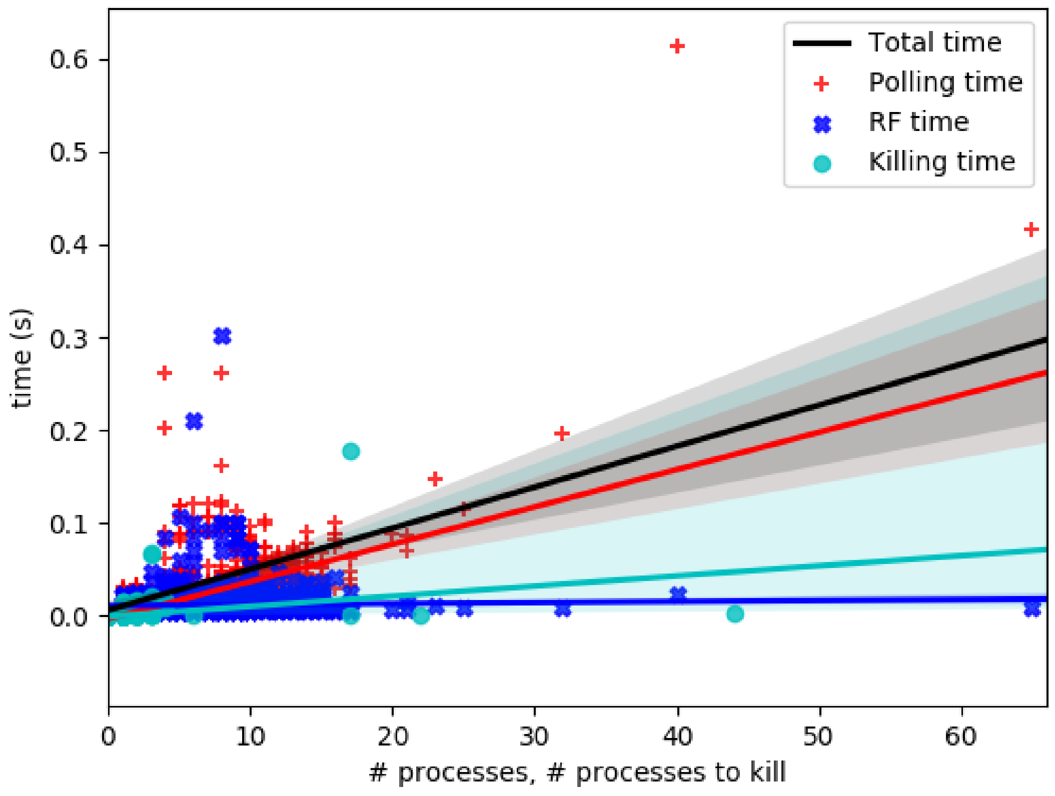


Figure 5.12: Mean time to collect data, analyse data with Random Forest, and kill varying numbers of processes

5.9 Future Work

Behavioural malware analysis research using machine learning regularly reports $>$95% classification accuracy. Though useful for analysts, behavioural detection should be deployed as part of endpoint defensive systems to leverage the full benefits of a detection model. Dynamic analysis is not frequently used for endpoint protection, perhaps because it takes too long in data collection to deliver the quick verdicts required for good user experience. Real-time detection on the endpoint allows for observation of the full trace without the user having to wait. However, real-time detection also introduces the risk that malware will cause damage to the endpoint. This risk requires that processes detected as malicious are automatically killed as early as possible to avoid harm.

There are some key challenges to implementation, which have been outlined in this chapter:

- The need for signal separation drives the use of individual processes and only partial traces can be used.
- The significant drop in accuracy on the unseen test set, even without process killing demonstrates that additional features may be necessary to improve detection accuracy.
- With the introduction of process killing, the poor performance of the models on either benignware classification (RF min 2 alerts: TNR 81% with an 88% TPR on the test set) or on malware classification (GBDT regressor min 4 alerts: 56% TPR with a 94% TNR on the test set) means that considerable further work is needed before very early stage real-time detection can be considered for real-world use.
- Real-time detection using full execution traces of processes however, may be viable. This is useful to handle VM-aware malware which may only reveal its true behaviour in the target environment. Although the more complex approach using DQNs algorithms did not outperform the supervised models with some additional

statistical thresholds, the regression models had better performance in correctly classifying benignware. Reinforcement learning could still be useful for real-time detection and automated cyber defence models, but the DQN in these experiments did not perform well.

- Despite the theoretical unsuitability of supervised learning models to state-action problems, these experiments demonstrate how powerful supervised learning can be for classification problems, even if the problem is not precisely the one that the model is attempting to solve.
- Future work may require a more comprehensive manual labelling effort at the process level and perhaps labelling sub-sections of processes as malicious or benign.

An additional consideration for real-time detection with automated actions is whether this introduces an additional denial-of-service vector using process injection for example to trigger process killing. This may also however indicate that an attacker is present and therefore aid the user.

5.10 Conclusions

This chapter has built on previous work in real-time detection to address some of the key challenges: signal separation, detection with partial execution traces and computational resource consumption with a focus on preventing harm to the user, since real-time detection introduces this risk.

Behavioural malware detection using virtual machines is a well-established research field yielding high detection accuracy in recent literature [48, 87, 89, 156]. However, as is shown here, fixed-time execution in a sandbox may not reveal malicious functionality. Real-time malware analysis addresses this issue but risks executing malware on the endpoint and requires detection to take place at the process level, which is more challenging as the definition of a malicious process can be unclear. These two reasons may account for the limited literature on real-time detection. Looking forward real-time detection may become more popular if static data manipulation and VM-evasion continue to be used and the costs of malicious execution continue to rise. Real-time detection does not need to be an alternative to these approaches, but could hold complementary value as part of a defence-in-depth endpoint security.

To the best of our knowledge previous real-time detection work has used up to 5 simultaneous applications, whereas users may use far more. This gives rise to *RQ4: Is it possible to separate the signals of tens of simultaneous benign and malicious processes?* Experimenting with up to 35 concurrent applications, it is possible to distinguish malicious and benignware, this is the third contribution of this thesis: *C3: Real-time malware detection against a significantly larger number of user background applications than have previously been investigated with analysis of the impact of the increase* this chapter has demonstrated that many machine learning algorithms can handle up to 1000 processes and maintain very fast execution speeds.

Automatic actions are necessary in response to a detection if the goal is to prevent harm. Otherwise this is equivalent to letting the malware fully execute and simply monitor its behaviour since human response times are unlikely to be quick enough for fast-acting malware. Sun et al. [188] identified this truth in their research but their work measured the number of correct detections rather than looking at the damage caused to the user. To prevent damage, detection prior to the malicious payload should take place. Previous work has looked at damage as a percentage of execution trace [48] or using several minutes of data [188].

From a user perspective the question is not ‘What percentage of malware was executed?’ or ‘Was the malware detected in 5 or 10 minutes?’ but ‘How much damage as been done?’. Perhaps the lack of very early detection was because it did not yield promising results, leading to *RQ3: Does the use of lower latency algorithms impact* *upon the speed of accurate malware detection?* and *RQ5: Is it possible to detect and kill malicious processes early enough to prevent malicious damage?*

The final contribution of this thesis *C4: The first model to try and mitigate malware damage in real-time using malicious process detection and killing* addresses both of these. By creating statistical filters on top of supervised learning models it was possible to prevent 92% of files from being corrupted by fast-acting ransomware thus reducing the requirements on the user or organisation to remediate the damage, since it was prevented in the first instance (the rest of the attack vector would remain a concern).

This approach does not achieve the detection accuracies of state of the art offline behavioural analysis models, but these models typically use the full post-execution trace of malicious behaviour. Delaying classification until post-execution negates the principal advantages of real-time detection. However, the proposed model presents an initial step towards a fully automated endpoint protection model which becomes increasingly necessary as adversaries become more and more motivated to evade offline automated detection tools.

Chapter 6

Conclusions, Limitations, Future Work

The field of automatic malware detection using machine learning techniques continues to be widely researched and published in with various data sources, algorithms, operating systems and file types. This field is closely tied to the commercial antivirus market, but it is unclear how much ML (academic- or industry-led) is being used in commercial products as this information cannot be verified without insider knowledge. Increasingly, some works are focusing on the weaknesses of ML and how research findings may not hold in practice. This thesis has uncovered and addressed several unanswered questions, which deserve attention when considering adopting research in practice. This chapter first summarises the findings of this thesis then presents limitations, future work arising from this research and finally come concluding remarks.

6.1 Thesis summary

This thesis has focused on dynamic malware detection due to the difficulty of obfuscating dynamic data. It has looked at machine learning approaches to draw out the distinctions between malware and benignware since these models can be retrained automatically, thus reducing the delay and expense of hand-generating rules.

The first contribution of this thesis complements the growing voice in malware detection literature warning against the vulnerabilities of machine learning methods. Whilst concept drift and adversarial samples are well-documented, research addressing the performance of dynamic detection models in different contexts had not been addressed. Just as concept drift can be seen as a change in context over time, but the idiosyncrasies of cyber ecosystems can also be considered different contexts. In order to test whether the highest performing lab models can be assumed to perform best in other environments Chapter 3 compared classification accuracy on datasets from the public internet and those captured by a commercial organisation. The difference in classification accuracies on unseen test sets confirmed the uncontroversial argument that the model would perform differently. The contribution of this work however is

C1: A demonstration that the most accurate type of dynamic data in laboratory testing may not be the best-performing or most robust data type when tested on data of a different provenance.

The data type used to train the model had a far greater impact than the algorithm chosen, and the most commonly used dynamic malware detection data type, API calls, did not produce the most robust results, though it performed the best under lab. conditions. This work was limited to comparing just two environments but may prove valuable for commercial adoption of research either as suggestion of an alternative data source to trial. More generally it highlights the need for security practitioners to have a way to evaluate the performance of models in their own environment.

The second contribution of this thesis looks at one of the primary criticisms levelled at dynamic data, that it takes too long time to collect to be suitable for endpoint detection. The key research question asked whether the more-reliable dynamic data can compete with static data analysis times for endpoint protection. Chapter 4 examined the classification accuracy in the first few seconds of execution and made the following contribution:

C2: The first model to predict whether or not software is malicious based on its early behaviour.

This contribution is significant as it potentially enables quick dynamic analysis to augment existing endpoint defences. The omission of entire malware families from the training set indicated that it may be possible to detect zero-days using this approach and that new malwares have more in common with existing malware in the injection mechanism even if the payload is significantly different. This was the first demonstration of the capabilities for early stage prediction on the initial execution trace of software samples. Whilst it performed well on the test set and on unseen malware families, it could potentially be obfuscated with simple benign activity injection at the start of execution.

This limitation inspired the third and most forward-looking research question of this thesis. Chapter 5 imagines that early-stage dynamic malware detection has become widely adopted and the issues associated with a reliance on fixed observation periods in a VM. This chapter argued that real-time detection is the future of endpoint protection as the malicious behaviour cannot be concealed. Previous work in this area had not considered that a live detection should yield some benefit for the user i.e. stop the malware, as such our work focused on killing malicious processes within a short time frame. Three key research questions which had not previously been addressed concerned the ability to distinguish malicious processes with a large amount of background noise: *RQ4: Is it possible to separate the signals of tens of simultaneous benign and malicious processes?*; the requirement to identify malware as early as possible and the viability of killing processes automatically when there is a risk of making the endpoint unusable if benignware is compromised too *RQ3: Does the use of lower latency algorithms impact upon the speed of accurate malware detection?*

The experiments sought to address the impact of an automated detection system on damage prevention, which has not previously been explored, with the following two contributions:

C3: Real-time malware detection against a significantly larger number of user background applications than have previously been investigated with analysis of the impact *of the increase*

and

C4: The first model to try and mitigate malware damage in real-time using malicious process detection and killing

By filtering supervised model outputs it was possible to reduce fast-acting ransomware encryption by 92% in a real-time trial. The detection accuracy for benignware requires further work but demonstrates the challenge of creating a real-time detection paradigm that is both capable of preventing actual damage and not interrupting user experience, this has not been addressed in previous work.

6.2 Limitations

Chapters 3, 4 and 5 have each outlined limitations of the experimental results specific to those chapters. Here is a summary of some of the wider limitations of the research in this thesis

6.2.1 Dataset labelling

Each of these chapters uses labelled datasets. These labels have referred to whether or not the sample is malicious, the family to which the malware belongs and sometimes to try and date the sample. The sources used for labelling malware are imperfect and change over time [2, 66]. The impact of this imperfect labelling is likely to reduce model accuracy [11] and may see biases towards certain malware above others. For example, the storage of malicious hashes is an increasing problem for antivirus vendors and one method for handling this is to maintain a list of hashes of the more likely malware than older samples. This limits the reliability of VirusTotal as a means to label samples and brings into question the method used to retrain models - must this ultimately rely on labelling samples and is labelling samples easier than devising hand-written rules?

The answer is ’yes’ since labelling samples is a prerequisite to devising rules however it highlights the reliance of supervised learning on human effort and it’s limitations as a means to fully automate malware detection.

Furthermore, some of the malware is adware, which is sometimes not considered malicious but rather an unwanted programme. In these experiments adware is considered malware since it is desirable to detect and remove it.

6.2.2 Ransomware as a Case Study

Chapters 4 and 5 both use ransomware as a case-study to measure the ability to detect zero-day malware and the damage prevented by automatic process killing respectively. Although ransomware is a considerable concern in cybersecurity, the focus of this thesis is on general malware detection. Arguably ransomware is one of the less subtle types of malware, especially when the focus is placed on fast-acting samples as in Chapter 5. As such it may be considered relatively easy to detect. This is not a problem in so far as ransomware causes a lot of damage and if it’s easy to detect that is good. However, there are other types of malware which can have catastrophic impact on businesses and these tend to be highly sophisticated and tailored attacks (APTs). This subset is not investigated in this thesis due in part to the bespoke nature of these malware likely requiring the specific infrastructure of its victims in order to function. However, future work could assess the capability of real-time detection to detect these attacks since real-time detection can be used directly in the victim environment and may therefore observe these behaviours. This would be challenging for researchers to carry out without the co-operation of multiple industrial environments in order to ensure the attacker’s intended context is accurately replicated.

6.2.3 Machine Metrics

As mentioned in Chapter 3, the reliance on machine metrics is machine-specific which would require models to be recalibrated to endpoints if real-time monitoring is adopted, VM-based approaches do not suffer this problem. Future work could examine the security implications of retraining models locally, whether this introduces new vulnerabilities or gives better or worse protection to machines with particular characteristics.

6.3 Future Work

This thesis seeks to address some of the areas untouched by previous research in automatic malware detection for the benefit of the research community and industrial adopters alike. The contributions described here are also a starting point for continued work in this area.

6.3.1 Other file types and operating systems

This thesis has focused on Windows7 executables. Though Windows7 was the most prevalent operating system globally [68] at the time of data collection and Windows executables are the most commonly submitted file to VirusTotal [203], these models could be validated by investigating whether they are capable of detecting malicious PDFs, URLs and other potential vehicles for malware, as well as applications which run on other operating systems.

6.3.2 Robustness

Relating to the robustness of data there are a number of avenues for future research. The first is a general machine learning question: are sparse feature vectors more vulnerable to contextual and concept drift? Within the malware detection domain the results presented in Chapter 3 (and [158]) do not imply that machine metrics are the answer to cross-context accuracy robustness but are better interpreted as a cautionary tale. The problem of contextual difference should be treated like concept drift and assume that retraining needs to take place. Further research could investigate the relative benefits of decentralised fine-tuning of detection models at edge nodes (enterprises) or whether it is better simply to train the model at the edge in the first place. Research into when a model requires retraining as used in adversarial sample and concept drift defences may also be usable here.

From a usability perspective the longevity of all of the results in this thesis must be tested over a significant time period. The results comparing data sources, early detection and real-time detection could be used to guide the direction of model building for long-term testing.

6.3.3 Earlier detection

The early stage malware detection (Chapter 4 and [156]) indicates that dynamic malware detection may not be as inconvenient in terms of analysis time as static methods. This could form the basis of a centralised analysis point on a local network e.g. at an ingestion point but it is not well-suited to being run on individual machines due to the computational overheads of running virtual machines. Future work could also look at even smaller time increments. This approach is vulnerable to manipulating data at the start of a malicious program but it could be possible to combine static and dynamic analysis in order to try and fast-forward through the benign portion, however this thesis considered that the more forward-looking response was to investigate real-time detection and automatic agents.

6.3.4 Improving Real-time Detection Accuracy

Far less research has been conducted in the area of real-time detection. In part it is because this area is much more challenging. The results in Chapter 5 give an early indication that there may be promising research to be developed from this work. Future directions include: quantifying damage and learning user profiles to improve benignware classification; but the greatest challenge to high accuracy for the research presented in this thesis was simply conducting supervised learning to label the sub-processes as benign or malicious. Future research could look at labelling at the process level and labelling processes as malicious or benign depending on their enacting certain activity. This is an intensive manual process but may be bootstrapped by starting with specific malware families with known behaviour.

Another avenue to consider is robustness against interference from the malware - memory locking was able to block the model from performing as intended, though security controls could be implemented to prevent this, malware authors may devise new methods to limit the capability of the detection model.

6.3.5 When Does Harm Begin?

This research has looked at dynamic malware detection, arguing that the behaviour is harder to disguise than the code and therefore early behavioural detection should be preferred in order to prevent malicious damage. One broad question leads on from this: to what extent does malware reveal itself before it caused harm? This is particularly challenging as it requires a definition of ’harm’, which in itself is context-dependent.

6.4 Concluding remarks

The contributions of this thesis are intended to take the stakeholders in malware detection into account in ways that they have not been in previous work.

The demonstration that the best model in the lab does not perform best in the field has not previously been explicitly tested for dynamic malware analysis. As significant players in the antivirus market offer sandbox-based analysis, this is an important message for those considering using such products.

Early stage malware detection considers the malware author and ease with which static code can be manipulated. This thesis presented the first attempt at classifying malware within a time-bound limit in order to make dynamic analysis temporally competitive with static analysis for endpoint protection. An incidental discovery of this work suggests that different malware shares more in common through injection mechanisms than through payload behaviour, this in turn suggests that it may be possible to detect malware before it is able to cause harm.

This thesis concludes by considering the user, with limited computational resources and the adversary seeking to evade detection. The weaknesses of static and virtual-machine based analysis strongly suggest that real-time detection live on the endpoint will be necessary to keep pace with trends in the malware landscape. Real-time detection is a challenging and perhaps controversial security measure, but it does circumvent the problems of conditional execution and resource intensity required by sandbox full-trace analysis. Here it has been demonstrated for the first time that a general malware detector (rather than specific to a family) is capable of preventing damage to the endpoint. I hope that future research will use the contributions of this thesis to uncover further possibilities in automated malware detection.

filetype	features	classifier	μ(f1 public)	μ(f1 org.)	abs. μ(f1 public) μ(f1 org.)	μ(abs. Δ f1)	μ(abs Δ tnr)	μ(abs. Δ tpr)
DOC	all	NN	97.5	71.1	26.4	26.4	1.2	27.3
DOC	all	RF	98.2	41.4	56.8	56.8	0.2	45.3
DOC	all	SVM	94.8	68.1	26.8	26.8	3.7	28.9
DOC	api	NN	97.0	69.4	27.5	27.5	2.8	43.6
DOC	api	RF	98.0	31.3	66.7	66.7	0.5	77.6
DOC	api	SVM	85.9	82.6	3.2	3.4	29.9	27.2
DOC	api sequential	NN	92.4	27.8	64.6	64.6	2.0	72.1
DOC	api sequential	RF	98.1	30.9	67.2	67.2	6.3	78.0
DOC	api sequential	SVM	78.1	30.9	47.3	47.3	8.6	56.0
DOC	machine metrics	NN	92.8	22.6	70.1	70.1	2.4	79.2
DOC	machine metrics	RF	94.1	26.5	67.7	67.7	4.8	77.5
DOC	machine metrics	SVM	84.7	49.2	35.5	35.5	6.0	55.6
DOC	network	NN	88.1	42.7	45.4	45.4	5.5	62.3
DOC	network	RF	91.2	34.7	56.5	56.5	3.6	69.4
DOC	network	SVM	74.2	57.9	16.3	16.3	10.5	34.1
EXE	all	NN	80.0	64.1	15.9	15.9	1.7	10.2
EXE	all	RF	85.3	71.8	13.6	13.6	0.8	10.4
EXE	all	SVM	77.1	63.2	14.0	14.0	1.1	9.0
EXE	api	NN	77.1	62.1	15.1	15.1	2.7	17.3
EXE	api	RF	81.4	65.2	16.1	16.1	2.6	20.0
EXE	api	SVM	70.8	57.5	13.3	13.3	1.7	15.7
EXE	api sequential	NN	76.7	70.5	6.1	6.7	3.7	8.6
EXE	api sequential	RF	81.3	65.2	16.1	16.1	3.1	20.7
EXE	api sequential	SVM	71.1	73.6	2.5	2.6	4.1	1.3
EXE	machine metrics	NN	71.8	71.8	0.1	3.6	8.6	5.8
EXE	machine metrics	RF	77.8	72.6	5.2	5.5	8.3	14.0
EXE	machine metrics	SVM	68.4	70.2	1.8	2.6	6.4	3.7
EXE	network	NN	61.0	50.8	10.2	10.2	4.1	12.9
EXE	network	RF	73.3	61.5	11.8	11.8	4.0	18.3
EXE	network	SVM	51.0	40.0	11.0	11.0	2.7	10.7
PDF	all	NN	98.7	98.3	0.4	0.7	0.6	0.4
PDF	all	RF	99.4	98.8	0.6	0.8	0.5	0.4
PDF	all	SVM	98.2	97.7	0.5	1.0	0.8	0.6
PDF	api	NN	98.4	97.5	0.8	0.9	2.2	0.8
PDF	api	RF	98.9	98.4	0.5	1.0	1.1	1.4
PDF	api	SVM	98.0	97.5	0.5	0.9	1.7	1.2
PDF	api sequential	NN	54.6	55.2	0.6	2.8	0.5	2.7
PDF	api sequential	RF	98.0	97.7	0.3	1.2	1.1	1.7
PDF	api sequential	SVM	11.3	12.1	0.8	3.2	0.0	1.8
PDF	machine metrics	NN	99.0	98.3	0.7	0.9	1.3	1.2
PDF	machine metrics	RF	99.2	98.4	0.7	0.9	1.3	0.9
PDF	machine metrics	SVM	98.2	97.2	1.0	1.2	2.6	0.9
PDF	network	NN	98.4	97.9	0.5	0.8	1.4	1.1
PDF	network	RF	98.3	97.8	0.5	0.5	1.6	1.2
PDF	network	SVM	90.6	89.7	0.9	2.0	3.1	3.2
XLS	all	NN	97.0	57.7	39.4	39.4	2.2	1.1
XLS	all	RF	99.5	77.8	21.7	21.7	2.7	0.3
XLS	all	SVM	97.6	76.8	20.8	20.8	2.6	1.3
XLS	api	NN	96.2	0.0	96.2	96.2	0.1	92.8
XLS	api	RF	99.2	77.8	21.4	21.4	2.8	7.3
XLS	api	SVM	97.3	60.2	37.1	37.1	0.0	46.5
XLS	api sequential	NN	97.4	4.9	92.6	92.6	0.0	92.4
XLS	api sequential	RF	98.6	62.0	36.6	36.6	0.0	49.0
XLS	api sequential	SVM	96.8	0.0	96.8	96.8	0.0	93.9
XLS	machine metrics	NN	98.5	11.9	86.6	86.6	1.3	86.6
XLS	machine metrics	RF	98.8	81.7	17.0	17.0	1.3	12.5
XLS	machine metrics	SVM	97.7	87.8	9.9	9.9	0.5	17.9
XLS	network	NN	95.8	54.9	40.8	40.8	7.7	3.5
XLS	network	RF	97.3	75.2	22.1	22.1	1.1	20.1
XLS	network	SVM	40.8	79.9	39.1	52.9	3.2	66.9

Table 1: Mean (μ) F-scores for withheld public test set, organisation test set, absolute difference between mean F-scores, mean absolute difference between f-scores per model, mean absolute difference in true positive rate (TPR) and true negative rate (TNR) per model. Evaluated using a public and commercial dataset for 4 filetypes, 3 algorithms, 6 input feature sets averaged over 10 models (10-fold cross-validation). **N.B.** Scores have been multiplied by 100 for easier reading

	val			test		
model	f1	tnr	tpr	f1	tnr	tpr
AdaBoostModel_glo_pro	77.58	55.46	91.60	67.04	49.80	88.67
AdaBoostModel_glo_pro mean process tree	78.01	55.46	92.44	66.75	50.48	87.59
AdaBoostModel_glo_pro process tree min alerts: 1	78.87	55.46	94.12	62.29	34.03	90.35
AdaBoostModel_glo_pro process tree min alerts: 2	78.87	55.46	94.12	62.29	34.03	90.35
AdaBoostModel_glo_pro process tree min alerts: 3	78.87	55.46	94.12	62.29	34.03	90.35
AdaBoostModel_glo_pro process tree min alerts: 4	78.87	55.46	94.12	62.29	34.03	90.35
AdaBoostModel_glo_pro rolling mean window: 2	79.22	70.59	84.87	69.57	60.88	84.88
AdaBoostModel_glo_pro rolling mean window: 3	79.37	72.27	84.03	69.59	61.53	84.39
AdaBoostModel_glo_pro rolling mean window: 4	80.67	80.67	80.67	68.44	67.80	77.34
AdaBoostModel_glo_pro sum alerts min: 2	80.66	78.15	82.35	69.35	66.58	79.89
AdaBoostModel_glo_pro sum alerts min: 3	81.20	83.19	79.83	67.83	70.92	73.88
AdaBoostModel_glo_pro sum alerts min: 4	80.87	84.87	78.15	65.92	73.32	69.00
AdaBoostModel_pro	75.34	47.06	92.44	65.64	45.79	88.89
AdaBoostModel_pro mean process tree	75.86	48.74	92.44	65.74	47.83	87.59
AdaBoostModel_pro process tree min alerts: 1	75.68	45.38	94.12	60.31	26.46	91.17
AdaBoostModel_pro process tree min alerts: 2	75.68	45.38	94.12	60.31	26.46	91.17
AdaBoostModel_pro process tree min alerts: 3	75.68	45.38	94.12	60.31	26.46	91.17
AdaBoostModel_pro process tree min alerts: 4	75.68	45.38	94.12	60.31	26.46	91.17
AdaBoostModel_pro rolling mean window: 2	78.03	64.71	86.55	69.35	59.63	85.47
AdaBoostModel_pro rolling mean window: 3	77.99	67.23	84.87	68.91	59.20	84.99
AdaBoostModel_pro rolling mean window: 4	80.83	79.83	81.51	69.01	66.44	79.40
AdaBoostModel_pro sum alerts min: 2	81.12	75.63	84.87	67.91	61.06	81.68
AdaBoostModel_pro sum alerts min: 3	81.17	80.67	81.51	66.64	64.97	76.42
AdaBoostModel_pro sum alerts min: 4	79.66	80.67	78.99	64.25	68.12	70.14
AdaBoostModel_pro_tree	75.08	47.73	94.96	64.12	30.58	86.60
AdaBoostRegression_pro_process	56.63	100.00	39.50	15.06	97.92	8.40
DTModel_glo_pro	84.53	71.43	94.12	71.41	58.19	90.62
DTModel_glo_pro mean process tree	85.93	73.95	94.96	71.49	59.48	89.70
DTModel_glo_pro process tree min alerts: 1	84.64	70.59	94.96	65.59	42.42	91.27
DTModel_glo_pro process tree min alerts: 2	84.64	70.59	94.96	65.56	42.34	91.27
DTModel_glo_pro process tree min alerts: 3	84.64	70.59	94.96	65.56	42.34	91.27
DTModel_glo_pro process tree min alerts: 4	84.64	70.59	94.96	65.56	42.34	91.27
DTModel_glo_pro rolling mean window: 2	88.70	88.24	89.08	75.09	70.49	86.94
DTModel_glo_pro rolling mean window: 3	87.87	87.39	88.24	74.57	69.77	86.61
DTModel_glo_pro rolling mean window: 4	89.08	93.28	85.71	74.04	74.29	81.63
DTModel_glo_pro sum alerts min: 2	89.74	91.60	88.24	75.48	72.64	85.69
DTModel_glo_pro sum alerts min: 3	89.47	94.12	85.71	74.00	75.62	80.38
DTModel_glo_pro sum alerts min: 4	88.39	94.96	83.19	70.19	77.30	72.63
DTModel_pro	89.76	82.35	95.80	71.53	56.54	92.25
DTModel_pro mean process tree	90.91	84.03	96.64	72.13	59.38	91.06
DTModel_pro process tree min alerts: 1	90.20	82.35	96.64	64.45	37.04	92.79
DTModel_pro process tree min alerts: 2	90.20	82.35	96.64	64.42	36.97	92.79
DTModel_pro process tree min alerts: 3	90.20	82.35	96.64	64.42	36.97	92.79
DTModel_pro process tree min alerts: 4	90.20	82.35	96.64	64.42	36.97	92.79
DTModel_pro rolling mean window: 2	93.16	94.96	91.60	73.82	66.19	88.40
DTModel_pro rolling mean window: 3	91.77	94.96	89.08	73.49	66.15	87.80
DTModel_pro rolling mean window: 4	90.75	95.80	86.55	72.05	69.38	82.38
DTModel_pro sum alerts min: 2	92.17	95.80	89.08	73.43	67.44	86.56
DTModel_pro sum alerts min: 3	90.75	95.80	86.55	71.53	69.63	81.25
DTModel_pro sum alerts min: 4	89.29	95.80	84.03	67.58	70.81	73.55
DTModel_pro_tree	85.93	73.48	97.48	70.40	43.02	91.57
DTRegression_pro_process	89.06	80.67	95.80	71.62	57.98	91.22
GBDTModel_glo_pro	80.44	63.87	91.60	72.62	59.73	91.71
GBDTModel_glo_pro mean process tree	80.88	63.87	92.44	72.76	60.63	91.22
GBDTModel_glo_pro process tree min alerts: 1	81.75	63.87	94.12	66.32	43.28	92.14
GBDTModel_glo_pro process tree min alerts: 2	81.75	63.87	94.12	66.32	43.28	92.14
GBDTModel_glo_pro process tree min alerts: 3	81.75	63.87	94.12	66.32	43.28	92.14
GBDTModel_glo_pro process tree min alerts: 4	81.75	63.87	94.12	66.32	43.28	92.14
GBDTModel_glo_pro rolling mean window: 2	85.12	83.19	86.55	76.06	71.50	87.80
GBDTModel_glo_pro rolling mean window: 3	84.52	84.03	84.87	75.87	71.82	87.15
GBDTModel_glo_pro rolling mean window: 4	84.12	86.55	82.35	75.25	76.69	81.57
GBDTModel_glo_pro sum alerts min: 2	84.87	84.87	84.87	76.46	75.65	84.66
GBDTModel_glo_pro sum alerts min: 3	85.22	89.08	82.35	74.12	78.77	77.78
GBDTModel_glo_pro sum alerts min: 4	84.44	90.76	79.83	71.99	81.10	72.30
GBDTModel_pro	80.73	62.18	93.28	71.31	58.09	90.51
GBDTModel_pro mean process tree	82.05	64.71	94.12	71.76	59.59	90.14
GBDTModel_pro process tree min alerts: 1	80.71	59.66	94.96	64.88	40.34	91.33
GBDTModel_pro process tree min alerts: 2	80.71	59.66	94.96	64.87	40.30	91.33
GBDTModel_pro process tree min alerts: 3	80.71	59.66	94.96	64.87	40.30	91.33
GBDTModel_pro process tree min alerts: 4	80.71	59.66	94.96	64.87	40.30	91.33
GBDTModel_pro rolling mean window: 2	84.68	79.83	88.24	75.08	71.60	85.91
GBDTModel_pro rolling mean window: 3	84.08	80.67	86.55	74.99	71.71	85.64
GBDTModel_pro rolling mean window: 4	84.39	84.87	84.03	73.91	76.05	79.84
GBDTModel_pro sum alerts min: 2	85.48	84.03	86.55	74.50	74.40	82.33
GBDTModel_pro sum alerts min: 3	85.11	86.55	84.03	72.35	76.77	76.59
GBDTModel_pro sum alerts min: 4	83.84	88.24	80.67	70.17	78.38	71.71
GBDTModel_pro_tree	79.02	59.09	94.96	71.08	46.68	90.51
GBDTRegression_pro_process	89.71	87.39	91.60	71.84	80.57	72.52
MLPModel_glo_pro	66.67	13.45	93.28	57.92	19.00	90.68
MLPModel_glo_pro mean process tree	67.48	16.81	93.28	59.79	25.74	90.51
MLPModel_glo_pro process tree min alerts: 1	67.46	13.45	94.96	57.61	17.64	90.84
MLPModel_glo_pro process tree min alerts: 2	67.46	13.45	94.96	57.61	17.64	90.84
MLPModel_glo_pro process tree min alerts: 3	67.46	13.45	94.96	57.61	17.64	90.84
MLPModel_glo_pro process tree min alerts: 4	67.46	13.45	94.96	57.61	17.64	90.84
MLPModel_glo_pro rolling mean window: 2	67.96	28.57	88.24	58.73	32.20	84.17
MLPModel_glo_pro rolling mean window: 3	67.79	34.45	84.87	58.90	34.31	83.20
MLPModel_glo_pro rolling mean window: 4	68.75	41.18	83.19	58.32	40.55	78.16
MLPModel_glo_pro sum alerts min: 2	68.94	38.66	84.87	59.51	39.19	81.30
MLPModel_glo_pro sum alerts min: 3	69.04	45.38	81.51	58.47	43.17	76.80
MLPModel_glo_pro sum alerts min: 4	70.63	53.78	79.83	57.23	47.72	71.76

Table 2: Summary of process killing models, validation and test set score metrics [Table 1 of 3]

	val			test		
model	f1	tnr	tpr	f1	tnr	tpr
MLPModel_pro	71.43	56.30	79.83	57.54	52.53	69.38
MLPModel_pro mean process tree	72.18	57.14	80.67	57.06	53.53	67.97
MLPModel_pro process tree min alerts: 1	72.32	54.62	82.35	57.41	49.41	71.06
MLPModel_pro process tree min alerts: 2	72.32	54.62	82.35	57.41	49.41	71.06
MLPModel_pro process tree min alerts: 3	72.32	54.62	82.35	57.41	49.41	71.06
MLPModel_pro process tree min alerts: 4	72.32	54.62	82.35	57.41	49.41	71.06
MLPModel_pro rolling mean window: 2	71.77	66.39	74.79	48.88	63.18	50.35
MLPModel_pro rolling mean window: 3	72.65	68.91	74.79	49.23	63.71	50.57
MLPModel_pro rolling mean window: 4	72.34	73.95	71.43	46.40	67.05	45.26
MLPModel_pro sum alerts min: 2	73.86	72.27	74.79	47.14	66.40	46.50
MLPModel_pro sum alerts min: 3	73.68	78.99	70.59	45.27	68.12	43.36
MLPModel_pro sum alerts min: 4	73.30	82.35	68.07	44.48	69.63	41.73
MLPModel_pro_tree	71.38	31.82	97.48	64.36	21.07	92.52
MLPRegression_pro_process	38.89	53.78	35.29	54.83	48.73	67.05
MLPRegression_pro_process mean process tree	37.32	57.14	32.77	56.75	57.37	65.15
NBModel_glo_pro	67.25	9.24	96.64	55.07	10.36	89.49
NBModel_glo_pro mean process tree	67.25	9.24	96.64	55.62	12.69	89.38
NBModel_glo_pro process tree min alerts: 1	67.25	9.24	96.64	55.00	10.18	89.43
NBModel_glo_pro process tree min alerts: 2	67.25	9.24	96.64	55.00	10.18	89.43
NBModel_glo_pro process tree min alerts: 3	67.25	9.24	96.64	55.00	10.18	89.43
NBModel_glo_pro process tree min alerts: 4	67.25	9.24	96.64	55.00	10.18	89.43
NBModel_glo_pro rolling mean window: 2	67.69	19.33	92.44	55.20	15.10	87.05
NBModel_glo_pro rolling mean window: 3	67.73	26.05	89.08	55.32	17.32	86.02
NBModel_glo_pro rolling mean window: 4	67.76	31.09	86.55	54.28	21.08	81.68
NBModel_glo_pro sum alerts min: 2	68.17	27.73	89.08	55.70	19.40	85.64
NBModel_glo_pro sum alerts min: 3	67.99	31.93	86.55	54.42	22.27	81.30
NBModel_glo_pro sum alerts min: 4	68.03	36.97	84.03	51.32	25.39	73.44
NBModel_pro	67.06	8.40	96.64	55.60	7.03	92.63
NBModel_pro mean process tree	67.06	8.40	96.64	56.17	9.61	92.41
NBModel_pro process tree min alerts: 1	67.06	8.40	96.64	55.56	6.78	92.68
NBModel_pro process tree min alerts: 2	67.06	8.40	96.64	55.56	6.78	92.68
NBModel_pro process tree min alerts: 3	67.06	8.40	96.64	55.56	6.78	92.68
NBModel_pro process tree min alerts: 4	67.06	8.40	96.64	55.56	6.78	92.68
NBModel_pro rolling mean window: 2	67.69	19.33	92.44	56.01	13.41	89.81
NBModel_pro rolling mean window: 3	67.52	25.21	89.08	56.18	15.81	88.78
NBModel_pro rolling mean window: 4	67.99	31.93	86.55	54.94	19.61	83.90
NBModel_pro sum alerts min: 2	68.61	29.41	89.08	56.52	18.14	88.13
NBModel_pro sum alerts min: 3	68.21	32.77	86.55	55.27	21.30	83.63
NBModel_pro sum alerts min: 4	67.81	37.82	83.19	52.25	24.42	75.77
NBModel_pro_tree	66.10	10.61	98.32	61.25	8.63	92.69
NBModel_pro_tree mean process tree	66.10	10.61	98.32	61.25	8.63	92.69
RFModel_glo_pro	89.68	83.19	94.96	76.43	67.19	92.52
RFModel_glo_pro mean process tree	90.48	84.03	95.80	76.34	67.66	91.92
RFModel_glo_pro process tree min alerts: 1	90.91	84.03	96.64	69.45	50.56	92.95
RFModel_glo_pro process tree min alerts: 2	90.91	84.03	96.64	69.45	50.56	92.95
RFModel_glo_pro process tree min alerts: 3	90.91	84.03	96.64	69.45	50.56	92.95
RFModel_glo_pro process tree min alerts: 4	90.91	84.03	96.64	69.45	50.56	92.95
RFModel_glo_pro rolling mean window: 2	92.70	94.96	90.76	80.77	78.88	89.38
RFModel_glo_pro rolling mean window: 3	91.30	94.96	88.24	80.19	78.67	88.51
RFModel_glo_pro rolling mean window: 4	90.27	95.80	85.71	79.86	82.86	83.69
RFModel_glo_pro sum alerts min: 2	91.30	94.96	88.24	81.50	81.53	87.97
RFModel_glo_pro sum alerts min: 3	90.27	95.80	85.71	79.99	84.01	82.76
RFModel_glo_pro sum alerts min: 4	88.79	95.80	83.19	76.11	85.37	75.01
RFModel_pro	92.37	87.39	96.64	74.57	62.71	92.95
RFModel_pro mean process tree	92.74	88.24	96.64	74.79	64.04	92.20
RFModel_pro process tree min alerts: 1	92.74	88.24	96.64	68.75	48.23	93.39
RFModel_pro process tree min alerts: 2	92.74	88.24	96.64	68.75	48.23	93.39
RFModel_pro process tree min alerts: 3	92.74	88.24	96.64	68.75	48.23	93.39
RFModel_pro process tree min alerts: 4	92.74	88.24	96.64	68.75	48.23	93.39
RFModel_pro rolling mean window: 2	93.22	94.12	92.44	78.28	73.83	89.76
RFModel_pro rolling mean window: 3	91.38	94.12	89.08	77.47	73.25	88.78
RFModel_pro rolling mean window: 4	89.96	94.12	86.55	77.08	77.70	83.85
RFModel_pro sum alerts min: 2	91.38	94.12	89.08	77.98	74.65	88.40
RFModel_pro sum alerts min: 3	89.96	94.12	86.55	77.05	77.52	83.96
RFModel_pro sum alerts min: 4	88.50	94.12	84.03	73.11	79.35	75.61
RFModel_pro_tree	90.35	82.58	98.32	74.20	52.44	92.74
RFRegression_pro_process	91.94	87.39	95.80	74.77	66.05	90.35
SVMModel_glo_pro	65.23	15.97	89.08	57.34	24.24	86.23
SVMModel_glo_pro mean process tree	65.23	15.97	89.08	58.11	27.39	85.91
SVMModel_glo_pro process tree min alerts: 1	65.23	15.97	89.08	57.32	23.81	86.45
SVMModel_glo_pro process tree min alerts: 2	65.23	15.97	89.08	57.32	23.81	86.45
SVMModel_glo_pro process tree min alerts: 3	65.23	15.97	89.08	57.32	23.81	86.45
SVMModel_glo_pro process tree min alerts: 4	65.23	15.97	89.08	57.32	23.81	86.45
SVMModel_glo_pro rolling mean window: 2	65.15	26.05	84.03	57.98	33.52	81.84
SVMModel_glo_pro rolling mean window: 3	64.65	31.09	80.67	58.14	35.46	80.98
SVMModel_glo_pro rolling mean window: 4	64.31	38.66	76.47	56.76	40.37	75.34
SVMModel_glo_pro sum alerts min: 2	65.05	36.13	78.99	58.35	39.08	79.13
SVMModel_glo_pro sum alerts min: 3	64.75	42.02	75.63	57.05	43.24	74.15
SVMModel_glo_pro sum alerts min: 4	64.89	51.26	71.43	54.70	47.40	67.59
SVMModel_pro	66.47	5.88	96.64	56.92	10.33	93.71

Table 3: Summary of process killing models, validation and test set score metrics [Table 2 of 3]

		val			test	
SVMModel_pro mean process tree	67.25	9.24	96.64	57.55	13.34	93.33
SVMModel_pro process tree min alerts: 1	66.47	5.88	96.64	56.21	7.28	93.88
SVMModel_pro process tree min alerts: 2	66.47	5.88	96.64	56.21	7.28	93.88
SVMModel_pro process tree min alerts: 3	66.47	5.88	96.64	56.21	7.28	93.88
SVMModel_pro process tree min alerts: 4	66.47	5.88	96.64	56.21	7.28	93.88
SVMModel_pro rolling mean window: 2	66.87	15.97	92.44	58.60	22.02	90.30
SVMModel_pro rolling mean window: 3	67.30	24.37	89.08	58.82	24.42	89.27
SVMModel_pro rolling mean window: 4	67.99	31.93	86.55	57.98	28.97	84.66
SVMModel_pro sum alerts min: 2	67.96	28.57	88.24	59.52	27.61	88.73
SVMModel_pro sum alerts min: 3	68.90	35.29	86.55	59.06	33.35	84.12
SVMModel_pro sum alerts min: 4	68.75	41.18	83.19	56.68	38.87	76.10
SVMModel_pro_tree	65.73	9.09	98.32	61.79	9.88	93.19
dqn	51.71	72.27	44.54	27.74	55.50	26.94
random_search_glo_pro_RNN	87.69	77.31	95.80	71.83	59.63	90.24
random_search_glo_pro_RNN mean process tree	88.03	78.15	95.80	72.50	61.67	89.81
random_search_glo_pro_RNN_Regression	85.71	72.27	95.80	72.44	61.78	89.59
random_search_pro_RNN	91.20	85.71	95.80	72.63	59.63	91.82
random_search_pro_RNN mean process tree	91.20	85.71	95.80	73.03	60.92	91.49
random_search_pro_RNN_Regression	88.37	78.99	95.80	72.71	60.70	91.06
random_search_pro_RNN_tree	88.19	80.67	94.12	73.72	65.79	88.56

Table 4: Summary of process killing models, validation and test set score metrics [Table 3 of 3]

References

[1] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems*, pages 86–103. Springer, 2013.

[2] S. Afroz. How to build realistic machine learning systems for security?, Jan. 2020. Online [Accessed 02-March-2020] `https://www.usenix.org/conference/enigma2020/presentation/afroz`.

[3] M. Ahmadi, A. Sami, H. Rahimi, and B. Yadegari. Malware detection by behavioural sequential patterns. *Computer Fraud & Security*, 2013(8):11–19, 2013.

[4] F. Ahmed, H. Hameed, M. Z. Shafiq, and M. Farooq. Using spatio-temporal information in api calls with machine learning algorithms for malware detection. In *Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence*, pages 55–62. ACM, 2009.

[5] M. Alazab, R. Layton, S. Venkataraman, and P. Watters. Malware detection based on structural and behavioural features of api calls. *Proceedings of the 1st international cyber resilience conference*, 2010.

[6] Y. Alosefer and O. Rana. Honeyware: A web-based low interaction client honeypot. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 410–417, April 2010. doi: 10.1109/ICSTW.2010.41.

[7] B. Alsulami, A. Srinivasan, H. Dong, and S. Mancoridis. Lightweight behavioral malware detection for windows platforms. In *Malicious and Unwanted Software (MALWARE), 2017 12th International Conference on*, pages 75–81. IEEE, 2017.

[8] V. M. Alvarez. Writing yara rules, 2015. [Online [Accessed 30-September-2020] https://yara.readthedocs.io/en/v3.4.0/writingrules.html.

[9] Amazon.com. Amazon laptops, October 2018. Online [Accessed 05-October-2018] https://www.amazon.com/Notebooks-Laptop-Computers/b?ie=UTF8&node=565108.

[10] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane. Graph-based malware detection using dynamic analysis. *Journal in computer Virology*, 7(4):247–258, 2011.

[11] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck. Dos and don'ts of machine learning in computer security. *arXiv preprint arXiv:2010.09470*, 2020.

[12] A. Bacci, A. Bartoli, F. Martinelli, E. Medvet, F. Mercaldo, and C. A. Visaggio. Impact of code obfuscation on android malware detection based on static and dynamic analysis. In *ICISSP*, pages 379–385, 2018.

[13] M. B. Bahador, M. Abadi, and A. Tajoddin. Hpcmalhunter: Behavioral malware detection using hardware performance counters and singular value decomposition. In *Computer and Knowledge Engineering (ICCKE), 2014 4th International eConference on*, pages 703–708. IEEE, 2014.

[14] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *International Workshop on Recent Advances in Intrusion Detection*, pages 178–197. Springer, 2007.

[15] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11. Citeseer, 2009.

[16] U. Bayer, E. Kirda, and C. Kruegel. Improving the efficiency of dynamic malware analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 1871–1878. ACM, 2010.

[17] C. Belgaid, A. d'Azémar, G. Fieni, and R. Rouvoy. Pyrapl, 12 2019. Software version 0.2.3.1: `https://pypi.org/project/pyRAPL/`.

[18] J. Benda, A. Longtin, and L. Maler. Spike-frequency adaptation separates transient communication signals from background oscillations. *Journal of Neuroscience*, 25(9):2312–2321, 2005.

[19] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

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

[21] A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral detection of malware on mobile handsets. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, pages 225–238. ACM, 2008.

[22] R. R. Branco, G. N. Barbosa, and P. D. Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. *Black Hat*, 1, 2012.

[23] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly,

B. Holt, and G. Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[24] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.

[25] P. Burnap, R. French, F. Turner, and K. Jones. Malware classification using self organising feature maps and machine activity data. *computers & security*, 73: 399–410, 2018.

[26] P. Burnap, R. French, F. Turner, and K. Jones. Malware classification using self organising feature maps and machine activity data. *Computers & Security*, 73: 399–410, 2018.

[27] G. Canfora, A. Di Sorbo, F. Mercaldo, and C. A. Visaggio. Obfuscation techniques against signature-based detection: A case study. In *2015 Mobile Systems Technologies Workshop (MST)*, pages 21–26, 2015.

[28] R. Canzanese, M. Kam, and S. Mancoridis. Toward an automatic, online behavioral malware classification system. In *Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on*, pages 111–120. IEEE, 2013.

[29] D. Carlin, P. O'Kane, and S. Sezer. A cost analysis of machine learning using dynamic runtime opcodes for malware detection. *Computers & Security*, 85:138–155, 2019.

[30] J. Cervantes, X. Li, and W. Yu. Svm classification for large data sets by considering models of classes distribution. In *2007 Sixth Mexican International Confer-*

ence on Artificial Intelligence, Special Session (MICAI), pages 51–60, 2007. doi: 10.1109/MICAI.2007.27.

[31] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[32] D. H. □. Chau, C. Nachenberg, J. Wilhelm, A. Wright, and C. Faloutsos. Polonium: Tera-scale graph mining and inference for malware detection. In *Proceedings of the 2011 SIAM International Conference on Data Mining*, pages 131–142. SIAM, 2011.

[33] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE international conference on dependable systems and networks with FTCS and DCC (DSN)*, pages 177–186. IEEE, 2008.

[34] Y. Chen, Z. Shan, F. Liu, G. Liang, B. Zhao, X. Li, and M. Qiao. A gene-inspired malware detection approach. In *Journal of Physics: Conference Series*, volume 1168, page 062004. IOP Publishing, 2019.

[35] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[36] F. Chollet. Keras, 2015. URL `https://github.com/fchollet/keras`.

[37] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[38] C. Cimpanu. Ransomware mentioned in 1000 sec filings over the past year, 05

2020. [Online [Accessed 11-February-2021] https://www.zdnet.com/article/ransomware-mentioned-in-1000-sec-filings-over-the-past-year/].

[39] Cisco. Snort, 2013. URL https://www.snort.org/.

[40] A. Continella, A. Guagnelli, G. Zingaro, G. De Pasquale, A. Barenghi, S. Zanero, and F. Maggi. Shieldfs: a self-healing, ransomware-aware filesystem. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 336–347. ACM, 2016.

[41] D. Corpora. Govdocs digital corpus, 2019. Online [Accessed 15-Feburary-2019], http://downloads.digitalcorpora.org/corpora/files/govdocs1/zipfiles/.

[42] M. Corporation. Lock pages in memory, 2017. Online [Accessed 23-January-21 https://docs.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/lock-pages-in-memory.

[43] R. Correa. How fast does ransomware encrypt files? faster than you think, April 2016. URL https://blog.barkly.com/how-fast-does-ransomware-encrypt-files.

[44] M. A. Cusumano. The changing labyrinth of software pricing. *Communications of the ACM*, 50(7):19–22, 2007.

[45] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu. Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3422–3426. IEEE, 2013.

[46] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[47] A. Damodaran, F. Di Troia, C. A. Visaggio, T. H. Austin, and M. Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12, 2017.

[48] S. Das, Y. Liu, W. Zhang, and M. Chandramohan. Semantics-based online malware detection: Towards efficient real-time protection against malware. *IEEE transactions on information forensics and security*, 11(2):289–302, 2016.

[49] O. E. David and N. S. Netanyahu. Deepsign: Deep learning for automatic malware signature generation and classification. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, July 2015. doi: 10.1109/IJCNN.2015.7280815.

[50] O. E. David and N. S. Netanyahu. Deepsign: Deep learning for automatic malware signature generation and classification. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2015.

[51] Z. Dehlawi and N. Abokhodair. Saudi arabia's response to cyber conflict: A case study of the shamoon malware incident. In *2013 IEEE International Conference on Intelligence and Security Informatics*, pages 73–75. IEEE, 2013.

[52] K. Dempsey, V. Pillitteri, C. Baer, R. Niemeyer, R. Rudman, and S. Urban. Assessing information security continuous monitoring (iscm) programs: Developing an iscm program assessment, 2020-05-21 00:05:00 2020.

[53] O. E. Dictionary. ”robust, adj. and n.”. URL `https://www.oed.com/view/Entry/166651`. Online: [Accessed 1st June 2022] `https://www.oed.com/view/Entry/166651`.

[54] Y. Ding, X. Xia, S. Chen, and Y. Li. A malware detection method based on family behavior graph. *Computers & Security*, 73:73–86, 2018.

[55] Dissent. Leon medical center confirms ransomware attack, credits employees and staff with providing quality care despite attack, 2020. Online [Accessed 02-January-2021] `https://www.databreaches.net/leon-medical-center-confirms-ransomware-attack-credits-employees-and-staff-with-providing-quality-care-despite-attack/`.

[56] Dissent. Uk: Transform hospital group falls prey to ransomware attack, 2020. Online [Accessed 02-January-2021] `https://www.databreaches.net/uk-transform-hospital-group-falls-prey-to-ransomware-attack/`.

[57] E. M. Dovom, A. Azmoodeh, A. Dehghantanha, D. E. Newton, R. M. Parizi, and H. Karimipour. Fuzzy pattern tree for edge malware detection and categorization in iot. *Journal of Systems Architecture*, 2019.

[58] R. Durve and A. Bouridane. Windows 10 security hardening using device guard whitelisting and applocker blacklisting. In *2017 Seventh International Conference on Emerging Security Technologies (EST)*, pages 56–61. IEEE, 2017.

[59] Y. Fang, B. Yu, Y. Tang, L. Liu, Z. Lu, Y. Wang, and Q. Yang. A new malware classification approach based on malware dynamic analysis. In J. Pieprzyk and S. Suriadi, editors, *Information Security and Privacy*, pages 173–189, Cham, 2017. Springer International Publishing. ISBN 978-3-319-59870-3.

[60] P. Faruki, V. Laxmi, M. S. Gaur, and P. Vinod. Behavioural detection with api call-grams to identify malicious pe files. In *Proceedings of the First International Conference on Security of Internet of Things*, pages 85–91. ACM, 2012.

[61] I. Firdausi, C. lim, A. Erwin, and A. S. Nugroho. Analysis of machine learning techniques used in behavior-based malware detection. In *2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies*, pages 201–203, Dec 2010. doi: 10.1109/ACT.2010.33.

[62] W. Fleshman, E. Raff, R. Zak, M. McLean, and C. Nicholas. Static malware detection & subterfuge: Quantifying the robustness of machine learning and current anti-virus. In *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 1–10. IEEE, 2018.

[63] P. S. Foundation. Psutil python library, 2017. URL `https://pypi.python.org/pypi/psutil`.

[64] E. Gandotra, D. Bansal, and S. Sofat. Malware analysis and classification: A survey. *Journal of Information Security*, 5(02):56, 2014.

[65] A. Gangwal, E. Leverett, E. Jardine, E. Burns, and D. Geer. Averages don't characterise the heavy tails of ransoms. *eCrime Research*, 08 2020.

[66] I. Gashi, V. Stankovic, C. Leita, and O. Thonnard. An experimental study of diversity with off-the-shelf antivirus engines. In *2009 Eighth IEEE International Symposium on Network Computing and Applications*, pages 4–11. IEEE, 2009.

[67] I. Ghafir, M. Hammoudeh, V. Prenosil, L. Han, R. Hegarty, K. Rabie, and F. J. Aparicio-Navarro. Detection of advanced persistent threat using machine-learning correlation analysis. *Future Generation Computer Systems*, 89:349–359, 2018.

[68] GlobalStats. Market share of windows operating system versions, 2018. Online [Accessed 10-November-2018] `http://gs.statcounter.com/os-version-market-share/windows/desktop/worldwide#monthly-201708-201709-bar`.

[69] GlobalStats. Operating system market share, 2020. Online [Accessed 05-September-2020] `https://gs.statcounter.com/os-market-share`.

[70] A. Goldblum. What algorithms are most successful on kaggle?, 2016. Online [Accessed 20-Sept-2020] `https://www.kaggle.com/antgoldbloom/what-algorithms-are-most-successful-on-kaggle`.

[71] V. Golovko, S. Bezobrazov, P. Kachurka, and L. Vaitsekhovich. Neural network and artificial immune systems for malware and network intrusion detection. In *Advances in machine learning II*, pages 485–513. Springer, 2010.

[72] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[73] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294. ACM, 2012.

[74] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 2016.

[75] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435*, 2016.

[76] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel. Adversarial examples for malware detection. In S. N. Foley, D. Gollmann, and E. Snekkenes, editors, *Computer Security – ESORICS 2017*, pages 62–79, Cham, 2017. Springer International Publishing. ISBN 978-3-319-66399-9.

[77] C. Guarnieri, A. Tanasi, J. Bremer, and M. Schloesser. The cuckoo sandbox, 2012.

[78] J. T. Haller. Portable apps, 2019.

[79] X. Han, T. F. J. Pasquier, A. Bates, J. Mickens, and M. I. Seltzer. UNICORN: runtime provenance-based detector for advanced persistent threats. *CoRR*, abs/2001.01525, 2020. URL `http://arxiv.org/abs/2001.01525`.

[80] S. S. Hansen, T. M. T. Larsen, M. Stevanovic, and J. M. Pedersen. An approach for detection and family classification of malware based on behavioral analysis. In *2016 International Conference on Computing, Networking and Communications (ICNC)*, pages 1–5, Feb 2016. doi: 10.1109/ICCNC.2016.7440587.

[81] W. Hardy, L. Chen, S. Hou, Y. Ye, and X. Li. Dl4md: A deep learning framework for intelligent malware detection. In *Proceedings of the International Conference on Data Mining (DMIN)*, page 61. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2016.

[82] Hiscox. Uk small businesses targeted with 65,000 attempted cyber attacks per day, 2018.

[83] T. K. Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995.

[84] H. Holm. Signature based intrusion detection for zero-day attacks: (not) a closed chapter? In *2014 47th Hawaii International Conference on System Sciences*, pages 4895–4904, Jan 2014. doi: 10.1109/HICSS.2014.600.

[85] W. Hu and Y. Tan. Black-box attacks against rnn based malware detection algorithms. In *Workshops at the mThirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[86] H.-D. Huang, C.-S. Lee, H.-Y. Kao, Y.-L. Tsai, and J.-G. Chang. Malware behavioral analysis system: Twman. In *Intelligent Agent (IA), 2011 IEEE Symposium on*, pages 1–8. IEEE, 2011.

[87] W. Huang and J. W. Stokes. Mtnet: A multi-task neural network for dynamic malware classification. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*,

DIMVA 2016, pages 399–418, New York, NY, USA, 2016. Springer-Verlag New York, Inc. ISBN 978-3-319-40666-4. doi: 10.1007/978-3-319-40667-1_20. URL `http://dx.doi.org/10.1007/978-3-319-40667-1_20`.

[88] Y. Huang, U. Verma, C. Fralick, G. Infantec-Lopez, B. Kumar, and C. Woodward. Malware evasion attack and defense. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 34–38. IEEE, 2019.

[89] M. Ijaz, M. H. Durad, and M. Ismail. Static and dynamic malware analysis using machine learning. In *2019 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, pages 687–691, 2019.

[90] M. Imran, M. T. Afzal, and M. A. Qadir. Using hidden markov model for dynamic malware analysis: First impressions. In *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, pages 816–821, Aug 2015. doi: 10.1109/FSKD.2015.7382048.

[91] K. Inc. Kaggle, 2020. Online [Accessed 20-September-2020] `https://www.kaggle.com/`.

[92] R. Islam, R. Tian, L. M. Batten, and S. Versteeg. Classification of malware based on integrated static and dynamic features. *Journal of Network and Computer Applications*, 36(2):646–656, 2013.

[93] T. Isohara, K. Takemori, and A. Kubota. Kernel-based behavior analysis for android malware detection. In *Computational Intelligence and Security (CIS), 2011 Seventh International Conference on*, pages 1011–1015. IEEE, 2011.

[94] K. Jickling. Ransomware downed uvm medical center systems, but no payment made, 2020. Online accessed [2nd January 2021] `https:`

//vtdigger.org/2020/12/22/ransomware-downed-uvm-medical-center-systems-but-no-payment-made/.

[95] J. Kang, S. Jang, S. Li, Y.-S. Jeong, and Y. Sung. Long short-term memory-based malware classification method for information security. *Computers & Electrical Engineering*, 77:366–375, 2019.

[96] N. Karampatziakis, J. W. Stokes, A. Thomas, and M. Marinescu. Using file relationships in malware classification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–20. Springer, 2012.

[97] H. J. Kelley. Gradient theory of optimal flight paths. *Ars Journal*, 30(10):947–954, 1960.

[98] Y. Ki, E. Kim, and H. K. Kim. A novel approach to detect malware based on api call sequence analysis. *International Journal of Distributed Sensor Networks*, 11 (6):659101, 2015.

[99] T. Kim, B. Kang, and E. G. Im. Runtime detection framework for android malware. *Mobile Information Systems*, 2018, 2018.

[100] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im. A multimodal deep learning method for android malware detection using various features. *IEEE Transactions on Information Forensics and Security*, 14(3):773–788, 2019.

[101] J. Kinable and O. Kostakis. Malware classification based on call graph clustering. *Journal in computer virology*, 7(4):233–245, 2011.

[102] D. P. Kingma and J. Ba. Adam: 'a' method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980.

[103] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

[104] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX security symposium*, volume 4, pages 351–366, 2009.

[105] B. Kolosnjaji, A. Zarras, T. Lengyel, G. Webster, and C. Eckert. Adaptive semantics-aware malware classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 419–439. Springer, 2016.

[106] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert. *Deep Learning for Classification of Malware System Call Sequences*, pages 137–149. Springer International Publishing, Cham, 2016. ISBN 978-3-319-50127-7. doi: 10.1007/978-3-319-50127-7_11.

[107] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert. Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence*, pages 137–149. Springer, 2016.

[108] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European Signal Processing Conference (EUSIPCO)*, pages 533–537. IEEE, 2018.

[109] D. Kong and G. Yan. Discriminant malware distance learning on structural information for automated malware classification. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1357–1365. ACM, 2013.

[110] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[111] J. Leblond. L'hôpital d'albertville victime d'une cyberattaque, 2020. Online [Accessed 02-January-2021] `https://www.francebleu.fr/infos/societe/l-hopital-d-albertville-est-victime-d-une-cyberattaque-1608729976`.

[112] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[113] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an, and H. Ye. Significant permission identification for machine-learning-based android malware detection. *IEEE Transactions on Industrial Informatics*, 14(7):3216–3225, 2018.

[114] L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

[115] S. Linnainmaa. The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. *Master's Thesis (in Finnish), Univ. Helsinki*, pages 6–7, 1970.

[116] Z. C. Lipton. A critical review of recurrent neural networks for sequence learning. *CoRR*, abs/1506.00019, 2015. URL `http://arxiv.org/abs/1506.00019`.

[117] L. Liu, B.-s. Wang, B. Yu, and Q.-x. Zhong. Automatic malware classification and new malware detection using machine learning. *Frontiers of Information Technology & Electronic Engineering*, 18(9):1336–1347, 2017.

[118] G. LLC. Google scholar, 2020. URL `https://scholar.google.co.uk/`. Online [Accessed 12/3/2020] `https://scholar.google.co.uk/`.

[119] P. I. LLC. Ninth annual cost of cybercrime study: Unlocking the value of improved cybersecurity protection, 2019.

[120] A. M. Lungana-Niculescu, A. Colesa, and C. Oprisa. False positive mitigation in behavioral malware detection using deep learning. In *2018 IEEE 14th International Conference on Intelligent Computer Communication and Processing (ICCP)*, pages 197–203. IEEE, 2018.

[121] Z. Ma, H. Ge, Y. Liu, M. Zhao, and J. Ma. A combination method for android malware detection based on control flow graphs and machine learning algorithms. *IEEE access*, 7:21235–21245, 2019.

[122] A. Makandar and A. Patrot. Malware analysis and classification using artificial neural network. In *2015 International conference on trends in automation, communications and computing technology (I-TACT-15)*, pages 1–6. IEEE, 2015.

[123] C. Martin. Protecting the uk from the increasing cyber threat - the next steps, 2018.

[124] D. McKelvey. Security incident handling in an academic medical center, 1995.

[125] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, et al. Deep android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 301–308. ACM, 2017.

[126] M. Melis, D. Maiorca, B. Biggio, G. Giacinto, and F. Roli. Explaining black-box android malware detection. In *2018 26th European Signal Processing Conference (EUSIPCO)*, pages 524–528. IEEE, 2018.

[127] S. Mittal and S. Vaishay. A survey of techniques for optimizing deep learning on gpus. *Journal of Systems Architecture*, 99:101635, 2019.

[128] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[129] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[130] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[131] R. Mosli, R. Li, B. Yuan, and Y. Pan. Automated malware detection using artifacts in forensic memory images. In *2016 IEEE Symposium on Technologies for Homeland Security (HST)*, pages 1–6. IEEE, 2016.

[132] M. Mowbray. Moral status for malware! the difficulty of defining advanced artificial intelligence. *Cambridge Quarterly of Healthcare Ethics*, 30(3):517–528, 2021. doi: 10.1017/S0963180120001061.

[133] N. Naik, P. Jenkins, N. Savage, L. Yang, T. Boongoen, N. Iam-On, K. Naik, and J. Song. Embedded yara rules: strengthening yara rules utilising fuzzy hashing and fuzzy rules for malware analysis. *Complex & Intelligent Systems*, 7(2):687–702, 2021.

[134] S. Nari and A. A. Ghorbani. Automated malware classification based on network behavior. In *2013 International Conference on Computing, Networking and Communications (ICNC)*, pages 642–647. IEEE, 2013.

[135] L. Nataraj, V. Yegneswaran, P. Porras, and J. Zhang. A comparative assessment of malware classification using binary texture analysis and dynamic analysis. In *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, AISec '11, pages 21–30, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1003-1. doi: 10.1145/2046684.2046689. URL `http://doi.acm.org/10.1145/2046684.2046689`.

[136] M. Neugschwandtner, P. M. Comparetti, G. Jacob, and C. Kruegel. Forecast:

skimming off the malware cream. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 11–20. ACM, 2011.

[137] M. Nunes, P. Burnap, O. Rana, P. Reinecke, and K. Lloyd. Getting to the root of the problem: A detailed comparison of kernel and user level data for dynamic malware analysis. *Journal of Information Security and Applications*, 48:102365, 2019.

[138] I. OPSWAT. Windows anti-malware market share report, 2019. URL `https://metadefender.opswat.com/reports/anti-malware-market-share?date=Latest&lang=en`.

[139] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev. Malware-aware processors: A framework for efficient online malware detection. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 651–661, Feb 2015. doi: 10.1109/HPCA.2015.7056070.

[140] H. H. Pajouh, A. Dehghantanha, R. Khayami, and K.-K. R. Choo. Intelligent os x malware threat detection with code inspection. *Journal of Computer Virology and Hacking Techniques*, 14(3):213–223, 2018.

[141] D. Palmer. Petya ransomware: Cyberattack costs could hit $300m for shipping giant maersk, 2017. Online Accessed [03 January 2020] `https://www.zdnet.com/article/petya-ransomware-cyber-attack-costs-could-hit-300m-for-shipping-giant-maersk/`.

[142] N. Papernot. Characterizing the limits and defenses of machine learning in adversarial settings, 2018. PhD thesis.

[143] H. Pareek, S. Romana, and P. Eswari. Application whitelisting: approaches and

challenges. *International Journal of Computer Science, Engineering and Information Technology (IJCSEIT)*, 2(5):13–18, 2012.

[144] Y. Park, D. Reeves, V. Mulukutla, and B. Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, page 45. ACM, 2010.

[145] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas. Malware classification with recurrent networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1916–1920, April 2015. doi: 10.1109/ICASSP.2015.7178304.

[146] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[147] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

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

[149] N. Peiravian and X. Zhu. Machine learning for android malware detection using permission and api calls. In *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*, pages 300–305. IEEE, 2013.

[150] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 729–746, 2019.

[151] PolySwarm. Trending malware report, 2021. From Subscription malinglist dated 15th April 2021.

[152] B. Quintero, E. Martínez, V. Manuel Álvarezv, K. Hiramoto, J. Canto, and A. Bermúdez. Virustotal, 2004. URL `https://virustotal.com/`.

[153] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas. Malware detection by eating a whole exe. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[154] D. Rajeswaran, F. Di Troia, T. H. Austin, and M. Stamp. Function call graphs versus machine learning for malware detection. In *Guide to Vulnerability Analysis for Computer Networks and Systems*, pages 259–279. Springer, 2018.

[155] M. Rhode. Code for 'early-stage malware prediction using recurrent neural networks', 2017. URL `https://github.com/mprhode/malware-prediction-rnn`.

[156] M. Rhode, P. Burnap, and K. Jones. Early-stage malware prediction using recurrent neural networks. *Computers & Security*, 77:578–594, 2018.

[157] M. Rhode, P. Burnap, and K. Jones. Real-time malware process detection and automated process killing. *arXiv preprint arXiv:1902.02598*, 2019.

[158] M. Rhode, L. Tuson, P. Burnap, and K. Jones. Lab to soc: Robust features for dynamic malware detection. In *2019 49th Annual IEEE/IFIP International*

Conference on Dependable Systems and Networks–Industry Track, pages 13–16. IEEE, 2019.

[159] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 108–125, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70541-3. doi: 10.1007\/978-3-540-70542-0_6. URL `http:\/\/dx.doi.org\/10.1007\/978-3-540-70542-0_6`.

[160] K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.

[161] I. Rosenberg, A. Shabtai, L. Rokach, and Y. Elovici. Generic black-box end-to-end attack against rnns and other api calls based malware classifiers. *arXiv preprint arXiv:1707.05970*, 2017.

[162] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[163] M. Rosulek et al. Hash functions. *The Joy of Cryptography OE (1st)*, 2017.

[164] Z. Salehi, A. Sami, and M. Ghiasi. Using feature generation from api calls for malware detection. *Computer Fraud & Security*, 2014(9):9–18, 2014.

[165] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze. Malware detection based on mining api calls. In *Proceedings of the 2010 ACM symposium on applied computing*, pages 1020–1025. ACM, 2010.

[166] I. Santos, J. Devesa, F. Brezo, J. Nieves, and P. G. Bringas. Opem: A static-dynamic approach for machine-learning-based malware detection. In *Interna-*

tional Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions, pages 271–280. Springer, 2013.

[167] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, 15(1):83–97, Jan 2018. ISSN 1545-5971. doi: 10.1109/TDSC.2016.2536605.

[168] J. Saxe and K. Berlin. Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 11–20, Oct 2015. doi: 10.1109/MALWARE.2015.7413680.

[169] J. Saxe and H. Sanders. *Malware Data Science: Attack Detection and Attribution*. No Starch Press, 2018.

[170] H. Sayadi, A. Houmansadr, S. Rafatirad, H. Homayoun, et al. Comprehensive assessment of run-time hardware-supported malware detection using general and ensemble learning. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pages 212–215. ACM, 2018.

[171] H. Sayadi, N. Patel, S. M. PD, A. Sasan, S. Rafatirad, and H. Homayoun. Ensemble learning for effective run-time hardware-based malware detection: A comprehensive analysis and classification. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[172] N. Scaife, H. Carter, P. Traynor, and K. R. Butler. Cryptolock (and drop it): stopping ransomware attacks on user data. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 303–312. IEEE, 2016.

[173] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K. A. Yuksel, S. A. Camtepe, and S. Albayrak. Static analysis of executables for collaborative mal-

ware detection on android. In *2009 IEEE International Conference on Communications*, pages 1–5. IEEE, 2009.

[174] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[175] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. “andromaly”: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.

[176] H. Shi, J. Mirkovic, and A. Alwabel. Handling anti-virtual machine techniques in malicious software. *ACM Trans. Priv. Secur.*, 21(1), Dec. 2017. ISSN 2471-2566. doi: 10.1145/3139292. URL `https://doi.org/10.1145/3139292`.

[177] T. Shibahara, T. Yagi, M. Akiyama, D. Chiba, and T. Yada. Efficient dynamic malware analysis based on network behavior using deep learning. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7, Dec 2016. doi: 10.1109/GLOCOM.2016.7841778.

[178] P. Shijo and A. Salim. Integrated static and dynamic analysis for malware detection. *Procedia Computer Science*, 46:804–811, 2015.

[179] R. Sihwail, K. Omar, and K. Z. Ariffin. A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis. *International Journal on Advanced Science, Engineering and Information Technology*, 8(4-2):1662, 2018.

[180] M. Sikorski and A. Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.

[181] C. L. Smith. Understanding concepts in the defence in depth strategy. In *IEEE 37th Annual 2003 International Carnahan Conference onSecurity Technology, 2003. Proceedings.*, pages 8–16. IEEE, 2003.

[182] SoftAntenna. Msys2 installer, 2019. Online [Accessed 24-October-2020] `https://www.softantenna.com/softwares/7115-msys2-installer`.

[183] Softonic. Softonic.com, 2017. URL `https://en.softonic.com/`.

[184] S. Sonnenburg, G. Rätsch, S. Henschel, C. Widmer, J. Behr, A. Zien, F. d. Bona, A. Binder, C. Gehl, and V. Franc. The shogun machine learning toolbox. *The Journal of Machine Learning Research*, 11:1799–1802, 2010.

[185] sophos, 2020. URL `"https://www.sophos.com/en-us/medialibrary/Gated-Assets/white-papers/sophos-the-state-of-ransomware-2020-wp.pdf"`. Online [Accessed 22-July-20] `https://www.sophos.com/en-us/medialibrary/Gated-Assets/white-papers/sophos-the-state-of-ransomware-2020-wp.pdf`.

[186] SourceForge. Sourceforge.net, 2017. URL `https://sourceforge.net/`.

[187] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[188] R. Sun, X. Yuan, P. He, Q. Zhu, A. Chen, A. Grégio, D. A. S. de Oliveira, and X. Li. Learning fast and slow: PROPEDEUTICA for real-time malware detection. *CoRR*, abs/1712.01145, 2017. URL `http://arxiv.org/abs/1712.01145`.

[189] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine learning proceedings 1990*, pages 216–224. Elsevier, 1990.

[190] R. S. Sutton, A. G. Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[191] G. J. Széles and A. Coleşa. Malware clustering based on called api during runtime. In *International Workshop on Information and Operational Technology Security Systems*, pages 110–121. Springer, 2018.

[192] S. M. Tabish, M. Z. Shafiq, and M. Farooq. Malware detection using statistical analysis of byte-level file content. In *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, pages 23–31. ACM, 2009.

[193] A. Tamersoy, K. Roundy, and D. H. Chau. Guilt by association: large scale malware detection by mining file-relation graphs. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1524–1533. ACM, 2014.

[194] R. Tian, R. Islam, L. Batten, and S. Versteeg. Differentiating malware from cleanware using behavioural analysis. In *2010 5th International Conference on Malicious and Unwanted Software*, pages 23–30, Oct 2010. doi: 10.1109/MALWARE.2010.5665796.

[195] R. Tian, R. Islam, L. Batten, and S. Versteeg. Differentiating malware from cleanware using behavioural analysis. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 23–30. IEEE, 2010.

[196] S. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi. Malware detection with deep neural network using process behavior. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 577–582, June 2016. doi: 10.1109/COMPSAC.2016.151.

[197] A. Tsymbal. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*, 106(2):58, 2004.

[198] D. o. H. UK Government National Audit Office. Investigation: Wannacry cyber at-

tack and the nhs, Oct 2017. URL https://www.nao.org.uk/wp-content/uploads/2017/10/Investigation-WannaCry-cyber-attack-and-the-NHS.pdf.

[199] T. Urban, D. Tatang, T. Holz, and N. Pohlmann. Towards understanding privacy implications of adware and potentially unwanted programs. In *European Symposium on Research in Computer Security*, pages 449–469. Springer, 2018.

[200] V. Vapnik and A. Lerner. Pattern recognition using generalized portrait method. *Automation and Remote Control*, 24:774–780, 1963.

[201] VirusShare.com. Virusshare.com, 2017. URL https://virusshare.com/.

[202] VirusTotal. About us - virustotal, 2019. URL https://support.virustotal.com/hc/en-us/categories/360000160117-About-us. Online [Accessed 21-August-2019] https://support.virustotal.com/hc/en-us/categories/360000160117-About-us.

[203] VirusTotal. Statistics - virustotal, 2020. URL https://www.virustotal.com/en/statistics/. Online [Accessed 25-March-2020] https://www.virustotal.com/en/statistics/.

[204] M. Wadkar, F. Di Troia, and M. Stamp. Detecting malware evolution using support vector machines. *Expert Systems with Applications*, 143:113022, 2020.

[205] G. Wagener, A. Dulaunoy, et al. Malware behaviour analysis. *Journal in computer virology*, 4(4):279–287, 2008.

[206] P. Wang and Y.-S. Wang. Malware behavioural detection and vaccine development by using a support vector model classifier. *Journal of Computer and System Sciences*, 81(6):1012–1026, 2015.

[207] Q. Wang, W. Guo, K. Zhang, A. G. Ororbia II, X. Xing, X. Liu, and C. L. Giles. Adversary resistant deep neural networks with an application to malware detection.

In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1145–1153. ACM, 2017.

[208] W. Wang, M. Zhu, X. Zeng, X. Ye, and Y. Sheng. Malware traffic classification using convolutional neural network for representation learning. In *2017 International Conference on Information Networking (ICOIN)*, pages 712–717. IEEE, 2017.

[209] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[210] C. J. C. H. Watkins. Learning from delayed rewards, 1989. PhD thesis.

[211] M. R. Watson, A. K. Marnerides, A. Mauthe, D. Hutchison, et al. Malware detection in cloud computing infrastructures. *IEEE Transactions on Dependable and Secure Computing*, 13(2):192–205, 2016.

[212] I. H. Witten and E. Frank. Data mining: practical machine learning tools and techniques with java implementations. *Acm Sigmod Record*, 31(1):76–77, 2002.

[213] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69. IEEE, 2012.

[214] W.-C. Wu and S.-H. Hung. Droiddolphin: a dynamic android malware detection framework using big data and machine learning. In *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*, pages 247–252, 2014.

[215] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. K. Sangaiah. Android malware detection based on system call sequences and lstm. *Multimedia Tools and Applications*, 78(4):3979–3999, 2019.

[216] H. Xu, C. Caramanis, and S. Mannor. Sparse algorithms are not stable: A no-free-lunch theorem. *IEEE transactions on pattern analysis and machine intelligence*, 34(1):187–193, 2011.

[217] Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar. A survey on malware detection using data mining techniques. *ACM Computing Surveys (CSUR)*, 50(3):41, 2017.

[218] Y. Ye, L. Chen, S. Hou, W. Hardy, and X. Li. Deepam: a heterogeneous deep learning framework for intelligent malware detection. *Knowledge and Information Systems*, 54(2):265–285, 2018.

[219] C.-W. Yeh, W.-T. Yeh, S.-H. Hung, and C.-T. Lin. Flattened data in convolutional neural networks: Using malware detection as case study. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, pages 130–135. ACM, 2016.

[220] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, pages 297–300, Nov 2010. doi: 10.1109/BWCCA.2010.85.

[221] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE, 2010.

[222] X. Yuan. Phd forum: deep learning-based real-time malware detection with multi-stage analysis. In *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–2. IEEE, 2017.

[223] Z. Yuan, Y. Lu, and Y. Xue. Droiddetector: android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, 21(1): 114–123, 2016.

[224] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014.

[225] Y. Zhang, J. Pang, F. Yue, and J. Cui. Fuzzy neural network for malware detect. In *2010 International Conference on Intelligent System Design and Engineering Application*, volume 1, pages 780–783. IEEE, 2010.

[226] X. Zhu, C. Vondrick, C. C. Fowlkes, and D. Ramanan. Do we need more training data? *International Journal of Computer Vision*, 119(1):76–92, 2016.