Capgemini

MASTER THESIS

Dynamic detection of mobile malware using real-life data and machine learning

UNIVERSITY OF TWENTE.

J.S. Panman de Wit

August 17, 2018

Master Thesis

August 2018

Master Business Information Technology
Track: Data Science & Business
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente

Author

J.S. Panman de Wit

Graduation Committee

Dr. J. van der Ham
Faculty EEMCS, University of Twente

Dr. D. Bucur
Faculty EEMCS, University of Twente

Prof. Dr. M. Junger
Faculty BMS, University of Twente

S. Steensma, MSc.
Capgemini NL

Credits cover photo:
Original picture created by Rawpixel.com - Freepik.com
Screen image created by Freepik
Credits LaTeX template:
Latex template from LaTeXTemplates.com, originally created by Steve R. Gunn, and modified by Sunil Patel

Abstract

Mobile malwares are malicious programs that target mobile devices, which are an increasing problem. This is reflected by the rise of detected mobile malware samples per year. Additionally, the number of active smartphone users is expected to grow, stressing the importance of research on the detection of mobile malware.

Detection methods for mobile malware exists, although methods are still limited and incomprehensive. In this paper, we propose detection methods that use device information such as the CPU usage, battery usage, and memory usage for the detection of 10 subtypes of Mobile Trojans. The focus of this paper is the Android Operating System (OS) as it is dominating the mobile device industry with an 80 per cent market share.

This research uses a dataset containing device and malware data of 47 users for an entire year (2016) to create multiple mobile malware detection methods. By using real-life data this research provides a realistic assessment of its detection methods. Additionally, using this dataset we examine which features, i.e. aspects, of a device, are most important in detecting (subtypes of) Mobile Trojans. The performance of the following machine learning classifiers are assessed: Random Forest, K-Nearest neighbour, Naïve Bayes, Multilayer perceptron, and AdaBoost. All classifiers are assessed using a 4-fold cross-validation with holdout method. Additionally, the hyperparameters of all classifiers are tuned with the use of a GridSearch. Furthermore, we assess performances of classifiers when one model is trained for all subtypes of Mobile Trojans, and when separate models are trained for each subtype of Mobile Trojans.

Our results show that the Random Forest classifier is most suited for the detection of Mobile Trojans. The Random Forest classifier achieves an f1 score of 0,73 with an False Positve Rate (FPR) of 0.009 and False Negative Rate (FNR) of 0.380 when one model is created to detect all 10 subtypes of Mobile Trojans. Furthermore, our research shows that the Random Forest, K-nearest neighbour classifier, and AdaBoost classifiers achieve, on average, an f1 score > 0.72, an FPR of <0.02 and an FNR <0.33, when models are created separately for each subtype of Mobile Trojans. Moreover, we examine the usability of the different detection methods. By assessing multiple metrics such as the model size and training times, we analyse whether the methods can be deployed locally on devices. Lastly, we examine the cost and benefits, for businesses, associated with deploying self-made detection methods.

Acknowledgements

This thesis could not have been completed without the contribution and help of multiple persons.

First of all I would like to share my appreciation for my supervisors Dr. J. van der Ham, Dr. D. Bucur, and Prof. Dr. M. Junger for their outstanding guidance throughout my thesis process. Their contribution was crucial in improving the quality of this thesis. Additionally, I would like to thank Prof. Dr. L. Cavallaro from the Royal Holloway University of London. He was not part of the graduation committee nor part of the University of Twente. Nevertheless, he was open to share his expertise on mobile security through multiple Skype sessions. These sessions helped improve the quality of this thesis.

Furthermore, I owe a lot of thanks to Capgemini which provided me both with a working place and many interesting people to discuss my findings with. My special thanks goes out to S. Steensma, who has guided me within Capgemini and helped me to focus on the right matters throughout the process of working on my thesis.

Moreover, I would like to thank the Ben-Gurion University, that provided me with the dataset used in this research.

Lastly, I would like to thank my family for their support the past 10 months.

Sebastian Panman de Wit

Utrecht, August 2018

Contents

Chapter 1

Introduction

Nowadays smartphones have become an integral part of life, with people using their phone in both their private and professional life. There is an estimated of 2.6 billion active smartphone users globally at the time of writing, and this number is expected to grow by one billion by 2020 [1]. The rise in smartphone users has also led to an increase in malicious programs targeting mobile devices, i.e. mobile malware. Criminals try to exploit vulnerabilities on smartphones of other people for their own purposes. Additionally, over the past years malware authors have become less recreational-driven and more profit-driven as they are actively searching for sensitive, personal, and enterprise information [2].

Academic work is mainly divided into dynamic analysis and static analysis of mobile malware. Dynamic analysis refers to the analysis of malware during run-time, i.e. while the application is running. Static analysis refers to the analysis of malware outside run-time, e.g. by analysing the installation package of a malware. Dynamic analysis has advantages over static analysis but methods are still imperfect, ineffective, and incomprehensive [3]. An important limitation is that most studies developed malware detection methods based on analysis in virtual environments, e.g. analysis on a PC, instead of real mobile devices. An increasing trend is seen in malware that use techniques to avoid detection in virtual environments, thereby making methods based on analysis in virtual environments less effective than methods based on analysis on real devices [2]. Moreover, we found that most methods are assessed with i) malware running isolated in an emulator, and ii) malware running for a brief period. This kind of assessment does not reflect the circumstances of a real device with for example different applications running at the same time. Therefore, most research does not provide a realistic assessment of detection performances of their detection methods due to their unrealistic circumstances.

This paper compares the performance of multiple mobile malware detection methods, with real-life circumstances, on the detection of 10 different mobile malware types. The focus of this paper is on Android devices as this platform is dominating the mobile device industry with a market share of more than 80 percent [4]. The Sherlock dataset by the Ben-Gurion University [5] is used, containing malware data and device data of 47 users throughout the year 2016. At the moment of writing, no other research is known to us that used data with this high amount of real life users over a period of this extend. The malware data are logs of actions taken by different subtypes of Mobile Trojans, i.e. malware showing benign behaviour and performing hidden malicious actions. The device data are logs of system metrics of the devices, e.g. CPU usage, memory usage, battery usage. Tracking the system metrics did not require any adjustments to the Android Operating System (OS) such as rooting, i.e. adjusting the OS to allow for kernel-level control. This allows the detection methods of this research to be used on the majority of Android devices, as more than 95% of the Android devices are unrooted [6]. The dataset is used to train the following machine learning classifiers: i) Random Forest, ii) Naïve Bayes, iii) K-nearest neighbour, and iv) Multilayer Perceptron. The classifiers are trained to predict, given the system metrics of a device at a given moment, whether a Mobile Trojan is executing benign or malicious actions on a device. Taking the aforementioned real-life approach, this research provides a realistic assessment of detection methods and valuable knowledge on detecting mobile malware on real devices.

1.1 Research questions

This research uses the following main research question to address the current limitations of dynamic detection methods:

M.Q. 1 *How can we improve the dynamic detection of Mobile Trojans using hardware and software features (not requiring any root permissions), based on real-life data?*

The main research question is formulated based on an extensive literature research which is described in Sections 2.4 and 2.5. The findings of the literature research lead to the following four focus areas: i) dynamic detection ii) Mobile Trojans, iii) hardware and software features, features not requiring any root permissions, and iv) real-life data. The focus on dynamic detection is chosen because of its advantages over static analysis, which are described in Section 2.4.1. Mobile Trojans are the most prevalent malware type on Android devices and is therefore chosen; more on this can be found in Section 2.1.1. Hardware features and software features, not requiring any root permissions, are chosen because these features are present in the dataset used in this research. Additionally, as stated in the introduction of this Section, focusing on features not requiring any root permissions allows the detection methods of this research to be used on the majority of Android devices. Lastly, the focus on real-life data allows for i) realistic assessment of detection methods and ii) valuable insights on detecting mobile malware on real devices.

The following sub-questions are formulated to help answer the main research question:

S.Q. 1 *How do different machine learning techniques such as Random Forest, K-Nearest Neighbour, Naïve Bayes, and Multilayer Perceptrons, perform in detecting Mobile Trojans?*

The Random Forest, K-Nearest Neighbour, and Naive Bayes classifiers showed the most promising results in the literature that was consulted for this research. Neural networks, though scantily researched for the detection of mobile malware, show promising results[7]. Therefore, Neural Networks will be examined in this research together with the aforestated classifiers. Related works on dynamic mobile malware detection and the performances of the classifiers in these works can be found In Section 2.5. The answer to S.Q.1 is described in Chapter 6.

S.Q. 2 *What software and/or hardware features, that do not require root permissions, are the most crucial for the detection of Mobile Trojans?*

Mobile devices are limited in resources such as battery, CPU, and RAM capacity. Therefore examining which features are the most crucial in the detection of mobile malware, and which features can be excluded, improves the efficiency of the detection models. Additionally, the answer to this sub-question provides insights in which features are important in the detection of different subtypes of Mobile Trojans. Because these feature insights are drawn from real-life data, the findings reflect real-life circumstances rather than (clean) laboratory environments. The answer to S.Q.2 is described in Section 6.

S.Q. 3 *What is the usability of these different classifiers on a real device?*

This sub-question focuses on the usability of the different classifiers, given the aforementioned resource limitations. Usability refers to the system resource consumption (e.g. battery usage, RAM usage) of the different detection models. Usability from a business perspective is also analysed in S.Q.3. The costs and benefits for a business, associated with using, or not using, self-made mobile malware detection methods are examined. The usability regarding resources and the business usability are described in Chapter 7.

1.2 Research method and report structure

A research method is devised to answer the research questions in a structured manner. This research methodology is based on CRISP-DM, a widely used data science methodology [8]. This paper is organized according to the research methodology shown in Figure 1.1. The research methodology and the report structure is described below.

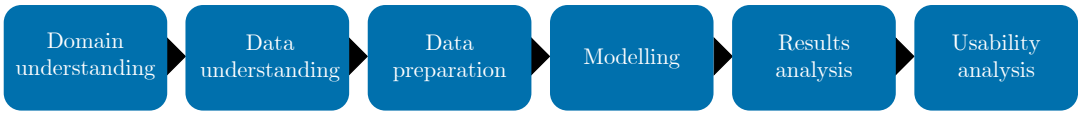


FIGURE 1.1: Research methodology

Domain understanding

This phase is needed to understand the domain of mobile malware. Relevant literature on mobile malware detection is found during this phase. Additionally, the impact of mobile malware on businesses is analysed. Furthermore, recent industry developments in mobile malware detection methods are examined. Chapter 2 contains the findings of this phase.

Data understanding

The dataset used in this research is provided by an external party. Therefore this phase is required to understand the content of the dataset provided. The dataset content is explored with the use of multiple visualisations such as histograms. This phase also consists of verifying the data quality. Chapter 3 contains the findings of this phase.

Data preparation

Multiple preparation steps are needed to construct a dataset that can be used for the creation of detection models. Chapter 4 describe the steps taken during this phase.

Modelling

This phase consists of selecting machine learning techniques, setting up experiments, and training and testing of the machine learning techniques. Chapter 5 describes the steps taken during this phase.

Results analysis

The results of the experiments and feature analysis are collected and documented during this phase. This phase presents the results needed to answer the sub-questions *S.Q.1* and *S.Q.2*. Chapter 6 contains the findings of this phase.

Usability analysis

This phase consists of analysing the usability of the detection models. The usability of detection models on real devices is analysed, using multiple metrics such as the training and testing times of classifiers. Additionally, the business usability of the detection models is examined with a cost-benefit analysis. This phase results in the answer to *S.Q.3*. Chapter 7 describes the findings of this phase.

Then Chapter 8 discusses the results of Chapters 6 and 7, and the limitations of this research. Lastly, Chapter 9.1 concludes with the answers to the research questions and suggest potential future work on this research.

Chapter 2

Background

Each subsection of this chapter describes the necessary background knowledge for a specific subsection of this thesis, to understand its content. The related subsections are shown in Figure 2.1.

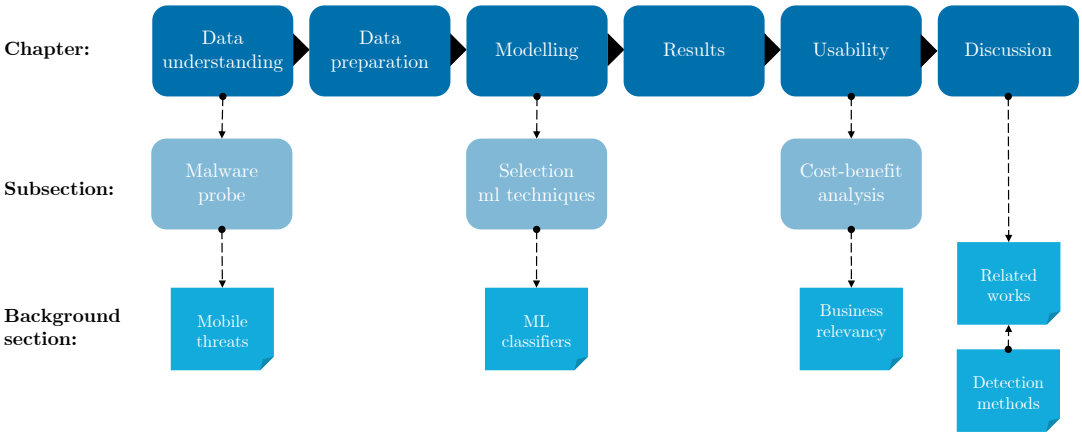


FIGURE 2.1: Background chapter overview

2.1 Mobile threats

Mobile malware differs from traditional (PC) malware. Below, the most relevant differences are listed based on [2].

- Mobile devices cross physical and network domains exposing them to more malware such as mobile worms. This kind of malware uses the physical movement of devices in order to propagate across networks.
- Most mobile devices have high application turnover due to the high availability of apps.
- The input methods of mobile devices increase the complexity of analysis. Touch commands such as swiping and tapping allow for more different input commands than the traditional mouse and keyboard input. This complicates the analysis of all possible input commands.
- Mobile devices are resource limited with for example a limited battery, CPU, and RAM capacity.
- Mobile devices are susceptible to a wide array of vulnerabilities due to their different ways of connecting to the outside world and the different types of technologies they use. Different connection methods such as Wifi, GPRS, 3G, Bluetooth, make the device more vulnerable. Additionally, the different technologies such as the camera, speaker, make the mobile device more susceptible to vulnerabilities through for example the drivers of these technologies.

The next section describes the different types of mobile malware.

2.1.1 Mobile malware types

To categorize the different mobile malware threats, this research uses the malware type classification of Google [9], shown in Table 2.1. This Table shows only the malware types examined in this research.

Malware type	Malicious behaviour description
Trojan	Appears benign but performs malicious activity without user's knowledge.
Adware	Shows advertisements to the user in an unexpected manner, e.g. on the home screen.
Denial of service (DOS)	Executes, or is part of, a cyber-attack (DOS attack) without user's knowledge.
Hostile downloader	Not malicious itself but downloads malware.
Phishing	Appears trustworthy and requests user authentication credentials, but sends the data to a third party.
Privilege escalation	Breaks the application sandbox or changes access to core security-related features, therefore compromising the integrity of the system.
Ransomware	Takes partial or complete control of system and/or data and asks for a payment to release control and/or data.
Spyware	Transmits sensitive data off the device.

**The adware type is not included in the Google classification as it 'does not put the device at risk'[6]. This research however, does include this type because adware performs unwanted behaviour on a device and is therefore malicious.*

TABLE 2.1: Malware classification

The actual distribution of the different types of malware is hard to estimate as detection numbers of Antivirus (AV) vendors rather reflect the efficacy of its detection methods than the actual distribution. However, using different sources helps in giving an impression of the Android malware ecosystem. Figure 2.2 shows the distribution of different types of malware according to the latest security report of Google [9] (left) and of the latest security report by Kaspersky [10] (right). Although Kaspersky uses a different terminology, both figures show the Trojan type being the most common malware. Note that malware types are not mutually exclusive.

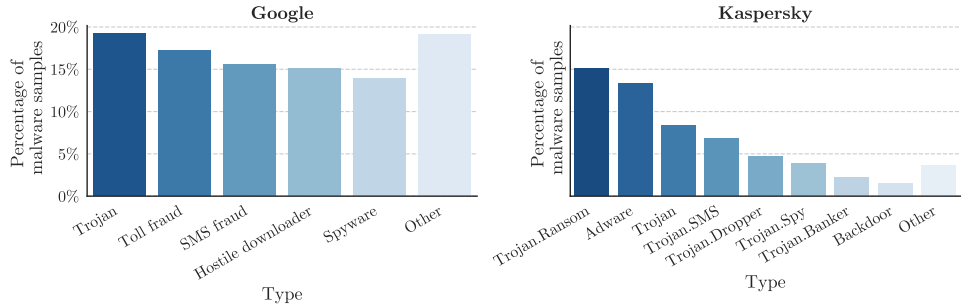


FIGURE 2.2: Malware type distribution according to Google [9] (left) and Kaspersky [10] (right)

2.1.2 Android security

Android is an open-source platform for mobile devices. Applications for Android are written in Java and compiled to Dalvik bytecode. An application can also contain native libraries, which can be invoked from the Java code. To install an application, the application needs to be in the form of a signed APK package. This package contains different files belonging to the application. The AndroidManifest file in the APK package describes the different permissions required by the applications. Permissions are required by an app to access sensitive APIs. These sensitive APIs allow the application to access system resources such as Bluetooth functions, location data, SMS or MMS functions, and data functions. Once installed, the application runs in an Application Sandbox as a separate process with a unique user ID. By default, applications cannot read any files of other applications but can only use interprocess communication mechanisms to communicate with each other. These mechanisms and a more elaborate description of the Android framework is given in Appendix D.

2.2 Machine learning classifiers

The definition for machine learning used throughout this research is: "the complex computation process of automatic pattern recognition and intelligent decision making based on training sample data" [11]. A more general definition of machine learning is "the process of applying a computing-based resource to implement learning algorithms" [11]. Based on different books on machine learning [11][12][13][14], the basic theory of the different Machine Learning techniques used in this research is described in this section.

Three categories of learning algorithms are: supervised learning, unsupervised learning, and semi-supervised learning. In supervised learning, the goal is to create a model which predicts y based on some x, given a training set consisting of examples pairs of (x_i, y_i). Here y_i is called the label of the example x_i. When y is continuous, the problem at hand is called a regression problem, and when y is discrete the problem at hand is called a classification problem. Throughout this research, the focus is on supervised learning as we try to detect whether a device described by some features x, contains malware that is performing malicious actions. In this case, the prediction value y takes the value 1 if a malicious application is performing malicious actions on the device and 0 if no malicious actions are performed on the device. The next Section describe the machine learning classifiers used in this research. Then Section 2.2.6 describes the metrics used to evaluate classifiers. Lastly, Section 2.2.7 describes the challenges of using machine learning to create mobile malware detection methods.

2.2.1 Random Forest

The Random Forest (RF) classifier is an ensemble classifier that uses multiple decision tree classifiers to classify test instances. An example of a decision tree is shown in Figure 2.3.

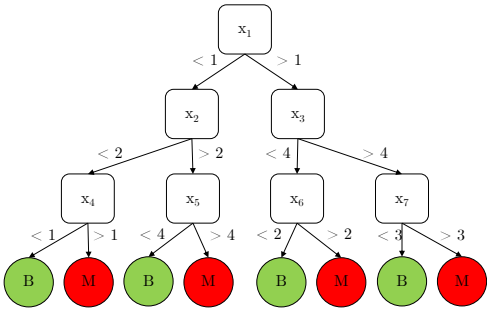


FIGURE 2.3: Example of Decision Tree

A major disadvantage of decision trees is their instability. Decision trees are known for high variance and often a small change in the data can cause a large change in the final tree. Random Forests try to reduce the variance of decision trees by taking multiple decision tree classifiers to classify testing instances. Then, classification is done using a majority vote among all the decision trees. Some advantages of Random Forest are i) it overcomes overfitting ii) it can deal with high-dimensional data. Disadvantages include i) accuracy depends on the number of trees ii) it is sensitive to an imbalanced dataset [3].

2.2.2 Naïve Bayes

Naïve Bayes (NB) is a statistical classifier that uses Bayes's theorem to predict the probability of given query instance belonging to a certain class. Bayes's theorem, also called Bayes's rule, calculates the probability of a hypothesis H being true, given some evidence e, according to the following formula:

$$P(H|e) = \frac{P(e|H) * P(H)}{P(e)}$$

where

$P(H|e)$ denotes the posterior probability of H, conditioned on e
$P(e|H)$ denotes the posterior probability of e conditioned on H
$P(H)$ denotes the prior probability of H
$P(e)$ denotes the prior probability of e

The classifier is called naïve because it assumes conditional independence, making the computation of the above formula less computationally expensive; especially for datasets with many features. Although Naïve Bayes assumes conditional independence, it performs well in domains where independence is violated [14]. Advantages of Naive Bayes are: i) high speed

ii) insensitive to irrelevant feature data iii) simple and mature algorithm. A disadvantage is that it requires the assumption of independence of features [3].

2.2.3 K-Nearest Neighbour

The K-nearest neighbour (KNN) is a distance-based classifier. Distance-based classifiers generalise from training data to unseen data by looking at similarities between training instances. Given a query instance q, the classifier finds the k training instances, the closest in distance to the query instance q. Subsequently, it classifies the query instance using a majority vote among the k neighbours. The distance from the query instance to its training instances can be calculated using different metrics such as the Euclidean distance, Minkowski distance, or Manhatten distance. An example of the k-nearest neighbour classification is given in Figure 2.4.

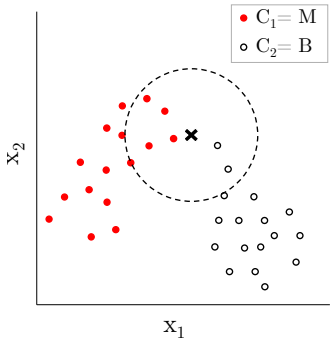


FIGURE 2.4: Example of K-Nearest Neighbour Classification

Advantages of KNN are [3]: i) high precision and accuracy ii) non-linear classification iii) no assumption of features. The disadvantages are i) it is sensitive to unbalanced sample set, ii) it is computational expensive.

2.2.4 Artificial neural networks

Artificial neural networks (ANN) is a machine-learning model that uses a structure of nodes, i.e. artificial neurons, to classify testing instances. These nodes are connected to each other by directed links. An ANN consists of an input layer, some hidden layers, and an output layer. Every directed link between neurons has some numeric weight shown as w_{ij} in the example ANN, shown in Figure 2.5. These numeric weights are used in the activation function of each node. This activation function is used to determine the output of a node. Different learning algorithms can be used to determine the number of hidden layers, the number of neurons, and the weights between the neurons. Some of the most popular are feed-forward back-propagation and radial basis function networks. This research uses the Multilayer Perceptron (MLP) classifier which is a class of ANN that uses backpropagation for learning.

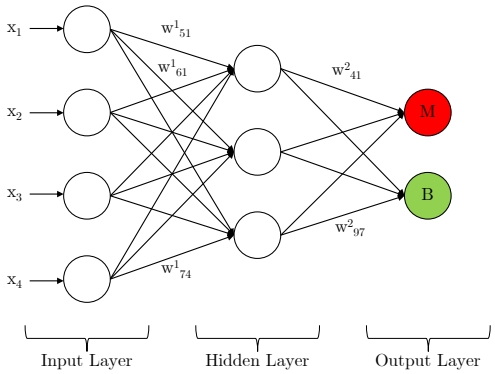


FIGURE 2.5: Example of an Artificial Neural Network

2.2.5 AdaBoost

Adaptive boosting (AdaBoost or Ada) is, like the Random Forest classifier, an ensemble classifier. AdaBoost uses multiple training iterations on subsets of the dataset to boost the accuracy of a (weak) machine learning classifier. The machine-learning classifier is first trained on a subset of the dataset. Then all training instances are weighted, with any sample not correctly classified in the training set being weighted more, thereby having a higher probability of being chosen in the training set of the next iteration. Likewise, any sample correctly classified is weighted less. This process is repeated until the set maximum number of estimators is reached. AdaBoost is known for offering accurate machine-learning classifiers [11]. However, a disadvantage of AdaBoost is that it is a greedy learning, i.e. offering suboptimal solutions. In this research, AdaBoost is used with (standard) decision trees.

2.2.6 Evaluation classifiers

Different performance metrics exist to evaluate a classifier. The most basic performance metrics are summarized in a confusion matrix. The design of a confusion matrix is shown in Table 2.2.

		Predicted class	
		Malicious	Benign
Actual Class	Malicious	True Postive (TP)	False Negative (FN)
	Benign	False Positive (FP)	True Negative (TN)

TABLE 2.2: Confusion Matrix

The confusion matrix shows how many malware instances were correctly classified as being malware (TP), how many malware instances were missed (FP), how many benign instances were correctly classified as being benign (TN), and how many benign classes were incorrectly classified (FN).

Other metrics and their formula are shown in Table 2.3. These metrics use the metrics shown in Table 2.2. A frequently used metric is the accuracy of a malware, defined by the percentage of correct predictions ($TP + TN$), of the total predictions ($TP + TN + FP + FN$). This metric, however, might not reflect the performance of a classifier well. In a skewed dataset, that is a dataset containing more of one class than the other, high accuracy can be achieved by always predicting the majority class. For example in a dataset consisting of 90% malicious actions and 10% benign actions, always predicting malicious actions results in an accuracy of 90%. In the case of a skewed dataset, the performance metrics Precision (PPV) and/or Recall (TPR), reflect the performance of a classifier more realistic. The harmonic mean of the Precision and Recall are reflected in the f1 score (F-score with $\alpha = 1$).

Metric	**Formula**
Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$
True Positive Rate (TPR)	$\frac{TP}{TP+FN}$
False Positive Rate (FPR)	$\frac{FP}{FP+TN}$
True Negative Rate (TNR)	$\frac{TN}{TN+FP}$
Precision (PPV)	$\frac{TP}{TP+FP}$
F-score (F-measure)	$(1+\alpha^2)(\frac{PPV*TPR}{\alpha^2(PPV+TPR)})$

TABLE 2.3: Performance Metrics

2.2.7 Automated detection

Two relevant challenges of using machine learning to create mobile malware detection methods are: i) the use of imbalanced datasets and ii) concept drift. Both concepts are described below.

Imbalanced dataset

Cybersecurity data is skewed most of the times, containing more benign data than malicious data. This results in a few challenges while training and testing machine learning classifiers. First, standard machine learning techniques are often biased towards the majority class in an imbalanced dataset [11]. Hence, standard metrics such as the accuracy do not reflect the actual performance of a model well [11]. In a skewed dataset containing 95% benign examples and 5% malicious examples, an accuracy of 95% might be the result of the classifier predicting benign labels 100% of the time. This research addresses this challenge by using metrics that take into account the skewness of a dataset, such as the f1 score which is the harmonic mean between the True Postive Rate and True Negative Rate.

Concept drift
The inability of detection models, trained on older malware, to detect new rapid evolving malware, is called concept drift [15]. A way to overcome this issue is to continuously retrain the models, based on new information.

2.3 Business relevancy

The increasing adoption of mobile devices in the workplace, rise in mobile cyber attacks on businesses, and recent legislation, show that mobile security in the workplace is becoming more relevant for businesses. These developments are described in more detail below.

1. Increasing adoption of mobile devices in the workplace:
A recent industry study on the adoption of mobile devices in the workplace shows that nearly 80% of the employees are using a mobile device for business purposes [16].

2. Rise in mobile cyber attacks on businesses:
A recent industry study surveying 588 IT security professionals from the Global 200 companies in the U.S. report that 67 per cent of the respondents said it was certain or likely that their organization had a data breach as a result of a mobile device used by an employee [17]. Another study from a cybersecurity company securing 500 devices of 850 organization show that 100 per cent of the organization experienced at least one mobile malware attack from July 2016 to July 2017.

3. Increased legislation on personal data protection:
A recent development increasing the importance of mobile security in the workplace is the recent General Data Protection Regulation (GDPR), enforced since May 25, 2018. This regulation controls the "processing by an individual, a company or an organisation of personal data relating to individuals in the EU" [18]. A recent study by Gartner predicts that by 2019, 30 per cent of organizations will face "significant financial exposure from regulatory bodies due to their failure to comply with GDPR requirements to protect personal data on mobile devices" [19][20].

To view how the detection methods in this research fit with the cybersecurity-related activities of business, the cybersecurity framework of The National Institute of Standards and Technology (NIST) [21] is used (shown in Figure 2.6). This framework help businesses manage cybersecurity-related risks. In this Section the framework is used to show in which activities, the detection methods of this research provide business value. Section 7.2 then describes a cost-benefit analysis of the created detection models from a business perspective.

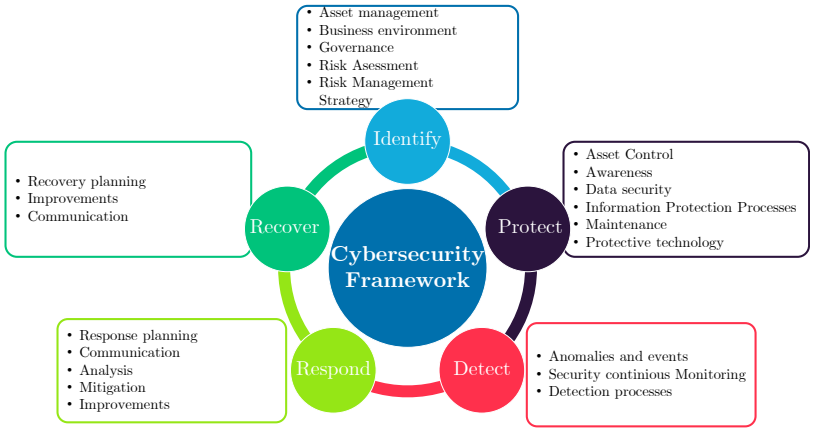


FIGURE 2.6: NIST Cybersecurity framework

The Cybersecurity framework of NIST identifies five main functions to manage cybersecurity-related risks. The detection methods created in this research fit within the *detect* category. This category is described as: 'develop and implement appropriate activities to identify the occurrence of a cybersecurity event'. Note that this research limits itself to only this category

and is not concerned with any of the other categories such as the protection, or recovering of mobile malware threats.

2.4 Mobile malware detection methods

There are numerous ways to detect mobile malware on smartphones. The taxonomy used in this research is a combination of the taxonomy of [3] and [22], and shown in Figure 2.7.

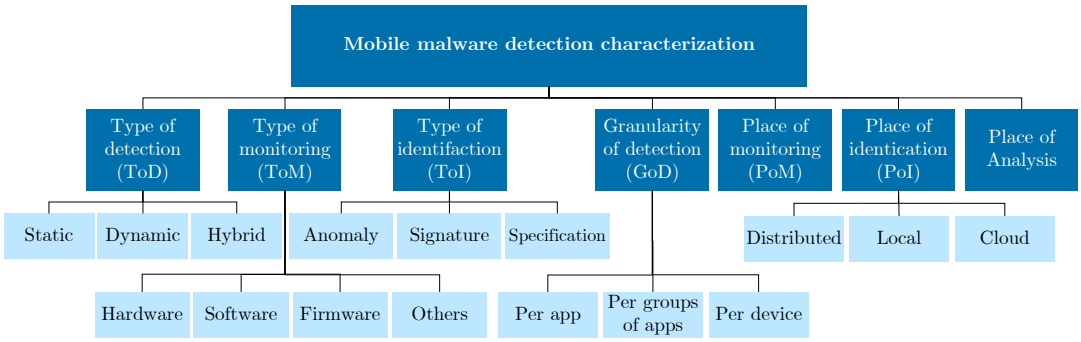


FIGURE 2.7: Mobile malware detection taxonomy

Figure 2.7 shows that detection methods are classified depending on the way the methods are designed. Below, the characterizations of the detection methods and their brief description is described.

Characterization	Description
Type of detection	The approach taken to collect features by the detection method.
Type of monitoring	The features being monitored / analysed by the detection method.
Type of identification	The way malware is identified by the detection method.
Granularity of detection	How fine or coarse, data is being analysed by the detection method.
Place of monitoring	
Place of identifcation	Where the different steps of the detection method take place.
Place of analysis	

TABLE 2.4: Mobile malware detection characterization description

2.4.1 Type of detection

The biggest differentiation in mobile malware detection methods is made regarding the approach to collect features [3]. There are three approaches to collect features: i) static, ii) dynamic, and iii) hybrid. Static methods try to detect malware without executing applications. In contrast, dynamic methods execute the application, and analysis occurs during run-time. A combination of static and dynamic analysis is called a hybrid approach. The biggest limitation to static analysis is that this type of analysis is susceptible to obfuscation techniques that remove or limit access to the code of malware. Additionally, other techniques such as the injection of non-java code, network activity, and the modifications of objects during runtime, are only visible at run-time. These limitations make them less effective towards zero-day vulnerabilities [2]. The limitations of static analysis can be solved using dynamic analysis methods, as these analyse applications during run-time. Drawbacks of dynamic analysis are that these methods are mostly accompanied with high false positive rates and are heavy on system resources [3]. Additionally, there are some drawbacks when dynamic analysis is done with the use of virtual environments, more on this in the paragraph below, describing the place of monitoring. Because static analysis is less effective on zero-day attacks and recently more Android malware samples are using techniques to prevent effective static analysis [2], this research focuses on the dynamic analysis of mobile malware.

2.4.2 Type of monitoring

The type of monitoring is defined by the features used within a mobile malware detection method. These features act as an input to the analysis of the detection method. Features can be categorized into three classes: i) hardware, ii) software, and iii) firmware. Hardware features are features that can be monitored and are specific to a device, e.g. battery, CPU, and memory features. Software features are characteristics that can be monitored during the run-time of software or by examining the software package, e.g. permissions, privileges, and network traffic. Firmware features are features from programs using read-only memory. Most firmware features require rooting privileges in the Android OS.
Table 2.5 shows an overview regarding the features used in dynamic mobile malware detection methods. This table is made using a recent literature review on dynamic mobile malware detection methods [3] and was consulted during the preliminary literature research of this research. During the preliminary literature research, few articles were found that focused on hardware features. Therefore, additional literature was searched on detection methods using hardware features. These articles are described in Section 2.5.

Category	Feature	Papers
Hardware	**Battery**	[23], [24], [25]
	CPU	[23], [24], [26]
	Memory	[23], [24], [26]
Software	**Permissions**	[24], [26], [27], [28], [29], [30], [31]
	Network Traffic	[32], [33], [34], [35]
	Information Flow	[36], [37]
	Covert Channel	[38]
Firmware	**System Calls**	[24], [28], [39], [40], [41], [42], [43], [44], [45], [46]
	API	[28], [31], [39], [43], [47]
	Library	[48]
Others	**Irrelevant Bad terms**	[49]
	Topology Graph	[50]
	Run-time behavior	[30], [45]

TABLE 2.5: Dynamic detection feature usage overview

2.4.3 Type of identification

The detection methods can also be characterized on the principle which guides the identification.

Signature-based detection
This type of detection, also known as misuse-based detection, uses signatures to identify malware. In static detection, these signatures can be, for example, binary patterns or snippets from software code. In dynamic detection, these signatures can be a pattern of behaviour. Known malware is used to extract patterns, and to form signatures for detection. Then these known signatures are used to detect malware. This type of detection is especially useful for known malware but less effective against zero day-attacks [3]. The process of signature-based detection method is shown in Figure 2.8. This figure illustrates an example of a signature-based detection model that uses snippets from software code as signatures.

Figure 2.8 shows that a signature-based detection model has an underlying signature database. This database contains signatures of malware. In this example, the different signatures contain three snippets of malicious software code, shown as three different squares next to the signature names. As an input, this example detection model takes the complete code of a software. This complete code is, in this example, separated into different parts, resulting in 10 snippets of software. These 10 snippets are compared to the different signatures in the database. If 3 out of 10 snippets match any signature from the example database, the example detection model identifies the application as malicious. In the example figure, signature 2 matches with the input software snippets, and therefore the app is identified as being malicious. There are two important issues with signature-based detection method. One is that any malicious app can only be identified if the signature is already known and thus in the signature database. Therefore it is less effective for detecting zero-day attacks. Additionally, the detection method can easily be bypassed if the malware authors slightly change their app,

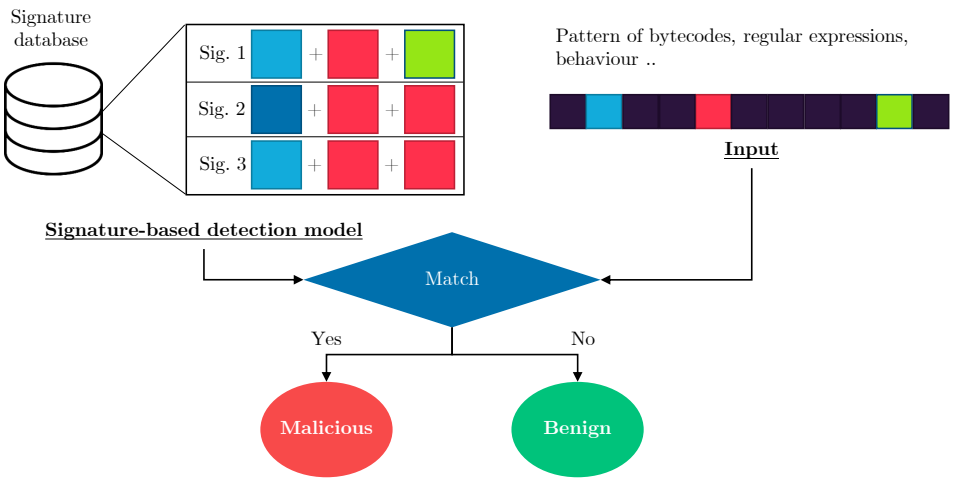


FIGURE 2.8: Signature-based detection method

in this case by changing the software code, therefore changing the signature of the app [2].

Anomaly-based detection
This type of detection is based around normal and anomalous behaviour. The former being behaviour which falls within the usual behaviour and the latter being behaviour differing from the normal behaviour. This type of detection is suitable for detecting zero day-attacks, however, they are also prone to false positives. Rare legitimate behaviour can be viewed as malicious by this type of detection. The process of anomaly-based detection method is shown in Figure 2.9

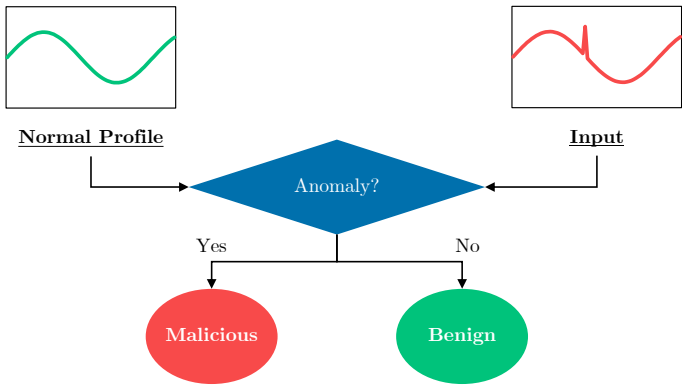


FIGURE 2.9: Anomaly-based detection method

Figure 2.9 shows that the detection method needs a profile of normal behaviour. Using this profile, the detection method checks whether any input is similar to this normal behaviour. In the figure, the normal behaviour is shown in a graph as some function over time. This graph can, for example, represent the CPU usage over time. In this case, the normal behaviour shows that CPU usage gradually declines and increases over time. The input, shown on the right in the figure, shows that the CPU usage has a spike. If this spike is higher than some given threshold, the input is flagged as an anomaly.

Specification-based detection
This is another type of anomaly-based detection method. It predefines authorized behaviours (specification), which are a certain set of rules that are allowed. Any behaviour not adhering to these rules is assumed to be malicious. One limitation is that it is nearly impossible to comprehensively and correctly create all the allowed rules [3]. The process of specification-based detection method is shown in Figure 2.10.

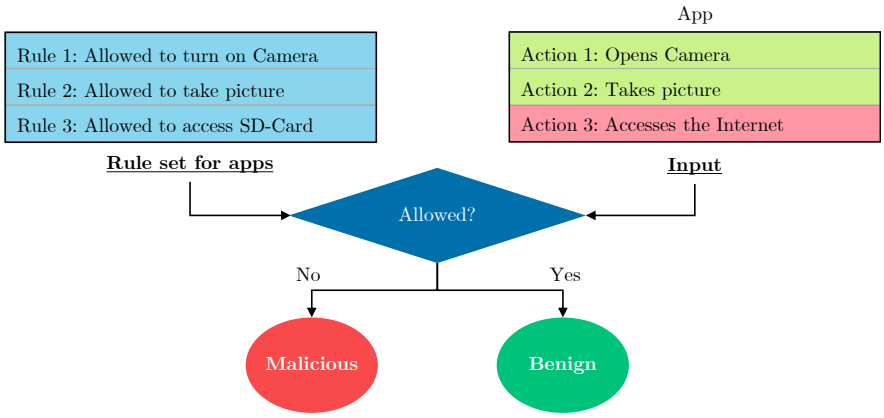


FIGURE 2.10: Specification-based detection method

In Figure 2.10, a rule set of three rules is used as an example. These three rules are actions allowed by applications. In this example, applications can turn on the camera, take a picture, and access the SD-card. This can be an example of a simple Camera app. The input comes in the form of actions. Assuming that the three rules in the rule set are the only ones defined, the input in Figure 2.10 would be flagged as malicious. This is because the first two actions in this example are allowed but the third action is not.

2.4.4 Granularity of detection

This categorization refers to the approach taken to handle the collected data during analysis. Malware detection methods can treat data from different applications separately (per app), per groups of apps, or per device. When the malware is a stand-alone application, treating the data per app results in good performance. However when malware is distributed and malicious activity is performed using multiple apps, treating the data per group of apps is more useful. Lastly, for certain types of malware such as rootkits, it could be useful to monitor the device as a whole.

2.4.5 Place of monitoring, identification and analysis

The place of monitoring, identification and analysis can differ between different malware detection methods. These activities can take place distributed, locally or in the cloud. When any of these activities are done in a distributed manner, multiple (trusted) devices are collaborating to achieve tasks within that activity. Locally refers to any activity taking place on the device itself. Lastly, the activities can take place in the cloud. Monitoring and analysing malware on phones require lightweight approaches as the resources on most devices are limited. Cloud solutions can help alleviate the aforementioned problem.
Emulators or virtual devices are used heavily by researchers for the monitoring, identification and/or analysis of malware [51]. These virtual environments are of relatively low cost and are more attractive for automated mass analysis which is commonly used with machine learning. However, using virtual environments to emulate devices can hinder effective detection of malware. Over the past years, there has been an increase in malware using methods to evade detection when being run in virtual environments [2] [52]. Some malware can detect and evade emulated environment by for example identifying missing phone identifiers and hardware. Other methods include, but are not limited to, the need for user input, measuring emulated scheduling behaviour, or running at odd times.

2.5 Related works

Related papers on dynamic malware detection using hardware features are found using a systematic literature research. The process of the systematical literature research is shown schematically in Appendix B. An overview of the papers found are shown in Table 2.6. This Table includes this paper for comparison. Section 2.5.1 describes the most important findings per paper. Developments in mobile security are examined to augment the knowledge on recent developments in mobile malware detection methods. The industry developments are described in Section2.5.2.
To the best of our knowledge, this research is the first in using data spanning a complete year of +45 real devices for the creation of mobile malware detection methods.

Article		Features		Training and testing			Performance			
Ref	**Year**	**Dynamic**	**Static**	**Benign**	**Malw.**	**Platform**	**Classifiers**	**Acc**	**TPR**	**FPR**
[53]	**2012**	Various (14)		40	4^{Cust}	2 Devices	BN, Histo, **J48**, Kmeans, LR, **NB**	0.809	0.786	0.475
[24]	**2013**	Bat, Binder, CPU, Mem, Netw	Perm	408^{PS}	$1330^{Ge,VT}$	VE + Monkey	**BN**, J48, LR, MLP, NB, RF	0.813	0.973	0.310
[26]	**2013**	Binder, CPU, Mem		408^{PS}	$1130^{Ge,\ VT}$	VE + Monkey	**RF**, BN, NB, MLP, J48, DS, LR	1.000	-	$\sqrt{MSE}$ =0.02
[54]	**2013**	CPU, Net, Mem, SMS		30^{PS}	5^{Cust}	Device	NB, **RF**, LR, SVM	-	0.990	0.001
[23]	**2014**	Bat, CPU, Mem, Netw		^{PS}	3^{Cro}	12 Devices	Gaussian Mixture + LDCBOF	≈1	≈1	≈0
[55]	**2014**	Bat, Time, Loc		-	2^{Cust}	11 Devices	Std dev	≈1	-	≈0
[56]	**2014**	Bat			Sens. Act.	Device	J48, LB, RF	-	-	-
[57]	**2016**	SC, SMS, UP	MD	PS	2800	3 Devices	1-NNeigh.	-	0.969	0.004
[38]	**2016**	Bat			7^{Cust}	1 Device	**NN**, DT	-	>0.85	-
[58]	**2016**	CPU, Mem		940^{PS}	1120^{Ge}	VE + Monkey	LR	-	0.855	0.172
[59]	**2016**	CPU, Mem, SC		1709^{PS}	1523^{Dr}	VE + Monkey	Kmeans + RF	0.670	0.610	0.280
[60]	**2016**	CPU, Mem, Net, Sto		1059^{PS}	1047^{Dr}	VE + Monkey	RF	0.995	0.820	0.007
[61]	**2017**	CPU, Mem, Net		0	$<5560^{Dr}$	VE + Monkey	C-SVM	0.820	-	-
This Paper	**2018**	CPU, Bat, Mem, Net, Sto		10^{Cust}	10^{Cust}	47 Devices	**RF**, NB, KNN, MLP, AdaBoost	0.96	0.65	0.01

Bat battery, *BN* Bayesian Network, *Cr* Crowdroid [62], *Cust* Custom, *Dr* Drebin [63], *DS* Decision Stump, *DT* Decision Tree, *Ge* Malware Genome Project [64] , *Histo* Histogram, *Kmeans* K Means Clustering, *LDCOF* Local Density Cluster Based Outlier Factor, *Loc* Location, *LR* Logistic Regression, *Mem* memory, *MD* metadata, *MLP* MultiLayerPerceptron, *NB* Naive Bayes, *Netw* network, *NN* Neural Network, *NNeighbour* Nearest Neighbour, *Perm* permissions, *PS* Play Store, *RF* Random Forest, *Std dev* Standard Deviation, *Sto* storage, *SVM* Support Vector Machine, *SC* system calls, *UP* user presence, *VE* Virtual Environment, *VS* VirusShare[65] *VT* VirusTotal[66]

TABLE 2.6: Related works

2.5.1 Academic works

A highly cited paper is by Shabtai et al. published in 2011 [53]. The authors designed a behavioural malware detection framework for Android devices called Andromaly. As features for this detection framework, they used 14 different categories of features resulting in a total of 88 collected features. The 14 different feature categories were: touch screen, keyboard, scheduler, CPU load, messaging, power, memory, applications, calls, processes, network, hardware, binder, and led. They used 40 benign applications and 4 self-developed malware applications. The 4 self-developed malware applications were a DOS Trojan, SMS Trojan, Spyware Trojan, and Spyware malware. 4 different experiments were run, differing in the device on which the model was trained and evaluated, and differing in which benign and malicious applications were included in the training set. To train their detection model, the following classifiers were used: Bayesian Network, J48, Histogram, K-means, Logistic

Regression, and Naïve Bayes. In the two experiments in which they used the same device for the training and testing of their model, the J48 decision tree classifier performed the best. In the first experiment, all benign and malicious applications were included in the training set, resulting in a TPR of 99% and an FPR of 0%. In this experiment, the training set was 80% of the total dataset and the testing set was 20% of the total dataset. The second experiment did not include all the benign and malicious applications in the training set, leading to a TPR of 91% and an FPR of 11%. In this experiment, the training set contained 3 of the 4 malicious applications and 3 of the 4 benign applications. The remaining malicious application and benign application were used for the testing set. In the two remaining experiments the device on which the model was tested, differed from the training device. For both experiments, the Naïve Bayes classifier performed the best. Including all benign and malicious applications in the training set led to a TPR of 91.3% and an FPR of 14.7%. The training set was created with the all feature vectors from one device. The testing set consisted of all the feature vectors from another device. Not including all the applications in the training set resulted in a TPR of 82.5% and an FPR of 17.8%. In this experiment, the training set consisted of the feature vectors of the 3 malicious applications and 3 benign applications from one device, and the testing test consisted of the feature vectors of the remaining malicious and benign application of another device.
Andromaly showed the potential of detecting malware based on dynamic features using machine learning, compared different classifiers, and used data collected from real devices for the training of its detection model. It also tested its robustness by experimenting with changing the training device from the testing device, and by not including all applications in the training set. The paper, however, is relatively old and much has changed regarding the malware ecosystem since 2012. Furthermore, Andromaly showed promising results but the False Positive Rates of all their four models were relatively high.

In [24], published in 2013, the authors propose a framework named STREAM, which was developed to enable rapid large-scale validation of mobile malware machine learning classifiers. Their framework used 41 features which were collected every 5 seconds from different emulators running in a so-called ATAACK cloud. The feature categories used were Binder, Battery, CPU, Memory, Network, and Permission features. The emulator used the Android Monkey application to simulate pseudo-random user behaviour such as touches on the touchscreen. To evaluate their detection model, the authors used a Random Forest, Naïve Bayes, Multilayer Perceptron, Bayes Network, Logistic Regression, and a J48 classifier. For their training set, they used 408 popular applications from the Google Play Store and 1330 malware applications from the Malware Genome Project database[64], and the VirusTotal database [66]. As the testing set, they used 24 benign applications from the Google Play Store, and 23 malware applications from the Malware Genome Project database, and the VirusTotal database. The best performing classifier was the Bayesian Network which had an accuracy of 81.26% with a TPR of 97.30% and an FPR of 31.03%.
This paper showed the potential of using dynamic features although the FPR for all their tested classifiers were relatively high. Additionally, this research ran applications separately for 90 seconds and made use of a virtual environment in the form of an emulator with user-like behaviour created by the Android Monkey tool. This lowers the confidence that the model would perform the same when evaluated on a real device with a real user.

In [26], published in 2013, an anomaly-based detection method is proposed which uses application behaviour features. This research used the dataset produced by the research of [24], mentioned in the previous paragraph. This dataset contained feature vectors from 408 popular applications from the Google Play store and 1330 different malicious applications from the Malware Genome Project and VirusTotal database. Only the Binder, CPU, and Memory features were used because, after evaluation of the dataset, the authors noticed the Battery and Network features being the same throughout the whole dataset. Another adjustment to the dataset was the balancing of the feature vectors with a technique called SMOTE. This was done because the benign feature vectors were under-sampled compared to the malicious feature vectors, due to the inclusion of only 408 benign applications compared to 1330 malware applications. The research used the Random Forest, Bayesian Network, Naive Bayes, MultiLayerPerceptron, J48, Decision Stump, and Logistic Regression classifiers.

Only the performance results are shown for the different Random Forest classifiers with different parameters. The authors used a 5-fold cross validation for the training and testing of their classifiers. The best performing classifier had 160 trees, used 8 different features, and had a tree depth of 16. This resulted in an accuracy of 99.9857% and a root MSE of 0.0183%. Only 2 False Positives were measured during this experiment.
This paper shows the potential of using dynamic features and Random Forest Classifiers. However, as this paper makes use of the dataset by Amos [24] it is sensitive to the same limitations; thus it is not known how this model would perform on a real device with a real user.

In [54], published in 2013, the authors evaluated different machine learning classifiers for their detection model. Their model used 10 features related to memory, network, CPU, and SMS for their detection model. 30 normal applications and five malware applications were used, however, the source of these applications is unmentioned. The malware applications were a Spyware, a Hostile Downloader, a Root[1], Spyware, and two Trojan Spyware applications. The benign and malicious applications were run on a real device but it is unknown how and how long the features were collected from these devices. To reduce the size of their feature set, the authors used the Information Gain algorithm. The features left over were related to the Memory, Virtual Memory, SMS, and CPU usage. The classifiers Naïve Bayesian, Logistic Regression, Random Forest, and SVM were evaluated. The training and testing were done with a ten-fold cross-validation. The best performing classifier was the Random Forest classifier with a TPR of above 98.8% for the different families of malware, and an FPR below 1%. This research shows the potential of using dynamic features but due to the lack of description of the feature collection, it is unknown how reliable the performance evaluations are. Additionally, only 5 different malware applications were tested.

In [23], published in 2014, multiple hardware features are taken for the use of anomaly-based detection of mobile malware. The features collected were CPU, memory, battery, amount of connection requests, and ICMP requests. Data from 12 smartphones was collected with the use of an application called Data Collector. These smartphones contained the most popular software in the Android market as benign application and three malware developed by [62] as malware. A Gaussian Mixture Model with a Cluster-Based Local Outlier Factor was used for their detection model. This model resulted in an FPR of almost zero and a TPR of almost 100%. This research shows the potential of using a Gaussian Mixture Model with user behavioural features for the detection of mobile malware, however, no description of the feature collection has been stated. This makes it hard to estimate the reliability of the performance evaluations. Additionally, only three different types of malware were used in this research.

In [25] the authors describe two techniques for detecting malware based on individual power consumption profiles, time, and location. This research has further been refined in [55] where they propose three power-consumption based techniques based on improved data. Both studies show that malware can be detected using power consumption based detection techniques with low False Positive Rates. Their first technique described in [55] uses location-specific power profiles of users. The reasoning behind creating such profiles was that users would be expected to use their device differently depending on their current location, and this would thus lead to different power consumption profiles. The technique was evaluated using over 10 users which ran two simulated malware. The first simulated malware was an SMS Spam malware and the second simulated malware was a Root Spyware. First, location-based power profiles were made for the users with devices not containing any simulated malware. Then by running simulated software and checking for anomalies in the location-based power profiles, the detection model would detect malware on the devices. An anomaly was reported whenever the power consumption would differ a certain amount of sigma outside of the normal power consumption. No complete results were mentioned albeit for the subset of 11 users, using one location, and a sigma of 2.5, a TPR of 100% was achieved with an FPR of 1.5%. The second technique was based on time-based power profiles. With this technique, different power profiles were made depending on the time of the day.

[1] a root malware attempts to gain system-level privileges

The reasoning behind this experiment was that users would be expected to use their device differently, and thus having a different power consumption profile, depending on the time of day. In the paper, results for two phones are described where the time profiles are based on blocks of six hours, resulting in four different time periods per day. For a sigma of 2.5%, a TPR of 43.7% and an FPR of 1.7% was achieved. The third technique combines created power profiles based on both location and time. A TPR of 82.7% and an FPR of 1% was achieved, based on results from two users, three different locations, and four different time periods The above-described research shows the potential of using location- and time-based power profiles for the detection of mobile malware. However, results are only shown for a small subset of users, making it hard to determine the actual performance of the detection models. Additionally, only two types of simulated malware were used for the detection model, making it hard to estimate how the detection model would perform on other types of mobile malware.

In [56], published in 2014, the authors examined the possibility of detecting sensitive activities with the use of power consumption measurements. Sensitive activities such as turning the screen on or off, enabling GPS/Wifi, enabling voice recorder, and more where examined. Data was collected using a physical measurement device for the battery level of a real mobile device. The classifiers evaluated were J48, LogitBoost, and Random Forest. No overall performance results are stated but they are given for the Wifi, GPS and Touch classifications. All three classifications had a True Positive Rate of more than 84% with the use of a window size, for the power measurement, of 4000 mSec. The False Positive Rate is not given but the Precision is given. For all classifications, the precision was more than 85% with a window size of 4000 mSec. The research was not focused on detecting specifically malware but the potential of using the power measurement to detect different sensitive activities was shown.

In [57], published in 2016, the authors improved their model from their earlier research [67] from 2012. In 2012 they designed an anomaly detection model named MADAM. The first version from 2012 used system calls, user presence, and SMS features. The second version added a static analysis of application packages. The newest version of the detection method consists of three modules namely the: App Risk Assessment, a Global Monitor, and a Per-App Monitor. The App Risk Assessment evaluates the app during install by analysing the metadata of the app. Whenever the app is evaluated as risky, it is added to a Suspicious list and it will be continuously monitored by the Per-App Monitor. The Global Monitor tracks the behaviour of the device during run-time by keeping track of the User Activity, Critical APIs, and System Calls. To detect malicious behaviour, two classifiers have been trained to detect malicious behaviour. The first classifier analyses the behaviour of the device on a short-term, and the second classifier analyses the behaviour of the device on a long term. Both classifiers were trained on a real device without malware, to recognise genuine behaviour. Malicious behaviour is artificially created by creating feature vectors that differ significantly from the genuine behaviour. This allows the detector to detect zero-day malware. The different classifiers trained and tested were the K-Nearest Neighbour Classifier, the Linear Discriminant Classifier, Quadratic Discriminant Classifier, Multi-Layer Perceptron, Parzen Classifier, and Radial Basis Function. The performances of these different classifiers are not mentioned but the authors state that the 1-Nearest Neighbour achieved the best classification results. The Per-App monitor is a signature-based detection method that analyses applications put on the Suspicious list. The signatures were created based on seven malicious behavioural patterns being: text messages sent by a non-default messaging app, text messages sent to numbers not in the user contact list, high number of outgoing message per period of time, high number of process per app, excessive foreground time for non-interacting and administrator app, unauthorized installation of new apps, unsolicited kernel level activity of background app. To evaluate the TPR, the detection framework has been tested against 2,800 malicious applications from the Malware Genome Project, Contagio[68], and VirusShare[65] dataset resulting in around 2800 different malware applications. These datasets contained Botnet, Installer, Trojan, Ransomware, Rootkit, SMS Trojan, and Spyware malware families. How long the different malware applications were run was not mentioned. Also, it is unknown whether the same device was used for the training and testing of the classifiers. The FPR has been evaluated using three real devices with three different level of usage intensity and

three different numbers of applications installed. The light usage device contained only 26 native apps, was mostly on standby, and only used for phone calls and text messages. The medium usage device contained 54 applications, used the device normally although no new apps were installed. The high usage device started with 52 applications, installed 91 new apps, and used his phone as often as possible. The framework achieved a TPR of 96% with their framework and a False Positive rate of less than 0.004%. Furthermore, they detected some zero-day attacks which were undetected by multiple Antivirus Software at that time. Additionally, the performance overhead of the used model is low. As the detection model used system calls as a feature, the detection model needed root permissions.
MADAM showed a high TPR and low FPR with low-performance overhead for their framework. The performance of their model was assessed on real devices differing the usage intensity over a period of one week, indicating that the performance results reflect real-life circumstances. A factor limiting the applicability of the framework is its requirement for root permissions as System Calls were used as features.

In [38], published in 2016, the authors used artificial intelligence tools to detect malware based on covert channels. This type of malware uses advanced mechanisms to bypass security frameworks and to extract information. A covert channel is created as a communication channel in order for two colluding applications to extract personal information. An example would be that one application called CCSender has access to sensitive data but does not have permissions to send this over a network. Using the covert channel, this application can then send information to another app, named CCReceiver in this example, who does have access to the network. This covert channel is used such that the communication is hidden. The covert channel can take different forms. For example, the CCSender can change the volume settings and the CCReceiver keeps track of any changes in the volume settings. With eight levels of volumes, this results in three bits per iteration of information. Seven self-made android Covert Channel malware applications were used in the research of [38]. Every malware used a different type of covert channel. The channels used in the research were: 1) type of intent, 2) file lock, 3) system load, 4) volume settings, 5) Unix socket discovery, 6) file size, 7) memory load. The detection was based on two levels of energy consumption measurements. First, a high-level energy consumption measurement was used with a modified version of PowerTutor. This application measured the energy consumption per process running on the system. Second, a middle-level energy consumption was used that measured energy consumption based on the current and voltage values stored in the /sys folder. Based on past values of consumptions in a clean system, without malware, a regression model was created to predict future behaviour. Any power consumption deviating disproportionately from this expected behaviour would be classified as anomalous. Additionally, a classification based model was used, using a set of energy-related features. This classification based model used data from the feature set during the presence and absence of colluding applications. To test their detection model the authors made use of Neural Networks and Decision Trees. The detection models were run on two different smartphones, both being in an idle state. The best performing classifier was the classification-based Neural Network. It was able to detect all seven colluding apps with an accuracy of more than 85%. No false positive information was stated.
The paper showed the potential of using battery features for the detection of malware using covert channels. The accuracy, however, was relatively low compared to other malware detection systems. Also, no FPR information was stated making it hard to estimate its real potential. Furthermore, the data was only collected from phones being in an idle-state and with applications running separately, making it hard to estimate the performance results of phones being in use by real users.

In [58], published in 2016, the authors used the memory usage and CPU usage as features for the dynamic detection of mobile malware. Their model was based on 1220 malware applications from the Malware Genome Project dataset. The 952 benign application were popular applications download from the Google Play store. Memory and CPU usage were tracked by running every application separately for 10 minutes in an Android emulator. The emulator was fed with user-like input by the Monkey application. Their initial feature set consisted of 57 features and the optimized feature set only contained 7 features. The

classification algorithm used was linear Logistic Regression with the use of a sliding window technique. For the training set, they used 571 benign applications from the Google Play store, and 727 malware applications from the Genome project. The test set contained 275 benign applications and 304 malware applications. Lastly, they used a validation set of 94 benign applications and 89 previously unseen malware applications. Previously unseen in this case means that the malware applications were neither in the training set nor the testing set. For the creation of the detection model, the window length, the threshold of the number of malware records contained in each window, and the number of checks differed. Their findings show that CPU and memory usage features in combination with Logistic regression and a simple sliding window technique can be used for the detection of malware. However, their best malware detection model, having a TPR of 95.7% of the malware, showed a relatively high false positive rate of around 25%. The detection model having the highest F-measure achieved a TPR 85.5% and an FPR of 17.2%.
This paper showed the potential of using the memory usage and CPU usage as features for the dynamic detection of mobile malware. However, the detection model showed relatively high false positive rates. Furthermore, by using an emulator with the Monkey application it is hard to estimate whether the performance results would be the same on real devices with real user input.

In [59], published in 2016, the researchers used features related to System Calls, Memory usage, and CPU usage. They traced three values for the CPU feature: i) CPU total, ii) CPU user, and iii) CPU kernel. As memory features, they monitored three types of memory consumption: i) memory consumption by the Dalvik Virtual Machine, ii) native memory usage, and iii) total memory usage. Per type of memory consumption, five features were extracted: i) total proportional set size, ii) the shared RAM, iii) the private RAM, iv) the Heap allocation, and iv) the Heap free. Benign apps were downloaded from the Google Play store and malicious applications were collected from the Drebin dataset [63]. The applications were run for 10 minutes in an Android emulator and features were collected every two seconds. The emulator was fed with user-input from the Monkey application. As detection models, they first used a K-means clustering algorithm to cluster apps based on similarity of memory and CPU usage. Then they used a Random Forest Classifier on every cluster that classified the applications based on their System Calls. For the training, they used 1000 benign applications and 1000 malware applications. The best performing model was the classifier with the 7-means clustering algorithm and a Random Forest classifier of 50 trees. It had an accuracy of 67% and an FPR of 28%.
This paper showed the potential of using System Calls, Memory, and CPU features for the detection of malware. However, due to the use of an emulator with the Monkey application, it is unknown whether the performance results would be the same for real devices with real user input. Also, the performance is relatively low compared to other detection methods. Furthermore, this detection method needs root permission as it uses System Calls as features.

In [60], published in 2016, the authors used features related to CPU, Memory, Storage and Network for their detection model. Benign applications were downloaded from the Google Play store and malicious applications were downloaded from the Drebin dataset. The applications were run for 60 seconds in an Android emulator with the use of the Monkey application. They used Random Forest classifier with different types of parameters. They used the features raw and with a transformation called Discrete Cosine Transformation. Furthermore, the authors changed the granularity of the detection method by either taking all the features, only the global features, or only the features of the application under analysis. The classifier using the global features was the best performing classifier. It was trained with a ten-fold cross-validation. This classifier had an accuracy of 99.52% and an FPR of 0.74%.
This research paper showed the potential of using CPU, Memory, Storage, and Network features for a detection model. The malicious applications and benign applications were all evaluated separately in an emulator with the Monkey Application for a brief time, making it hard to estimate whether the performance results would be the same on a real device used by a real user.

In [61] the authors used Support Vector Machines to create a detection model based on resource metrics of mobile phone devices. These resource metrics (features) were CPU, Memory and Network usage. The features were monitored system-wide and application-specific. They used the Drebin dataset for the malicious applications. It is unknown how many malware applications were included in their research as they took a subset of the Drebin dataset, not mentioning the amount. However, as the Drebin dataset consists of 5560 malware applications, the amount of malware applications used in this research is less than 5560. An emulator was used for the training on of their model. This emulator was fed with simulated user input events from the application Monkey. The classifier used was a C-SVM with a radial basis function kernel. Their detection model achieved an accuracy of 82 per cent. The False Positive rates are unknown but the precision of the model range from 10% to 90% depending on the malware family.
This paper shows the potential of the use of hardware features such as CPU, memory, and network usage. The model performed relatively bad, based on their high FPR, in comparison with the other detection methods. Furthermore, this paper used an emulator with the application Monkey, making it susceptible to the earlier mentioned limitations of the usage of an emulator.

In summary, it has been shown that using hardware features might be an effective way of detecting malware on mobile devices. The Naive Bayes, K-Nearest Neighbour, and Random Forest classifiers seem the most promising. Most recent papers have based their detection model on emulators and user-like input from the application Monkey making it hard to estimate their performances on real devices with real users. Additionally, most papers have only run the benign and malicious applications for a few minutes meaning that any malware showing malicious behaviour after a few minutes, would not be detected. Furthermore, a limited amount of research has shown high-performance results.

2.5.2 Industry developments

The recent industry developments on mobile malware detection are briefly described in this section to get a complete view of the state of the art of mobile malware detection methods. The focus is on the developments by Google, as this is the developer of the Android software. The statements below are based on the most recent security report of Google [6].

Dynamic Analysis
Google uses different techniques to check whether an application might perform malicious behaviour before this application is allowed to be published in the Google Play store. One technique includes performing a dynamic analysis on the applications. Only from 2016, Google started adding automated event injections to emulate for example user-input. This is what the Monkey application does as well, as mentioned in different papers described in Section 2.5.1. This improvement led to a three-fold increase in the number of apps flagged as potentially malicious. Google runs the applications in virtual environments and attempts to detect anti-cloaking techniques. These anti-cloaking techniques are ways to hinder analysis in virtual environments.

SafetyNet Integration
Google uses SafetyNet to identify apps and other threats throughout the Android ecosystem. SafetyNet uses security information from different devices such as security events, logs, and configurations. In 2014, Google focused on the detection of applications trying to abuse SMS. In 2016 it integrated its data with the Anomaly Correction Engine to focus on rooting applications. Later in 2016, the combination of the ACE and SafetyNet has been used to identify applications that make devices stop working.

Anomaly Correction Engine
Since late 2015, Google started monitoring changes in key device security indicators. Google combines the information across a large number of devices to predict which applications cause security changes on a device. The precision of the ACE detecting rooting apps in May

2016 was over 90% according to Google.

Additional Machine Learning researches
In 2016, Google analysed applications based on their installation patterns. Using a semi-supervised machine learning approach, Google tried to automatically group apps based on install patterns. Google described that it used document analysis and clustering techniques to group PHAs. It is unknown which documents were analysed. Using a clustering algorithm it was able to detect new variants of existing PHA families, detect inconsistencies in previous reviews, and suggest classifications for previously unseen apps based on which cluster an application was the closest to.

Other Antivirus solutions
Altough Google describes good performance results, a recent comparison of different Antivirus solutions shows a low detection rate of malware by Google [69]. Figure 2.11 shows the accuracy of different popular Antivirus solutions. The False Positive Rates of all antivirus solutions is 0%. This figure shows that Google Play Protect has a low accuracy in comparison with the most popular Antivirus solutions. Unfortunately, the inner workings of these different Antivirus applications are unknown. Furthermore, it is unknown which specific malware applications were used to evaluate the performances of the different Antivirus solutions in Figure 2.11. Research has shown that Antivirus solutions can easily be bypassed by different obfuscation methods [2]. However, this conclusion is based on relatively old research papers from 2012 to 2014. Whether Antivirus solutions are still easily bypassed is unknown to us at the moment of writing.

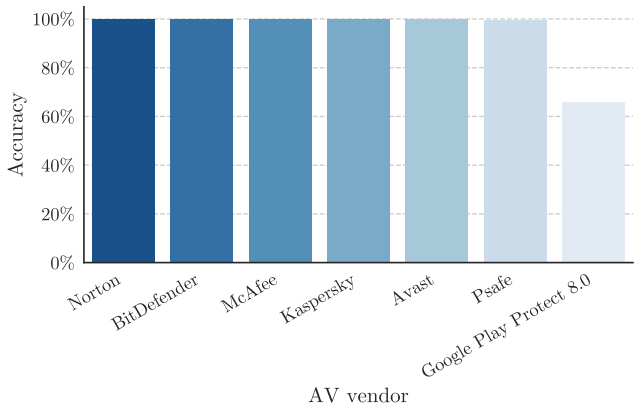


FIGURE 2.11: Performance of different AV vendor according to [69]

Chapter 3

Data Understanding

This chapter describes the dataset used throughout this research. Figure 3.1 shows the activities and their planned output for the data understanding phase. These activities are described in the following different sections. Section 3.1 describes how the dataset was created and how it was collected. Section 3.2 describes the basic characteristics of the dataset such as the data types and the number of data points. Section 3.3 describes the data exploration phase with tables, charts, and other visualisation tools to better understand the content of the dataset. Section 4.2 describes the data cleansing process.

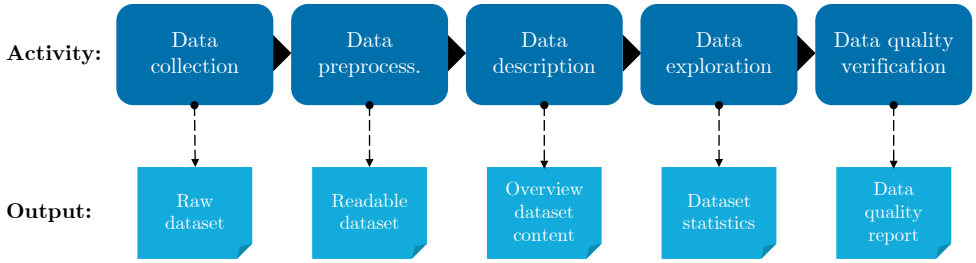


FIGURE 3.1: Data understanding activities and output

3.1 Data collection

The dataset [5] used throughout this research is provided by the Ben-Gurion University. This dataset was created by providing 50 volunteers with a Samsung Galaxy S5, which contained software to track device metrics, e.g. battery usage, CPU usage, memory usage. The device also contained a self-written malware that simulated different malware types every month and that logs of its actions taken. The dataset thus consists of device metrics data and malware data. A record of the device metrics dataset includes a specific device's values for its device metrics, the associated sampling timestamp, and an associated userid. A record of the malware dataset includes details of the actions taken by the malware application, the associated sampling timestamp and an associated userid. The volunteers were instructed to use the smartphone as their sole device for at least two years. and they consented to both the device tracking and the installation of the malware application. Additionally, they were instructed to use the malware application at least once every few days for a few minutes. The malware application (a Mobile Trojan) appeared benign although behaved maliciously, hence could be used as a normal app. Section 3.2.1 describes the malware application and its behaviour in more detail.

A timeline of the dataset creation is shown in Figure 3.2. This Figure shows that 11 malware versions were run in total. Version 10 was not the installation of malware albeit a simulation of device theft. An important event is the tracking of additional device metrics after malware version 5, which is also shown in Figure 3.2. As a result of the additional tracking, the dataset contains more data on the malware versions after version 5 than before version 5; more on this in Section 3.2.2.

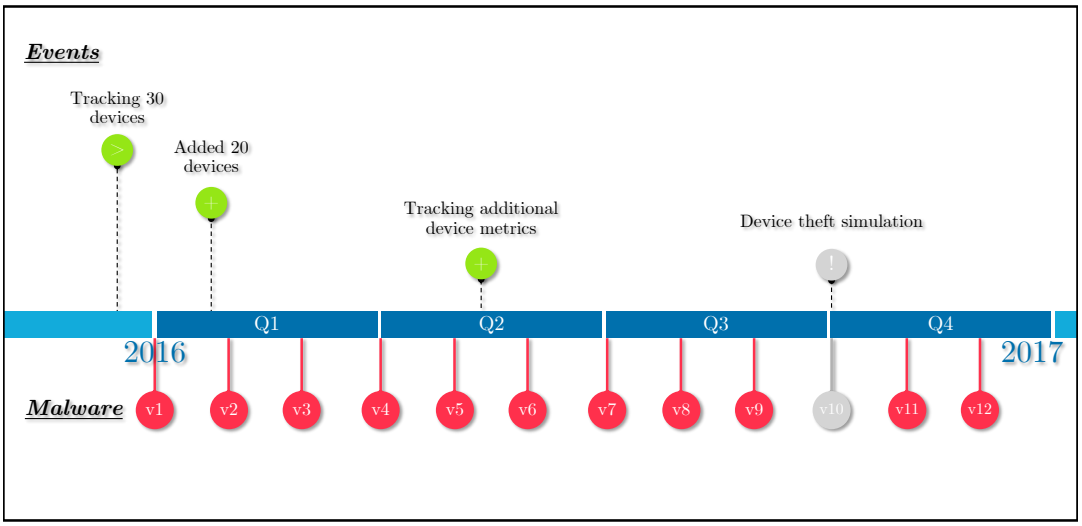


FIGURE 3.2: Timeline of dataset creation

3.2 Data description

The dataset is divided into 13 different probes. A probe is a grouping of multiple sensors that shared the same sample interval. For example system metrics, being part of the T4 probe were recorded every 5 seconds and are therefore joined in one probe.
Of the 13 probes, 6 probes are pull probes which recorded sensor data every fixed interval. Of the 13 probes, 7 probes are push probes which recorded sensor data as soon as new information arrived. For example, the SMS probe recorded a new SMS message immediately after the arrival of an SMS message.
The content, number of fields, and a description of the pull probes and push probes are shown respectively in Table 3.3 and Table 3.4.

Probe	Sample interval	Content	Nr. fields	Description
T0	24 hrs	Telephone info	15	Information on the current telephony configuration.
		Hardware info	6	The device's hardware configuration.
		System info	5	Kernel, SDK, baseband, and general information.
T1	60 sec.	Location	15	Location(anonymized via clustering), speed, and accuracy.
		Cell tower	5	Cell tower ID, type, and reception info.
		Device status	14	Brightness, volume levels, orientation, and modes.
		Wifi scan	4	Identifiers, encryption, frequency, and signal strength.
		Bluetooth scan	9	Identifiers, device class (type), parameters, and signal strength.
T2	15 sec.	Accelerometer	51	Stats on 800 samples captured 4 seconds at 200Hz.
		Linear accelerometer	51	
		Gryoscope	51	For each respective axis: mean, median, variance.
		Orientation	9	covariance between axis, middle sample.
		Rotation vector	12	FFT components and their statistics.
		Magnetic field	51	A subset of these features is extracted from the orientation, rotation, and barometer sensors.
		Barometer	16	
T3	10 sec.	Audio	21	Statistics over 5 seconds.
		Light	3	Luminosity
T4 (System)	5 sec.	Global app stats	98	Info on CPUs, memory, network traffic, IO interrupts, and WiFi AP
		Battery	14	Configuration and statistics on power consumption and temperature.
Apps	5 sec.	Local app stats	70	For each running application: stats on CPU, memory and network traffic, Linux level process information from the system /proc folder.

Adjusted from Table 9 in [5]

FIGURE 3.3: Overview pull probes

Probe	Content	Nr. fields	Description
App packages	App install, update, removal	11	Log off when applications are installed, updated, or removed: provides the app's version, hash of the APK, and list of permissions.
Broadcast	Broadcast intents	3	All Android broadcast intents (events): changes in password, Bluetooth, network, RSSI, app packages, wallpaper, volume.
Call log	Calls	5	Address, when, duration, outgoing or ingoing, and an indication if number is from user's contacts.
Moriarty (Malware)	Malware actions Malware sessions	6	All clues left by the Moriarty malware agent.
SMS log	SMSes	5	Address, when, outgoing or ingoing, and an indication if number is from user's contacts and if the content contains a URL.
Screen status	Screen on/off	2	Log off when the screen turns on or off
User presence	User present	1	Android USER_PRESENT intent log: a record of when the user begins interacting with the device.

Adjusted from Table 8 in [5]

FIGURE 3.4: Overview push probes

As described in Section 2.4.2, this research uses hardware features for the creation of mobile malware detection methods. Therefore, we selected the following probes: T4 probe, Apps probe, and Moriarty probe. These probes are highlighted in green in Tables tab:pullprobes and 3.4. For easier and faster reference, they are respectively referred to as: System probe, Apps probe, and Malware probe.

The sizes of the selected probes are shown in Table 3.5. Note here that the total size of the Apps probe is more than 1 TB, but the selected size is 9.9 GB. This reduction in size is because the Apps probe is filtered to contain only the data of the malware application. For this research, this is the only relevant data of the Apps probe. The three selected probes are described in more detail below.

	Malware	System	Apps
Total			
Size(GB)	0.11	100.00	>1,000.00
Nr. columns	9	130	58
Nr. rows	1,091,644	156,644,389	4,635,675,997
Selected			
Size(GB)	0.11	100.00	9.9
Nr. columns	9	130	58
Nr. rows	1,091,644	156,644,389	34,985,575

FIGURE 3.5: Relevant probe sizes

3.2.1 Malware probe

The malware probe tracked data of the self-written malicious application installed on the devices of the volunteers. Each malware probe's record is a log of the action taken by the user's malware application. The log contains details about the action and session. Additionally, the log contains other features, such as the malware version and timestamp.

Feature category	Nr. feats
Action	2
Session	3
Other	3
Total	**8**

TABLE 3.1: Malware probe features

Each malware version resembled a subtype of Mobile Trojan, therefore the malware versions showed benign behaviour and malicious behaviour. In total, 11 different malware versions were included. These malware versions are shown in Table 3.2. This table shows the benign behaviour, malicious behaviour (as described by [5]), the malware type (with the taxonomy of Section 2.1.1), a description of its actions, and the real malware on which the malware was based. Important to note is that each time data was transmitted by the malware application, the data was scrambled prior to sending to protect the privacy of the users.

The malware application started in either a benign or malicious session. Within a benign session, solely benign actions were performed. In a malicious session, the malware performed

Ver.	Benign behaviour	Malicious behaviour	Malware type	Description	Malware sample
1	Game	Contact theft	Spyware	Steals, encrypts, and transmits all contact stored on device.	SaveMe/ SocialPath
2	Web browser	General spyware	Spyware	i) Spies on location and audio, or ii) spies on web traffic and web history.	Code4hk/ xRAT
3	Utilization widget	Photo theft	Spyware	Steals photos that are taken and in storage, and takes candid photos of the user.	Photsy/ Phopsy
4	Sports app	SMS bank thief	Spyware	Captures and reports immediately on SMSs that contain codes and various keywords. Volunteers periodically enter one of our websites, enter their phonenr. and then enter a code which we respond to them via SMS	Spy. Agent.SI
5	Game	Phishing	Phishing	Makes fake shortcuts and notifications to login to Facebook, Gmail, and Skype.	Xbot
6	Game	Adware	Adware	Gathers information and places ads, popups and banners.	
7	Game	Madware	Spyware, Adware, Hostile downloader	Gathers private information and places shortcuts, notifications, and attempts to install new applications.	
8	Lock-screen	Ransomware	Ransomware	Performs either: 1) lock screen ransomware, or 2) crypto ransomware.	Simplocker.A/ SLocker
9	File Manager	Click-jacking	Privilege escalation	Tricks the user to activate accessibility services to then hijack the user interface.	Shedun (GhostPush)
10		*Device theft*			
11	Music Player	Botnet	DOS	Either performs: 1) DDoS attacks on command, or 2) SMS botnet activities	Tascudap.A/ Nitmo.A...
12	Web media. player	Recon. Infiltr.	Other	Maps the connected local network and searches for files and vulnerabilities.	

TABLE 3.2: Malware versions in the malware probe

both benign and malicious actions. The choice of the malware app to start in a benign or malicious session differed per malware type. Three different modes were identified during data exploration. Version 1 and 11, alternated every session between benign and malicious (mode i). Version 2, 6, and 7 changed to benign after two malicious sessions (mode ii). Lastly, Version 4, 5, and 8, were continuous in a malicious session (mode iii). After the malware application was started for the first time, the session type was stochastically determined. In the case of mode i, the probability of starting in a benign session was 1/2. In mode ii, the probability of starting in a benign session was 1/3. In mode iii, the probability of starting in a benign session was 0. A state diagram illustrating the aforementioned state changes is shown in Figure 3.6. Lastly, Versions 3 and 12 changed every day, being the fourth identified mode. This mode is not included in Figure 3.6 as it behaves the same way as mode i, except that the malware app checks for the mode of the last day, instead of the mode of the last run.

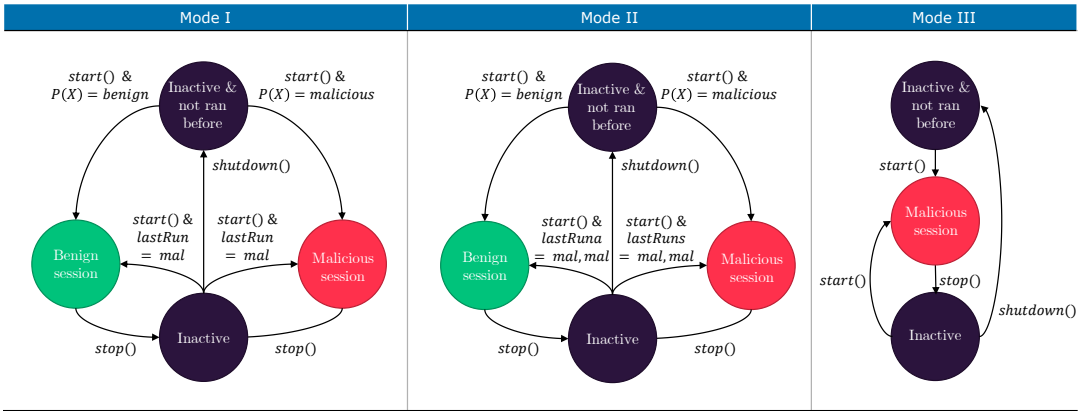


FIGURE 3.6: States of malware application in dataset

3.2.2 System probe

The system probe (T4 probe), tracked global device data every five seconds. The feature categories that were tracked are battery, CPU, network, memory, I/O interrupts, and storage. The number of features per feature category is shown in Table 3.3. The other feature category contains the following features: probe version, timestamp, and user id features. Each record of the system probe is a log of the user's global device data at a given time. The device data is taken from the /proc/ folder. Important to note here is that only 41 of 130 features, were tracked from the start of month 1. These features are called the Basic features. After month 5, 89 additional features (advanced Linux features) were tracked. These features are called the Advanced features. The advanced features are thus not available for malware version 1 until 5, as these were run before month 5. For a complete description all features of the System probe, see Appendix E.

Feature category	Nr. feats
Battery	15
CPU	18
I/O interrupts	27
Memory	38
Network	13
Storage	14
Wifi	2
Other	3
Total	**130**

TABLE 3.3: System probe features

3.2.3 Apps probe

The App probe recorded app data at every fixed time interval for each application installed on the device. As stated before, for this research, the only relevant data is the app data of the malware application. Each record of the apps probe is a log of the user's (malware) app data at a given time. The app data is taken from the /proc/ folder.
Important to note here is that, just as the Systems probe, the Apps probe contains Basic features and Advanced features. After month 5, 13 additional Linux fields were tracked. These features are called Advanced features from hereon and the other 35 are called Basic features. For a complete description of all features of the Apps probe, see Appendix F.

Feature category	Nr. feats
App cpu	13
App info	4
App memory	17
App network	4
App process	16
Other	4
Total	**58**

TABLE 3.4: Apps probe features

3.3 Data exploration

To verify the data quality of the probes and to understand the content of the datasets, the probe datasets are explored individually before the data integration phase. This phase is called the Data exploration I phase. An important finding during the exploration of the Malware dataset is the distribution of data among the different malware types. The malicious actions are overrepresented in the Malware dataset. Roughly 90 per cent of the malware data points are malicious actions and 10 per cent are benign data points. To have a representative dataset, i.e. a dataset reflecting real-life circumstances, the dataset is balanced to include fewer malicious data points and more benign data points. This process is described in Section 4.4.
The next Sections focus on the Data exploration II phase, which was performed after the data integration phase. This exploration is on the final dataset used for the training and testing of the detection models. For easier reference and to keep the structure logical, the Data exploration II phase, that took place after data integration, is reported in this Section.

3.3.1 Data distribution

An important finding of the second data exploration phase, is the difference in distribution of data per user as can be seen in Figure C.1. Some users are overrepresented in the dataset. This might suggest that some models are biased towards overrepresented users in the dataset. To overcome this issue, multiple testing modes are used, as described in Section 5.2.4

Another important finding is the difference in distribution of data over time. Figure 3.8 shows more data in months 1, 2, 6, and 7. This indicates that more data is available for malware versions 1, 2, 6, and 7 in comparison with the other malware versions.

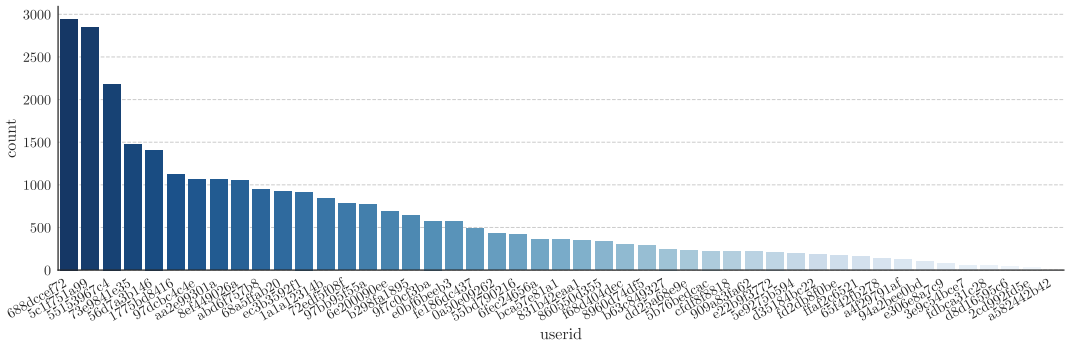


FIGURE 3.7: Data distribution of users

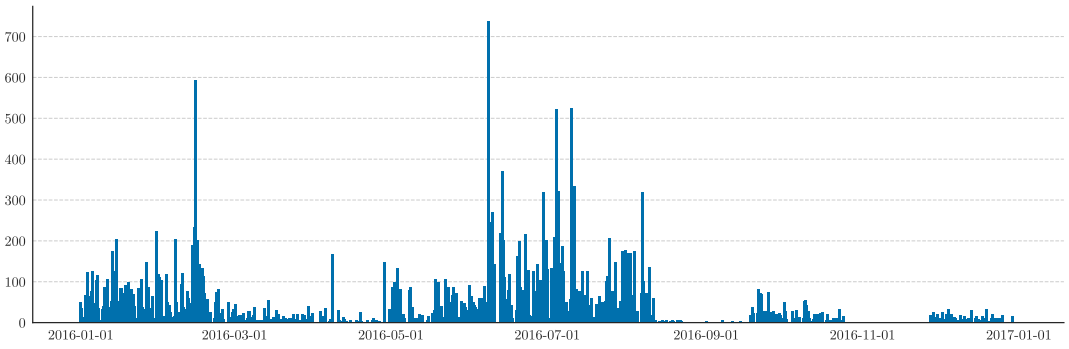


FIGURE 3.8: Data distribution over time

Lastly, the Figure 3.9 shows the distribution of malware versions per action type. As expected from the distribution of data over time (Figure C.2), more data is avilable for malware versions 1, 2, 6, 7 than the rest of the malware versions. To assess the performance of classifiers on seperate malware types, multiple training modes are used and described in Section 5.2.3.

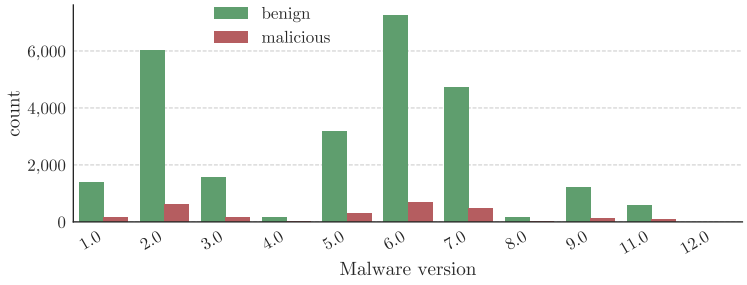


FIGURE 3.9: Data distribution of malware version per action type

3.3.2 Correlations in dataset

Figure 3.10 shows the top 10 highest (Pearson) correlation coefficients between action types and features (left) and malware session types and features (right). The postfix *_mor_app* is used for features from the Apps probe; the other features are from the System probe. For a description of the features please refer to Appendices F and E. The correlations were calculated by taking the binary value of the action type or session type string. The *malicious* string was set to a 1 and the *benign* string was set to a 0. Figure 3.11 shows the top 10 lowest (Pearson) correlation coefficients between malware actions and features (left) and malware sessions and features (right) The shown features might indicate the relative importance of these features for the detection of malicious actions and sessions.

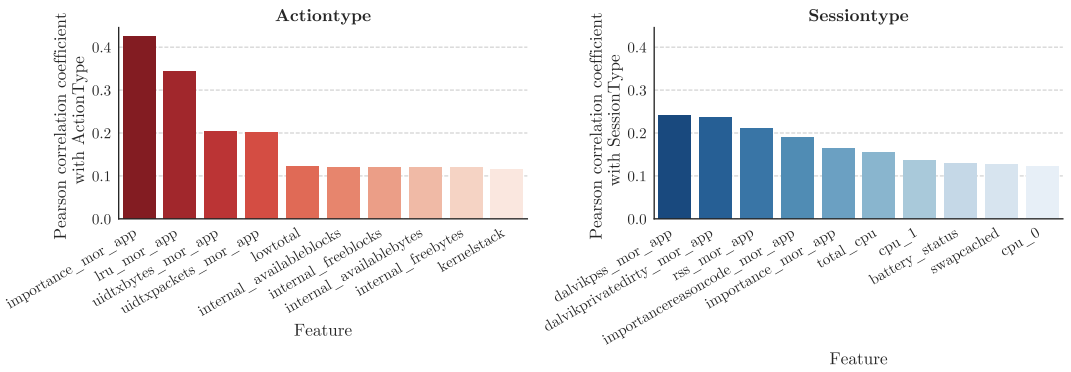


FIGURE 3.10: Top 10 highest correlation coefficients of action type and features (left) and session type and features (right)

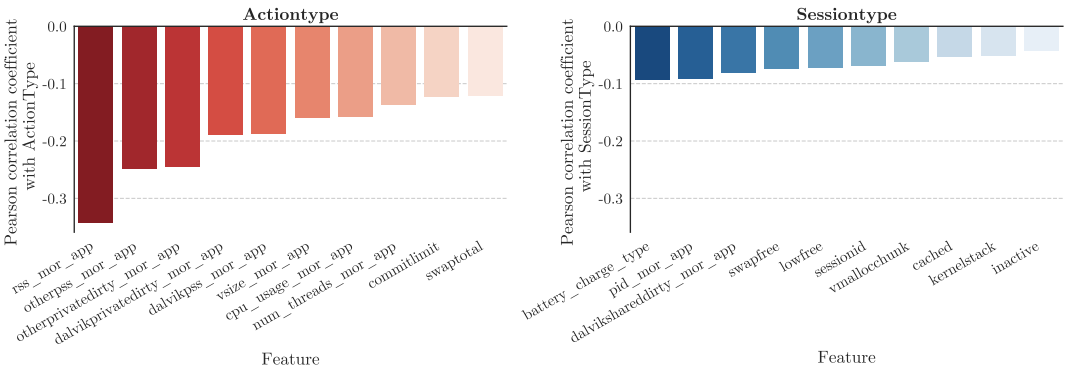


FIGURE 3.11: Top 10 lowest correlations of action type and features (left) and session type and features (right)

Chapter 4

Data Preparation

This chapter describes how the data was prepared for the modelling phase. Figure 4.1 shows the activities and their output, for the data preparation phase. Section 4.1 describes which data rows and columns were selected for this research. Section 4.2 describes the data cleansing. Section 4.3 describes how multiple data sources were integrated to create a dataset suitable for modelling. Section 4.4 describes the balancing of the dataset. Lastly, Section 4.5 describes how the data was formatted to allow the training and testing of different machine learning classifiers.

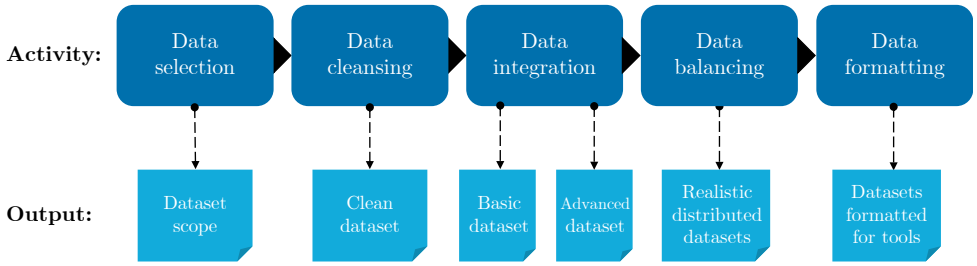


FIGURE 4.1: Data preparation activities and output

Note here that the data integration phase leads to two different datasets. One dataset, the Basic dataset, contains data from the entire year of 2016. The other dataset, the Advanced dataset, only contains data from 2016 Q3 and Q4. As described in Sections 3.1, additional columns were added to the System and Apps probe from the start of Q3. Therefore, data for these columns are only available from the start of Q3. The data integration phase is described in more detail in Section 4.3.

4.1 Data selection

The data selection is based on the relevance of items (i.e. rows) and attributes (i.e. columns) for the training and testing of different machine learning classifiers.

Relevant items
All rows from the year 2016 are selected because rows of this year contain both device data and malware data. From the malware probe, all rows containing data of malware version 11 are dropped. This version contains data of simulated device theft and is therefore not a malicious program.

Relevant attributes
Columns that are empty are dropped. The only column containing no data is *connected_wifi_ssid* column. No other columns are dropped during this phase to enable the data integration process. Some columns that should not be included in the experiments are needed for the data integration process. For example: the *userid* or *timestamp* of a record need not be included in the experiments. However, these columns are needed to join the probe datasets. Therefore the columns are selected during the experiment designs, as described in Section 5.2.5

4.2 Data cleansing

Data imperfections are resolved during the data cleansing process. This implies resolving i) missing data ii) data errors iii) measurement errors iv) coding inconsistencies v) and bad metadata.

4.2.1 Resolving missing data

Every column of the different probes is explored by retrieving basic statistical summaries, as described in Section 3.3. The amount of missing data per column is identified using the count of the columns included in these statistical summaries. Rows containing any missing data are dropped. No method is chosen to replace the missing data to minimize the adjustments to the original datasets.

4.2.2 Resolving data errors

Data errors refer to data that was entered incorrectly, usually due to typographical errors made in entering data.
Data errors are identified in two ways: i) all column statistics are reviewed to check for anomalies ii) column rules are set to check for errors in data. Based on the column statistics, no anomalies were found in the datasets that indicated any data errors.
Column rules are set per numerical columns to check whether a value is within an expected range. For example, the value for the battery level column is expected to be within a 0 to 100 range. The numerical columns and their rules are shown in Table 4.1. Additionally, possible values for string columns were identified. The possible values, the column, and the probe belonging to the column are shown in Table 4.2.

Rule	System columns	Apps columns
$i \in [0, 1]$	battery_invalid_charger	
$i \in [0, 2]$	battery_chargetype	
$i \in [0, 10]$	battery_health	
$i \in [-19, 20]$	-	nice
$i \in [0, 100]$	battery_level, battery_scale, cpu_0, cpu_1, cpu_2, cpu_3, total_cpu	rt_priority, cpu_usage
$i \in [-1800, 1800]$	battery_current_avg	
$i \in [0, \infty]$	*Remaining numerical columns*	*Remaining numerical columns*

TABLE 4.1: Rules for numerical columns of datasets

Possible values	Malware columns	System columns	Apps columns
"Benign", "Malicious"	actionType, sessionType	-	-
$endsWith$("GHz"), $endsWith$("MHz")	-	cpuhertz	-
"D", "R", "S", "T", "X", "Z"	-	-	state

TABLE 4.2: Rules for string columns of datasets

4.2.3 Resolving measurement errors

Measurement errors refer to data that is entered correctly but is based on an incorrect measurement scheme.
Measurement errors are identified by reviewing column statistics to check for anomalies.
No measurement errors were found for none of three probe datasets.

4.2.4 Resolving coding inconsistencies

Coding inconsistencies refer to inconsistencies between the meaning of a field and the meaning stated in a field description.
Coding inconsistencies are identified by reviewing column statistics to check for anomalies.

No coding inconsistencies were found for none of three probe datasets.

4.2.5 Resolving bad metadata

Bad metadata refers to a mismatch between a value and the documentation.
Bad metadata was found for the System probe. The documentation described one data schema, i.e. column structure, for the System probe. However, while reading the dataframes, two different data schemas were found. This issue was resolved by reading the dataset two times. Once with the first data schema, and once with the other data schema. During the first reading, the rows not conforming to the first data schema were dropped. During the second reading, the rows not conforming to the second data schema were dropped. Then, the resulting dataframes were joined to create the new System dataset. A detailed description of the solution is described in Appendix A.

4.3 Data integration

Three different probes are used throughout this research. The integration of these probes leading to the Basic dataset and Advanced dataset are shown in Figure 4.3. The Basic dataset is a dataset containing data with only Basic features. As described in Section 3.2, data from Q1 until and including Q4 of 2016 is available for these features. The Advanced dataset is a dataset containing data with both the Basic and Advanced features. As described in Section 3.2, data from Q3 until and including Q4 is available for these features.

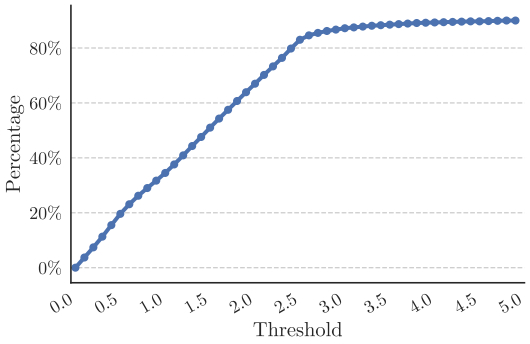


FIGURE 4.2: Trade-off data inclusion and threshold

Figure 4.3 shows that the System and Malware probe are cleaned before joining them on the *userid* and *timestamp*. For the Systems probe, the relevant features, i.e. columns, are selected before cleaning. For the Basic dataset, the relevant features are the Basic features and for the Advanced dataset, the relevant features are the Basic and Advanced features. The selection of relevant features is needed for the creation of the Basic dataset because the data cleansing process would otherwise remove all rows from Q1 and Q2. These rows contain null values for the Advanced features, which the data cleansing process removes. The selection of the relevant features is performed for the Advanced dataset for consistency.

After the cleaning of the Malware and System probes, the two probes are joined on the *userid* and *timestamp*. The *userid* must be an exact match. The *timestamp*, however, does not need to be an exact match because few exact matches exist. The Malware probe executed specific actions at random time intervals, whereas the System and Apps probe recorded data every 5 seconds. To increase the amount of data after joining, the timestamp of the Systems and Apps probe need to be within a certain positive threshold of the Malware timestamp. A threshold of five seconds is chosen as 82 per cent of the data is matched with this threshold. The trade-off between data inclusion and threshold is shown in Figure 4.2.

After joining the Malware and System probe, the resulting dataframe is joined with a cleaned Apps probe on the *userid* and *timestamp*, using the same 5-second threshold. Before joining, a selection of relevant features is performed for the Apps probe for similar reasons as the System probe. The data integration process leads to two datasets, i.e. the Basic dataset and Advanced dataset. Section 5.2.2 describes and compares both datasets.

After integration, the file size of the datasets are no longer excessive and can, therefore, be stored in memory. Therefore, the PySpark dataframes are transformed into Pandas dataframes. Nevertheless, the data is stored and used on the Hadoop cluster as an optimal use is made of distributing the training and testing of machine learning classifiers. This is

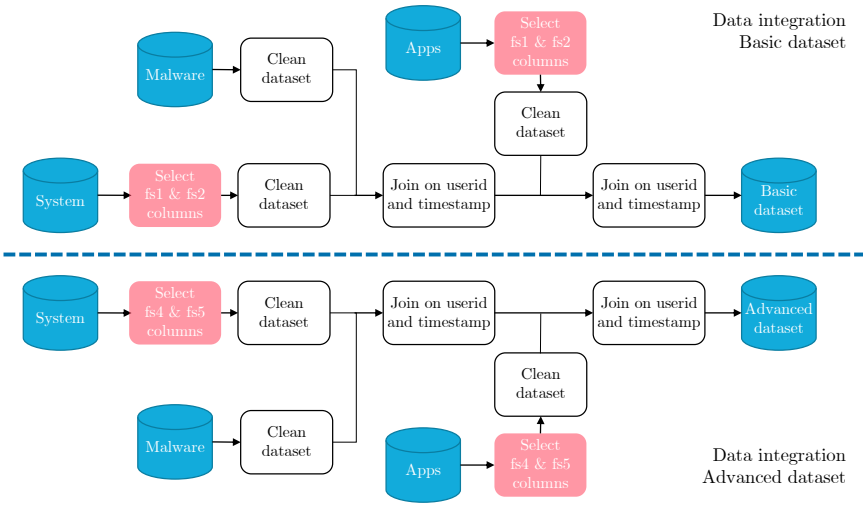


FIGURE 4.3: Data integration overview

described in Chapter 5.

4.4 Data balancing

As described in Section 3.3, the Malware dataset is overrepresented by malicious datapoints with around 90 per cent of the rows describing malicious actions. In real life, however, malicious applications and therefore the malicious actions performed, are relatively low compared to benign applications. Therefore, to better reflect real-life circumstances, the dataset is balanced to include fewer malicious actions and more benign datapoints. Two ways to balance the data are: i) increasing benign datapoints ii) decreasing malicious datapoints. The latter requires no addition of (artificial) data and is therefore chosen. The dataset is balanced by downsizing the number of malicious datapoints per malware type until 90 per cent of the rows are benign actions and 10 per cent is malicious. The malicious datapoints that are removed, are chosen at random.

4.5 Data formatting

The machine learning package *Sci-kit learn* is used for the training and testing of the machine learning classifiers. This package requires all categorical features to be numerical. Therefore the target label *actiontype* with values *benign* and *malicious* have been transformed into binary values. A zero value represents a benign action and one value represents malicious actions. Additionally, the columns with a string datatype are transformed using a OneHotEncoder. More on this in Section 5.2.5.

Chapter 5

Modelling

This chapter describes the modelling phase of this research. During this phase, the dataset is used for the training and testing of malware detection models. Figure 5.1 shows the activities and their output in the modelling phase. The activities are described in the following sections. Section 5.1 describes the selection of the different machine learning techniques. Section 5.2 describes the experiment setups. Section 5.3 describes how the machine learning techniques were used to train and test different classifiers. Lastly, Section 5.4 describes additional experiment performed, which were run after evaluation of the earlier experiments.

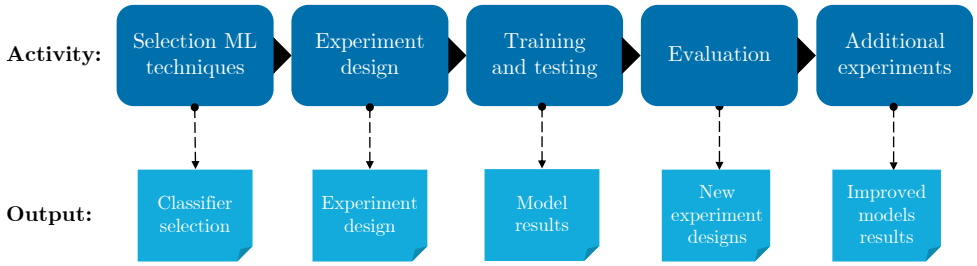


FIGURE 5.1: Modelling activities and output

5.1 Selection machine learning techniques

Based on previous studies on dynamic malware detection, described in Section 2.5, the most promising machine learning techniques are selected. The Random Forest, Naïve Bayes and K-Nearest Neighbour classifiers showed the highest performance in the actual malware detected (TPR) and low amount of false flags (FPR and FNR). Additionally, recent research on the use of Neural Networks for the detection of mobile malware has shown promising results. Therefore, this study uses Random Forest, Naive Bayes, K-Nearest Neighbour, and Neural Network classifiers for the detection model.

5.2 Experimental design

An overview of the experiments is shown in Table 5.1. This figure shows that the experiments are grouped per classifier, dataset, featureset, training mode, and testing set. These different groupings are described below. First, the target label of the classifiers is described.

Classifier	Dataset	Featuresets	Training mode	Testing mode
RF | NB | KNN | MLP | Ada	Basic | Advanced	fs1 | fs2 | fs3 | fs4 | fs5 | fs6	all	Normal holdout
RF | NB | KNN | MLP | Ada	Basic | Advanced	fs1 | fs2 | fs3 | fs4 | fs5 | fs6	permwtype	Normal holdout
RF | NB | KNN | MLP | Ada	Basic | Advanced	fs1 | fs2 | fs3 | fs4 | fs5 | fs6	all	Unknown device
RF | NB | KNN | MLP | Ada	Basic | Advanced	fs1 | fs2 | fs3 | fs4 | fs5 | fs6	permwtype	Unknown device

TABLE 5.1: Experiments overview

5.2.1 Label

Labels refer to the target label that the classifier tries to predict. Two potential labels exist for the detection of mobile malware: i) action type, ii) session type. Action type describes whether an action is benign or malicious and session type describe whether a session is benign or malicious. Actions belong to a specific session. Session types are fixed at the launch of the malware app. As described in Section 3.2.1, within benign sessions, only benign actions are performed and within malicious sessions, both benign and malicious actions are performed. The action types and session types are transformed to binary values, as classifiers require numerical values instead of string values. Therefore, a *benign* value for session types or action types are transformed to a zero, and a *malicious* value for sessiontypes or actiontypes are transformed to a one.

Experiments with both labels are run, although the focus is on action types. Action types are more preventive than session types. A model detecting malicious actions can immediately detect malware and countermeasures can be taken. A model detecting malicious sessions can only detect malware after the malicious behaviour already took place.

5.2.2 Datasets

As described in Section 4.3, two datasets are used for the training and testing of the classifiers. A comparison of both datasets is shown in Figure 5.2. This Figure shows that the Basic dataset is 3.7 times larger than the Advanced dataset. This is because the Advanced features were only available after 2016 (Q2). Additionally, the Basic dataset contains more different malware types than the Advanced dataset. For these reasons, the results of the Basic dataset are more relevant than the results of the Advanced dataset.

Dataset	Basic	Advanced
Size (MB)	21.4	11.41
Total rows	28.821	7.850
Total columns	103	204
Malware types	v1 – v12 (exc. v10)	v5 – v12 (exc. v10)
Possible featuresets	fs1 – fs3	fs1 – fs6

FIGURE 5.2: Datasets overview

5.2.3 Training mode

Training mode refers to how the training of the classifiers is performed. Two different training modes are used throughout this research. The first training mode, called *all*, uses the complete dataset to train one classifier model. The second training mode, called *peruser*, trains a classifier model per user resulting in 47 different models. The second training mode, *permwtype*, trains the classifier models per malware type, resulting in 10 different models. The different training modes are shown graphically in Figure 5.3.

5.2.4 Testing mode

Testing mode refers to how the testing of the classifiers is performed. Two testing modes are used throughout this research. Both testing modes use an x percentage of the data for training and the remaining *1-x* percentage of the data for testing. In the *normal holdout* testing mode, the training and testing data is sampled at random. In the *unknown device* testing mode, the testing data contain data of devices that are not in the training data. The *unknown device* testing mode is used to simulate a situation in which no data is yet available for a certain device. In that case, a model trained on from data of other devices might be a potential solution. Testing the models on data of other devices also helps to remove the potential bias of the model on a specific device. Note here that only the testing on the test set is adjusted, not the testing on the validation set during cross-validation. The cross-validation method is

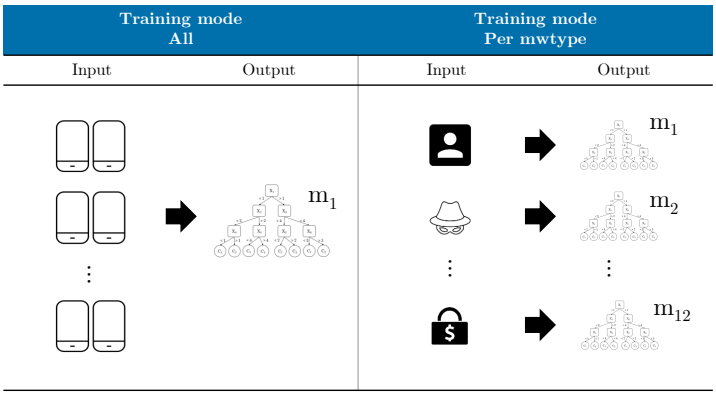


FIGURE 5.3: Training modes overview

described in Section 5.3. Adjusting the testing of the validation set is suggested for future research. More on this in Section 9.2

5.2.5 Featureset

Featureset refers to which features were used in the training and testing of the classifier. Six featuresets are used to train and test the classifiers. The featuresets are divided into Basic featuresets and Advanced featuresets, referring respectively to the features available before month 5 of 2016, and the features available after month 5 of 2016. An overview of the different featuresets is shown in Figure 5.4. Featuresets 1 and 4, i.e. Basic global features, and Advanced global features, include global device features from the System probe. Featureset 2 and featureset 5 include Apps features from the Apps probe. Featuresets 3 and featuresets 6 combine the Global and Apps features. As noted in Section 5.2.2 no malware data is available of featureset 4, 5, and 6, for malware version 1, 2, 3, 4, and 5.

All metadata features, such as timestamps or malware versions, are removed from all featuresets. From the remaining features, the string features are transformed using a OneHotEncoder. This transformation creates a new binary column for each possible value of a string column. For example: the feature *state* has five possible values: R, S, D, T, Z. The OneHotEncoder transforms this feature into five new columns: state_R, state_S, state_D, etc. For each row, the columns with the actual state have one value and, and the other columns have a zero value. The specific features per featureset are shown in Appendix H.

Featureset	Basic global (fs1)	Basic app (fs2)	Basic comb. (fs3)	Adv. global (fs4)	Adv. app (fs5)	Adv. comb. (fs6)
Total features	33	41	74	123	53	176
Features	Battery, CPU, Netw. traffic,	App CPU, App Memory, App Netw. Traffic, App process	*fs1 features + fs2 features*	Battery, CPU, Netw. traffic, I/O Interrupts, Processes, Storage	App CPU, App Memory, App Netw. Traffic, App process. + add. App CPU + add. App Memory + add. App process	*fs4 features + fs5 features*

FIGURE 5.4: Featuresets overview

5.3 Training and testing

Each classifier is trained and tested according to the experiment setups shown in Table 5.1. A 4-fold cross-validation is used to train the dataset. Herein, the dataset is divided into four

parts of equal size. For each training and testing iteration (i.e. a fold) one part is used as a validation set and the remaining three parts are used as a training set. The validation set is changed in each fold. After the four folds, the performance metrics are averaged to get the performance of a model. Subsequently, the test set, containing data that the model has not seen before, is used to get the actual performance of a model. A schematic overview of the cross-validation method and the holdout method is shown in Figure 5.5

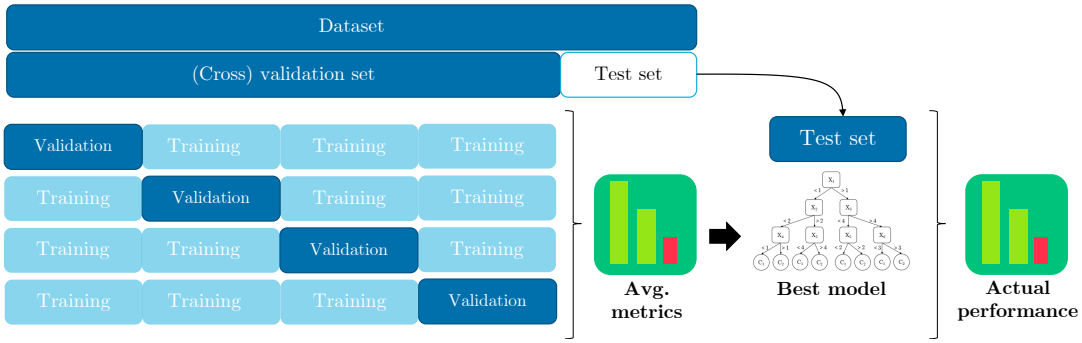


FIGURE 5.5: Schematic view of cross validation method and hold-out method

Using a GridSearch, different hyperparameters are tested for each model. In a GridSearch, each combination of hyperparameter setting in a search space is tested. After the GridSearch, the best model is chosen based on one performance metric. In this research, the *f1_score* is chosen for the selection of the best model. This metric is often used in research [3] and is a harmonic mean of the precision and recall of a performance. These two metrics are normally a trade-off of each other, and a high score indicates a good balance between these two metrics. The search space for the hyperparameter settings per classifier is shown in Table 5.2.

Classifier	Parameter	Search space
AdaBoost	Nr. estimators	[5, 10, 20, 50, 100, 200, 400]
Random Forest	Nr. trees	[3, 5, 10, 20, 40, 80, 160, 320]
	Maximum depth tree	[3, 5, 10, 20, 40, 80, 160, 320]
	Maximum features	[3, 5 10, 20, 40, 60, 80, 100, 120, 140, 171]
Naïve Bayes	Alpha	[0.1, 0.2, 0.3, .., 2.0]
K-nearest neighbours	Nr. neighbours	[1, 3, 7, 11, 21, 31, 61[
Multilayer Perceptron	Nr. layers	[5, 10, 15, 25, 45, 70]
	Layer size (nr. nodes per layer)	[5, 10, 15, 25]

TABLE 5.2: Search space per classifier of GridSearch (AdaBoost is described in Section 5.4)

The detection models are built using a server cluster with a Hadoop File System (HDFS). The HDFS is used to manage the storing, replication and retrieval of data. The cluster consists of 55 nodes with each 32 GB RAM and 12 cores. It uses Hadoop Yarn for job scheduling and monitoring. PySpark 2.2.0 is used to process the dataset. A distributed implementation of the machine learning classifiers of the package Scikit-learn is used to train and test the detection models. This distributed implementation, called Spark Sklearn, is adjusted to provide and log more information during the execution of experiments. Lastly, Jupyter is used to allow for relative easy script debugging. An overview of the environments and tools used can be seen in Figure 5.6.

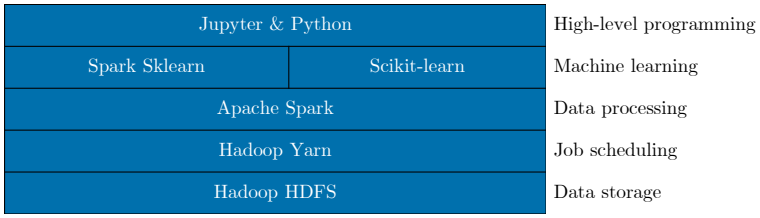


FIGURE 5.6: Data analysis environment and tools

5.4 Additional experiments

After the initial experiment setups, additional experiments were defined. First of all, a recursive feature elimination with cross-validation method was used to tune the number of features for the classifier models. Additionally, an AdaBoost (Ada) classifier was used to analyse the performance of a boosted ensemble classifier. The results of the experiments namely showed a good performance of the Random Forest classifier. The search space for the hyperparameter tuning of Ada is [5, 10, 20, 50, 100, 200, 400] for the hyperparameter *number of estimators*. Below, the recursive feature elimination method is described in more detail.

In recursive feature elimination with cross-validation (RFCV) the classifier is first trained and tested on the complete featureset according to the cross-validation method described in Section 5.3. Subsequently, the least important feature is removed and the classifier is trained again on the reduced featureset. This is repeated until one feature remains. The feature importance can be retrieved from any classifier that calculates feature coefficients or feature importances. In this research, the Random Forest classifier is used to determine the feature importance, as this classifier showed relatively good performance in the experiment results. The feature importance depends on the hyperparameters set in the Random Forest. Therefore, the RFCV is first performed with the Random Forest classifier, as is shown in Figure 5.7. This Figure shows that the feature importances are retrieved from the best performing model, i.e. model performing the best on the cross-validation set. Subsequently, the least important feature is removed and the Random Forest classifier is trained and tested on the reduced dataset. This process is repeated until one feature remains.

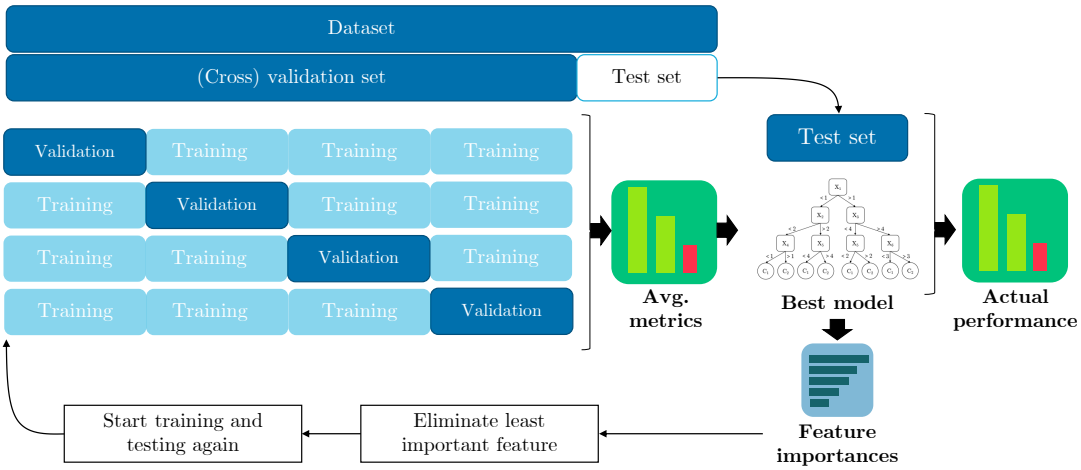


FIGURE 5.7: Recursive feature elimination with cross-validation for Random Forest

The features are ranked according to the iteration in which they were dropped. The feature dropped at the beginning is ranked last, and the feature remaining last is ranked first. The feature ranking is saved and used in the RFCV method of the remaining classifiers (NB, KNN, MLP, Ada). The RFCV method of these classifiers is shown in Figure 5.8. This Figure shows that these classifiers use the feature ranking calculated during the RFECV of the Random Forest classifier to determine which feature to drop. The feature with the lowest ranking is dropped after an iteration.

Note here that the feature ranking is calculated per experiment setup of the RF classifier, i.e. per dataset, training mode, testing mode, and featureset.

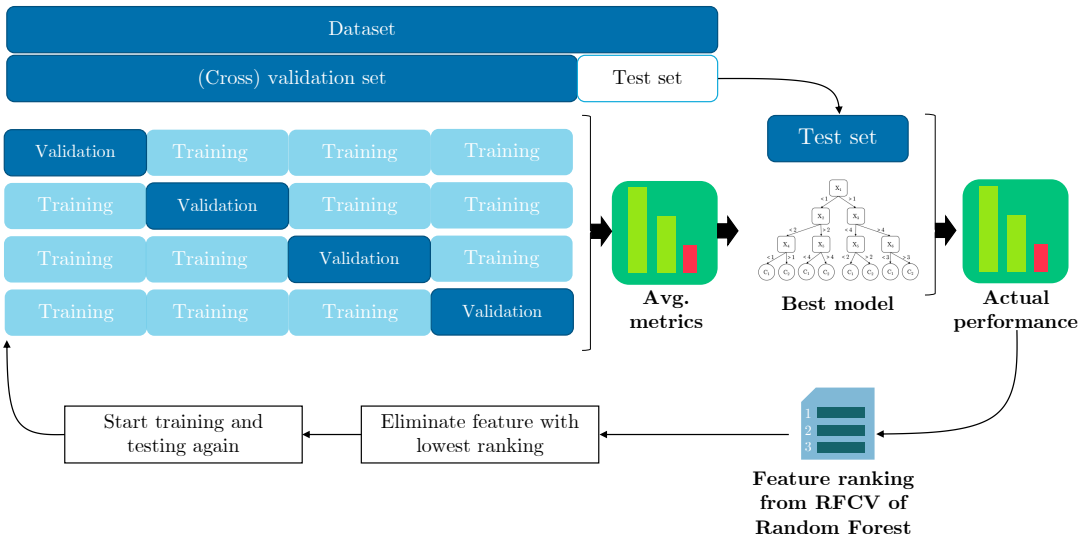


FIGURE 5.8: Recursive feature elimination with cross-validation for the remaining classifiers

Chapter 6

Results

This chapter describes the results of the experiments. An experiment setup refers to one experiment consisting of a specific classifier, training mode, testing mode, featureset, number of features selected, and hyperparameter search space. All results in this Chapter are shown for the best[1] number of features, i.e. the number of features leading to the highest f1 score for a particular experimental setup. Additionally, the results in this Chapter are only shown for the basic featuresets. As noted before, the advanced featuresets are only applicable to the advanced dataset that consists of fewer data points and fewer malware types. Therefore we focus on the basic dataset with basic featuresets.

In this chapter, when comparing classifier performances, colours are added to give a quick insight into relative differences between classifiers and are not meant to show whether a classifier is performing *good* or *bad*. Nevertheless, from here on in this Chapter, we will refer to *high* scores if the classifier has an f1 score above 0.7 (highlighted green in the Figures of this Section), *medium* scores if a classifier has an f1 score between 0.5 and 0.7 (highlighted orange in the Figures of this Section), and *low* scores if a classifier has an f1 score below 0.5 (highlighted red in the Figures of this Section).

Note that all statements in this Chapter regarding statistical significance are based on a McNemar test ($\alpha < 0.05$) [70] between models. The test is used to compare performances of machine learning classifiers on test sets. More on this in Section 8.2.3. The results of these tests are shown in Appendix J.

Section 6.1 compares the performances of classifiers. All results in Section 6.1 are shown for training mode *all*. Section 6.2 describes the performance of classifier for training mode *permwtype*.

6.1 Performance per classifier

This Section describes the performances of the classifiers for training mode *all* testing mode *normal holdout*. Subsections 6.1.1 - 6.1.5 describe the performances of the classifiers in detail. Subsection 6.1.6 compares the performances per classifiers and presents the main findings of the current Section.

6.1.1 Random Forest

The performance of RF for different featuresets and testing modes is shown in Table 6.1. This Table shows the following:

1. RF has the highest f1 score of >0.72 with featureset 2 and featureset 3. The differences between featuresets 2 and 3 are not statistically significant. This suggests that adding global features to app features does not result in a different performance, compared to only using app features.
2. The performance difference for the f1 score between the testing modes *normal holdout* and *unknown device* is statistically significant and above 0.15 for all featuresets, indicating that all classifiers have a lower performance when tested on new devices than if tested on same devices from the training set.

[1] hereafter, *best* always means best with regards to the highest f1-score

3. The performance of RF with featureset 1 is lower than with the other featuresets, for both testing modes *normal holdout* and *unknown device*, suggesting that the RF classifier with global features are not useful in detecting malicious actions of malware.
4. All False Positive Rates are below 0.1.
5. All False Negative Rates are above 0.34, suggesting that most models incorrectly classify a malicious action as benign more often than vice versa.

Best Random Forest classifier (training = all)						
Testing mode	**Normal holdout**			**Unknown device**		
Featureset	**fs1** (Basic global)	**fs2** (Basic apps)	**fs3** (Basic combined)	**fs1** (Basic global)	**fs2** (Basic apps)	**fs3** (Basic combined)
best nr. features	24	29	10	28	6	8
Accuracy	0.922	0.958	0.959	0.888	0.918	0.925
F1 score	0.422	0.730	0.722	0.249	0.562	0.561
FPR	0.022	0.013	0.009	0.052	0.057	0.043
FNR	0.670	0.346	0.380	0.776	0.362	0.423
Nr. trees	5	320	320	40	3	3
Max. depth tree	320	40	20	40	20	10
Max. features	24	10	3	28	5	8

TABLE 6.1: Performance of different classifiers on test set for testing mode *normal holdout* and *unknown device* (the highest f1 score of all featuresets for a particular testing mode is highlighted green)

The performances of the cross-validation results over the hyperparameter search space is shown in Figure 6.1, 6.2, and 6.3, for respectively the parameters *number of trees*, *maximum depth*, and *maximum features*. These performances are shown for the test results of the cross-validation folds for the featuresets with the best number of features, as shown in Table 6.1. The best values for all hyperparameters are shown in Table 6.1. The Figures below show the following:

1. Increasing the number of estimators from 3 to 320 trees improves the f1 score of the models by <2% for all featuresets. This suggests that although the best performing model (with featureset 2 for training *normal holdout*) uses 320 trees, fewer trees may have similar results.
2. Increasing the maximum depth from 3 to 320 improves the f1 score of the models by <25% on average for all featuresets. On average, the best performance is reached at a depth of 40 and 20, for featureset 2 and featureset 3 respectively.
3. Increasing the maximum features considered per split from 3 to (n) (number of features) improves the f1 score of the models by 5% on average for all featuresets.

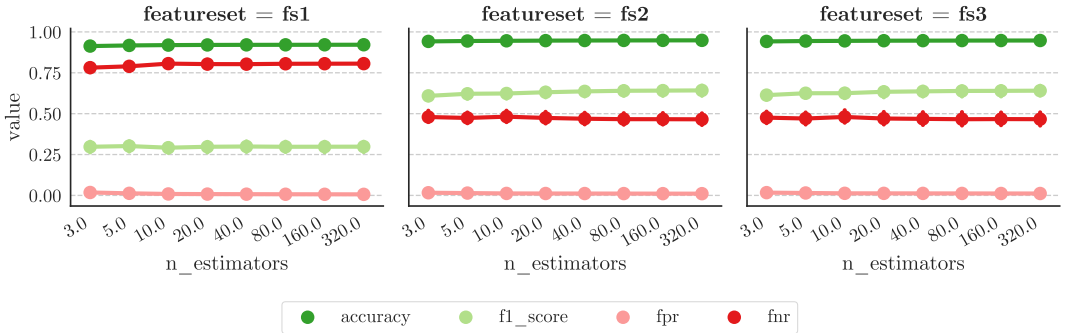


FIGURE 6.1: *Number of estimators* influence on performance for training mode is *all* and testing mode is *normal holdout*

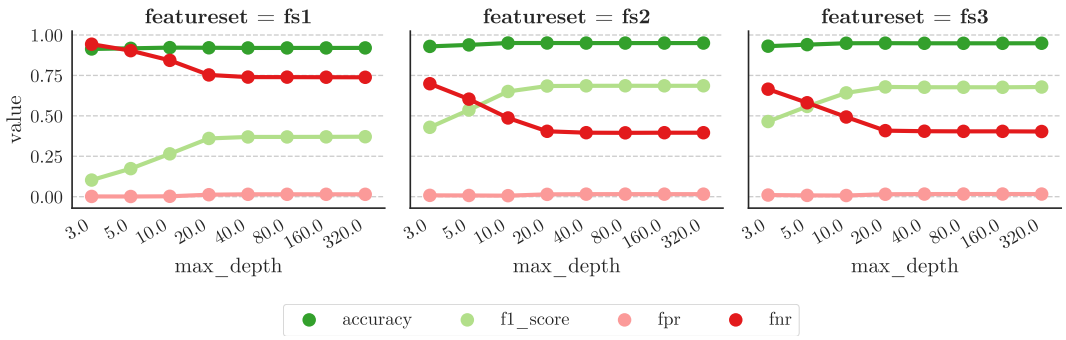


FIGURE 6.2: *Maximum depth* influence on performance for training mode is *all* and testing mode is *normal holdout*

FIGURE 6.3: *Maximum features* influence on performance for training mode is *all* and testing mode is *normal holdout*

Note here that each dot in the above Figures represents an average performance of a classifier over the complete search space, with one hyperparameter value being fixed and showed on the x-axis. The standard deviation of the performances is too small to appear in the above Figures.

RF with Featureset 2 and featureset 3 have similar results. Featureset 3 has the fewest number of features and is, therefore, more suitable for analysing the most important features for detecting malicious actions of malware. The features included in RF with featureset 2 are shown in Table 6.2. This Table shows that 10 features are sufficient for detecting malicious actions of malware.

Features	
dalvikprivatedirty_mor_app	otherpss_mor_app
dalvikpss_mor_app	rss_mor_app
importance_mor_app	stime_mor_app
num_threads_mor_app	utime_mor_app
otherprivatedirty_mor_app	vsize_mor_app

TABLE 6.2: Features included in best RF featureset 3 model. (*ordered by feature ranking from top to bottom, left to right*)

6.1.2 K-nearest neighbour

The performance of the K-nearest neighbour classifier for different featuresets and testing modes is shown in Table 6.3. This Table shows the following:

1. KNN has the highest f1 score of >0.67 with featureset 2 and featureset 3. The differences between featuresets 2 and 3 are not statistically significant. This suggests that adding global features to app features does not result in a different performance, compared to only using app features.
2. The performance difference in the f1 score between the testing modes *normal holdout* and *unknown device* is statistically significant and above 0.15 for all featuresets, indicating

that all classifiers have a lower performance when tested on new devices than if tested on same devices from the training set.

3. The performance of KNN with featureset 1 is lower than with the other featuresets, for both testing modes *normal holdout* and *unknown device*, suggesting that the RF classifier with global features are not useful in detecting malicious actions of malware.
4. All False Positive Rates are below 0.06.
5. All False Negative Rates are above 0.32, suggesting that most models incorrectly classify a malicious action as benign more often than vice versa.

Best K-nearest neighbour classifier (training = all)						
Testing mode	**Normal holdout**			**Unknown device**		
Featureset	**fs1** (Basic global)	**fs2** (Basic apps)	**fs3** (Basic combined)	**fs1** (Basic global)	**fs2** (Basic apps)	**fs3** (Basic combined)
best nr. features	26	13	9	30	4	11
Accuracy	0.897	0.944	0.946	0.824	0.917	0.910
F1 score	0.356	0.674	0.688	0.202	0.537	0.522
FPR	0.048	0.029	0.028	0.126	0.053	0.061
FNR	0.674	0.336	0.320	0.732	0.415	0.407
k (nr. neighbours)	1	1	1	1	31	1

TABLE 6.3: Performance of different classifiers on test set for testing mode *normal holdout* and *unknown device* (the highest f1 score of all featuresets for a particular testing mode is highlighted green)

The performances of the cross-validation results over the hyperparameter search space for k is shown in Figure 6.4. These performances are shown for the test results of the cross-validation folds for the featuresets with the best number of features, as shown in Table 6.3. The best values for k hyperparameters is also shown in Table 6.3. Figure 6.4 shows the following:

1. Overall, increasing the number of neighbours considered, decreases the f1 score of models for all featuresets.
2. The best number of neigbours to consider with training mode *all* is 1 for all featuresets.

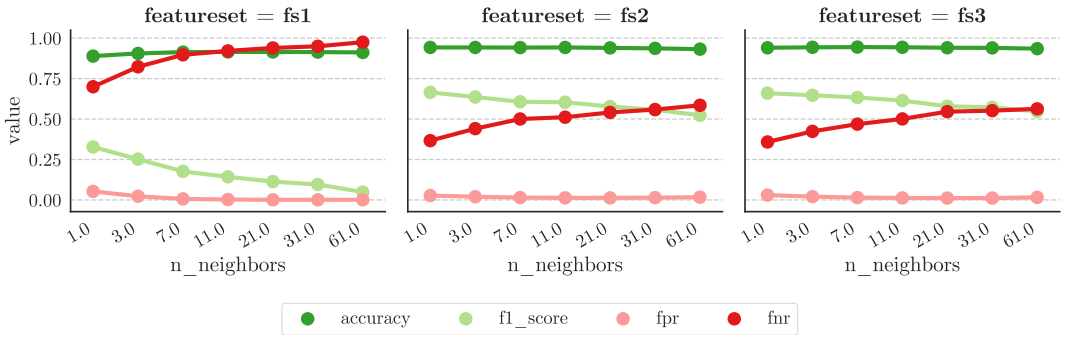


FIGURE 6.4: K influence on performance for training mode is *all* and testing mode is *normal holdout*

KNN with Featureset 2 and featureset 3 have similar results. Featureset 3 has the fewest number of features and is, therefore, more suitable for analysing the most important features for detecting malicious actions of malware. The features included in KNN with featureset 2 are shown in Table 6.4. This Table and Table 6.2 show that KNN with featureset 3 uses the same features as RF with featureset 3, excluding feature *stime*, which is present in the RF model and not in the KNN model. Note here that KNN uses the feature ranking of RF for feature selection with RFCV. From here on no specific features of featuresets are shown for the performance of the remaining classifiers, as the NB and MLP classifier show low performance.

Features	
rss_mor_app	otherpss_mor_app
utime_mor_app	dalvikprivatedirty_mor_app
dalvikpss_mor_app	num_threads_mor_app
otherprivatedirty_mor_app	vsize_mor_app
importance_mor_app	

TABLE 6.4: Features included in best KNN featureset 3 model.
(*ordered by feature ranking from top to bottom, left to right*)

6.1.3 Naïve Bayes

The performance of the Naïve Bayes for different featuresets and testing modes is shown in Table 6.5. This Table shows the following:

1. NB has the highest f1 score of >0.34 with featureset 2 and featureset 3. The differences between featuresets 2 and 3 are not statistically significant. This suggests that adding global features to app features does not result in a different performance, compared to only using app features.
2. The performance difference between the testing modes *normal holdout* and *unknown device* is statistically significant and above 0.1 for all featuresets, indicating that all classifiers have a lower performance when tested on new devices than if tested on same devices from the training set.
3. The f1 score of NB with featureset 1 is zero with an FNR of 1, indicating that these NB models only predict benign labels. This suggests that the NB classifier with global features are not useful in detecting malicious actions of malware.
4. All False Positive Rates are below 0.1.
5. All False Negative Rates are above 0.75, suggesting that most models incorrectly classify a malicious action as benign more often than vice versa.

Best Naïve Bayes classifier (training = all)						
Testing mode	**Normal holdout**			**Unknown device**		
Featureset	**fs1** (Basic global)	**fs2** (Basic apps)	**fs3** (Basic combined)	**fs1** (Basic global)	**fs2** (Basic apps)	**fs3** (Basic combined)
best nr. features	31	33	64	31	29	61
Accuracy	0.913	0.920	0.919	0.917	0.898	0.902
F1 score	0.000	0.343	0.351	0.000	0.259	0.278
FPR	0.000	0.015	0.017	0.000	0.040	0.037
FNR	1.000	0.760	0.748	1.000	0.785	0.772
Alpha	0.5	1.6	0.4	1.3	1.9	1.7

TABLE 6.5: Performance of different classifiers on test set for testing mode *normal holdout* and *unknown device* (the highest f1 score of all featuresets for a particular testing mode is highlighted green)

The performances of the cross-validation results over the hyperparameter search space for *alpha* is shown in Figure 6.5. These performances are shown for the test results of the cross-validation folds for the featuresets with the best number of features, as shown in Table 6.5. The best values for *alpha* is also shown in Table 6.5.
Figure 6.5 shows that the smoothing parameter has a small influence on the f1 score of the NB models with overall less than 0.1% difference between *alpha* settings.

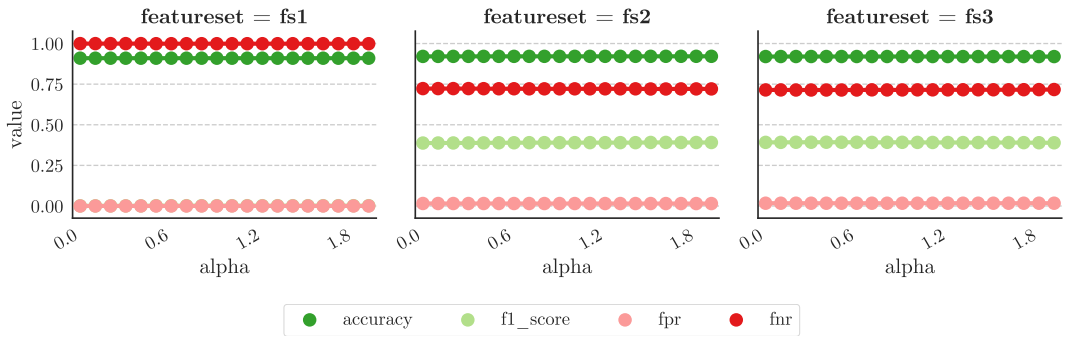


FIGURE 6.5: *Alpha* influence on performance for training mode is *all* and testing mode is *normal holdout*

6.1.4 Multilayer Perceptron

The performance of MLP for different featuresets and testing modes, is shown in Table 6.6. This Table shows the following:

1. MLP has the highest f1 score of (0.567) with featureset 3. The differences between featuresets 2 and 3 are statistically significant. This suggests that adding global features to app features results in a different performance, compared to only using app features.
2. The performance difference between the testing modes *normal holdout* and *unknown device* is statistically significant for all featuresets except featureset 2.
3. The f1 score of MLP with featureset 1 (training mode *normal holdout*) is zero with an FNR of 1, indicating that these MLP models only predict benign labels. This suggests that the MLP classifier with global features are not useful in detecting malicious actions of malware.
4. All False Positive Rates are below 0.04.
5. All False Negative Rates are above 0.53, suggesting that most models incorrectly classify a malicious action as benign more often than vice versa.

Best Multilayer perceptron classifier (training = all)						
Testing mode	**Normal holdout**			**Unknown device**		
Featureset	**fs1** (Basic global)	**fs2** (Basic apps)	**fs3** (Basic combined)	**fs1** (Basic global)	**fs2** (Basic apps)	**fs3** (Basic combined)
best nr. features	32	28	62	28	4	7
Accuracy	0.913	0.937	0.940	0.917	0.920	0.917
F1 score	0.000	0.493	0.567	0.114	0.490	0.000
FPR	0.000	0.007	0.014	0.007	0.040	0.000
FNR	1.000	0.648	0.546	0.935	0.533	1.000
Nr. layers	5	10	5	10	15	10
Nr. nodes per layer	25	25	15	10	15	15

TABLE 6.6: Performance of different classifiers on test set for testing mode *normal holdout* and *unknown device* (the highest f1 score of all featuresets for a particular testing mode is highlighted green)

The performances of the cross-validation results over the hyperparameter search space for the *number of layers* and *number of nodes per layer* is shown in Figures 6.6 and 6.7 respectively. These performances represent the test results of the cross-validation folds for the featuresets with the best number of features, as shown in Table 6.6. The best values for the hyperparameters are also shown in Table 6.6. The hyperparameter Figures show that overall, an increase in the *layer depth* coincides with a decrease in the f1 score of the MLP models. An increase in the layer size appears to positively affect the f1 score of MLP models. Again, it must be noted that the above statements are regarding average performances, as the findings are based on the average performance over all cross-validation folds, and over all parameter settings of the grid search. Therefore, a particular combination of hyperparameter settings might result in a better performance than expected based on the hyperparameter analysis.

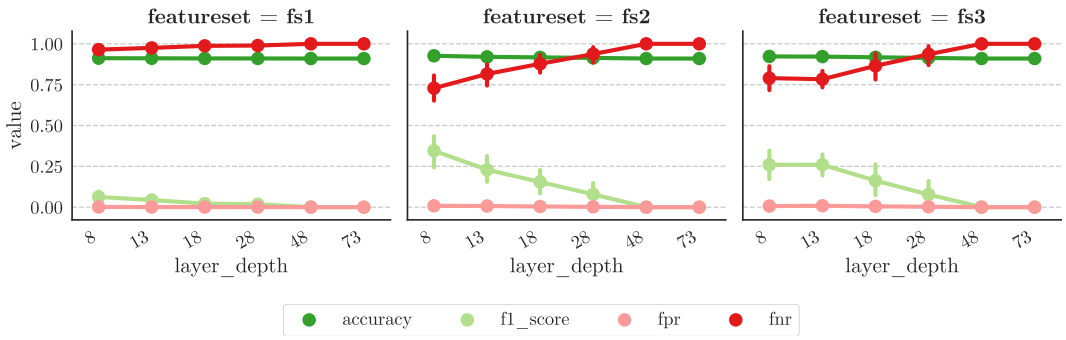


FIGURE 6.6: *Number of layers* influence on performance for training mode is *all* and testing mode is *normal holdout*

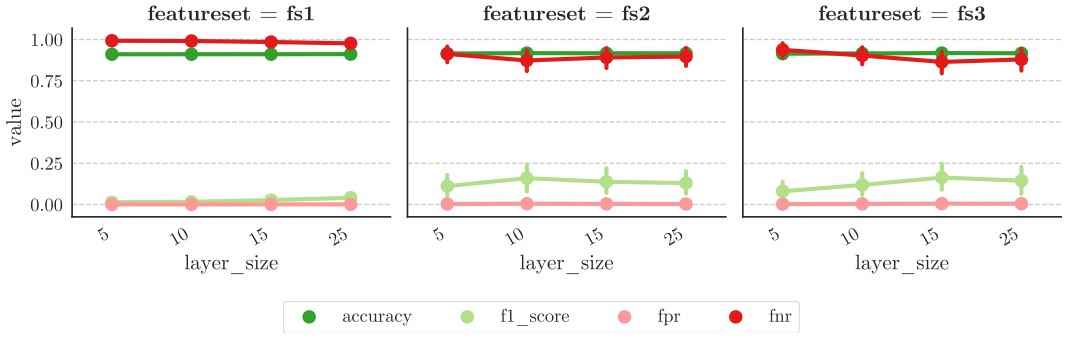


FIGURE 6.7: *Number of nodes per layer* influence on performance for training mode is *all* and testing mode is *normal holdout*

6.1.5 AdaBoost

The performance of Ada for different featuresets and testing modes, is shown in Table 6.7. This Table shows the following:

1. Ada shows the highest f1 score of >0.6 with featureset 2 and featureset 3. The differences between featuresets 2 and 3 are not statistically significant. This suggests that adding global features to app features does not result in a different performance, compared to only using app features.
2. The performance difference between the testing modes *normal holdout* and *unknown device* is not statistically significant among featuresets, e.g. Ada with featureset 2 does not perform differently for testing mode *normal holdout* than for testing mode *unknown device*.
3. The performance of Ada with featureset 1 is lower than with the other featuresets, for both testing modes *normal holdout* and *unknown device*. This suggests that the Ada classifier with global features are not useful in detecting malicious actions of malware.
4. All False Positive Rates are below 0.04.
5. All False Negative Rates are above 0.42, suggesting that most models incorrectly classify a malicious action as benign more often than vice versa.

Best AdaBoost classifier (training = all)						
Testing mode	**Normal holdout**			**Unknown device**		
Featureset	**fs1** (Basic global)	**fs2** (Basic apps)	**fs3** (Basic combined)	**fs1** (Basic global)	**fs2** (Basic apps)	**fs3** (Basic combined)
best nr. features	26	25	43	29	28	44
Accuracy	0.922	0.943	0.945	0.923	0.944	0.926
F1 score	0.226	0.595	0.610	0.207	0.583	0.563
FPR	0.003	0.012	0.013	0.005	0.013	0.042
FNR	0.868	0.522	0.502	0.878	0.528	0.423
Nr. estimators	400	400	400	400	400	50

TABLE 6.7: Performance of different classifiers on test set for testing mode *normal holdout* and *unknown device* (the highest f1 score of all featuresets for a particular testing mode is highlighted green)

The performances of the cross-validation results over the hyperparameter search space for the *number of estimators* is shown in Figure 6.8. These performances are shown for the test results of the cross-validation folds for the featuresets with the best number of features, as shown in Table 6.5. The best values for *number of estimators* is also shown in Table 6.7.
Figure 6.8 shows that the number of estimators increase the performance of the AdaBoost classifier. The maximum performance does not seem to be reached with the maximum number of estimators in the search space.

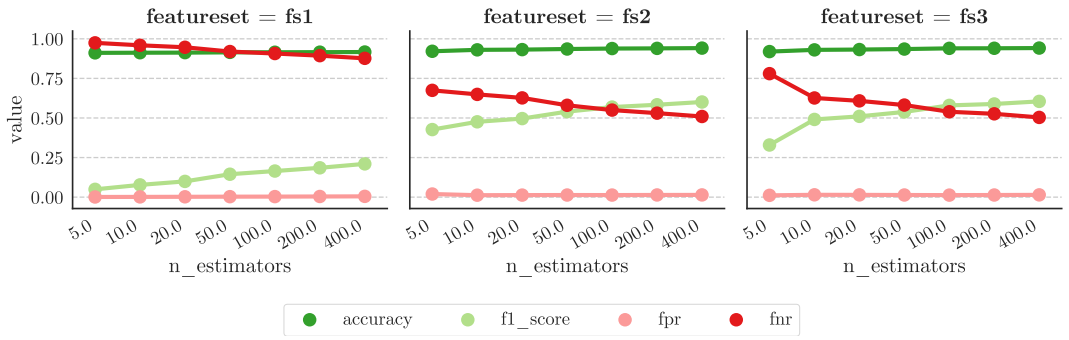


FIGURE 6.8: *Number of estimators* influence on performance for training mode is *all* and testing mode is *normal holdout*

6.1.6 Comparison classifiers

An overview of the performances of the classifiers for testing mode *normal holdout* is shown in Table 6.8. This Table shows the following:

1. RF with featureset 2 has the highest f1 score of 0.73 with featureset 2 and 0.72 with featureset 3. This indicates that the RF classifier is better suited for detecting malicious actions than other classifiers.
2. NB has an f1 score of <0.35 for all featuresets. This suggests that the Naïve Bayes classifier is not useful for detecting malicious actions of multiple malicious applications on multiple devices.
3. All classifiers have an f1 score <0.42 with featureset 1. This suggests that global features are not useful in detecting malicious actions.
4. RF with featureset 3 uses 10 app features (App CPU, App Memory, App Process) to detect malicious actions of malware, suggesting that 10 app features are sufficient in detecting malicious actions of mobile malware. These 10 features are shown in Table 6.2.
5. All classifiers have a higher FNR than FPR. This indicates that most models incorrectly classify a malicious action as benign more often than vice versa.

	Classifier	Ada			RF			KNN			NB			MLP		
Best model	Featureset	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
	Best nr. features	26	25	43	24	29	10	26	13	9	31	33	64	32	28	62
	Best accuracy	0.922	0.943	0.945	0.922	0.958	0.959	0.897	0.944	0.946	0.913	0.920	0.919	0.913	0.937	0.940
	Best f1 score	0.226	0.595	0.610	0.422	0.730	0.722	0.356	0.674	0.688	0.000	0.343	0.351	0.000	0.493	0.567
	Best FPR	0.003	0.012	0.013	0.022	0.013	0.009	0.048	0.029	0.028	0.000	0.015	0.017	0.000	0.007	0.014
	Best FNR	0.868	0.522	0.502	0.670	0.346	0.380	0.674	0.336	0.320	1.000	0.760	0.748	1.000	0.648	0.546
Feature categories in best features	Battery	X		X	X			X			X		X	X		X
	CPU	X		X	X			X			X		X	X		X
	I/O Interrupts															
	Memory	X		X	X			X			X		X	X		X
	Network traff.	X		X	X			X			X		X	X		X
	Storage															
	Wifi															
	App CPU		X	X		X	X		X	X		X	X		X	X
	App memory		X	X		X	X		X	X		X	X		X	X
	App netw. traff.		X	X		X			X			X	X		X	X
	App process		X	X		X	X		X	X		X	X		X	X

TABLE 6.8: Best models of classifiers for training mode *all* and testing mode *normal holdout*

6.2 Performance per malware type

6.2.1 Version 1 - Spyware - contacts theft

An overview with key details of malware version 1 is shown in Table 6.9. Malware version 1 was a puzzle game that stole contacts stored on a device. While running in a malicious session, the malware application stole, encrypted, and transmitted contacts to a remote server every 20 seconds. Each session, the malware application changed its session type from benign to malicious or vice versa. As described in Section 3.2.1, all malware applications perform only benign actions in benign sessions, and perform both benign actions and malicious actions in malicious sessions.

Spyware – contacts theft	
Benign app	Game
Malware type	Spyware
Description	Steals, encrypts, and transmits all contact stored on device.
Total nr. rows	1520

TABLE 6.9: Overview malware version 1

Figure 6.9 shows the count of actions per actiontype. This Figure shows the differences between malicious actions and benign actions taken by malware type 1. Note here that the action *App Mode change* refers to the user pausing or resuming the application. For a description of the other actions, refer to the documentation of the SherLock dataset[2].

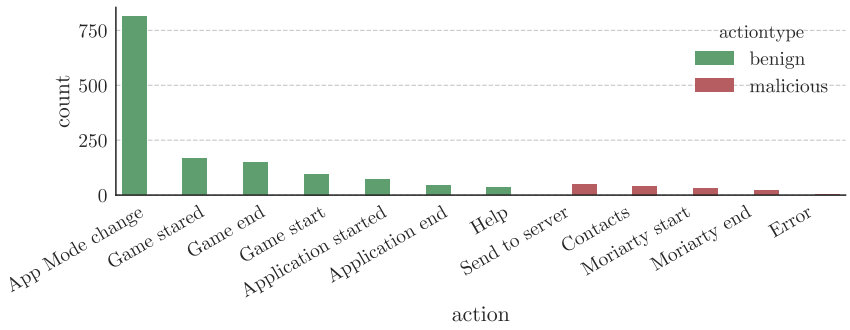


FIGURE 6.9: Distribution of actions of malware version 1 per actiontype

[2]http://bigdata.ise.bgu.ac.il/sherlock/

The performances of the classifiers on malware version 1 is shown in Figure 6.10. This Table shows the following:

1. The best performing classifier is Ada with featureset 3, having the only f1 score above 0.7. This indicates that global and app features combined are useful in detecting malware version 1.
2. KNN with featureset 1 uses 1 feature regarding network traffic (*traffic_totalrxbytes*) to achieve an f1 score of 0.455, where RF with featureset 1 needs 29 features for an f1 score of 0.49, suggesting that KNN is more efficient regarding feature usage in detecting malware version 1.
3. NB has a low f1 score (<0.5) for all featuresets, indicating that NB is not useful in detecting malware version 1.
4. The Multilayer Perceptron has an f1 score of 0 because it only predicts benign actions for all featuresets, resulting in an FNR of 1 and FPR of 0. This indicates that MLP is not useful in detecting malware version 1.

	Classifier	Ada			RF			KNN			NB			MLP		
Best model	Featureset	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
	Best nr. features	19	15	49	29	20	73	1	17	38	18	38	67	24	39	58
	Best accuracy	0.914	0.931	0.947	0.918	0.921	0.931	0.921	0.914	0.901	0.908	0.908	0.888	0.908	0.908	0.908
	Best f1 score	0.480	0.618	0.704	0.490	0.571	0.571	0.455	0.536	0.500	0.000	0.364	0.320	0.000	0.000	0.000
	Best FPR	0.036	0.036	0.025	0.033	0.043	0.025	0.022	0.047	0.062	0.000	0.029	0.051	0.000	0.000	0.000
	Best FNR	0.571	0.393	0.321	0.571	0.429	0.500	0.643	0.464	0.464	1.000	0.714	0.714	1.000	1.000	1.000
Feature caetgories in best features	Battery	X		X	X		X			X	X		X	X		X
	CPU	X		X	X		X			X	X		X	X		X
	I/O Interrupts															
	Memory	X		X	X		X			X	X		X	X		X
	Network traff.	X		X	X		X	X		X	X		X	X		X
	Storage															
	Wifi															
	App CPU		X	X		X	X		X	X		X	X		X	X
	App memory		X	X		X	X		X	X		X	X		X	X
	App netw. traff.		X	X		X	X		X	X		X	X		X	X
	App process		X	X		X	X		X	X		X	X		X	X

TABLE 6.10: Feature categories included in best models of classifiers for malware version 1

The best performing model, with the least number of features, is the AdaBoost classifier with featureset 2. The features included in this model are shown in Table 6.11. This Table shows that 49 global and app features are useful in detecting malicious actions of a mobile (Contact) Spyware malware version.

Features			
totalmemory_freesize	cpu_0	utime_mor_app	traffic_mobiletxbytes
uidrxbytes_mor_app	dalvikprivatedirty_mor_app	uidtxpackets_mor_app	pgid_mor_app
othershareddirty_mor_app	battery_voltage	traffic_totalwifirxbytes	traffic_totalrxpackets
traffic_totalwifitxbytes	rss_mor_app	traffic_totalwifitxpackets	ppid_mor_app
uidtxbytes_mor_app	cstime_mor_app	totalmemory_used_size	nativeprivatedirty_mor_app
stime_mor_app	otherprivatedirty_mor_app	cpu_1	traffic_totalwifirxpackets
total_cpu	traffic_mobiletxpackets	cmaj_flt_mor_app	start_time_mor_app
totalmemory_total_size	traffic_totaltxpackets	num_threads_mor_app	state_mor_app_R
battery_level	cpu_3	vsize_mor_app	cutime_mor_app
cpu_usage_mor_app	dalvikshareddirty_mor_app	traffic_totaltxbytes	
traffic_totalrxbytes	battery_temperature	traffic_mobilerxpackets	
pid_mor_app	dalvikpss_mor_app	cpu_2	
otherpss_mor_app	uidrxpackets_mor_app	nativeshareddirty_mor_app	

TABLE 6.11: Features included in best AdaBoost featureset 2 model.
(*ordered by feature ranking from top to bottom, left to right*)

6.2.2 Version 2 - Spyware - general

An overview with key details malware version 2 is shown in Table 6.12. Malware version 2 was a web browser that spied on the user in two ways. In malicious session 1, the malware applications spied on the location or audio of the user. In malicious session 2, the malware application spied on the web traffic and web history of the user. The malware application changed its session type each session in the following order: benign, malicious 1, malicious 2.

Spyware – general	
Benign app	Web browser
Malware type	Spyware
Description	i) Spies on location and audio, or ii) spies on web traffic and web history.
Total nr. rows	6635

TABLE 6.12: Overview malware version 2

Figure 6.10 shows the count of actions per actiontype. This Figure shows the differences between malicious actions and benign actions taken by malware type 2. Note here that the action *ON DOWN* refers to the user touching any screen object of the browser, e.g. a URL.

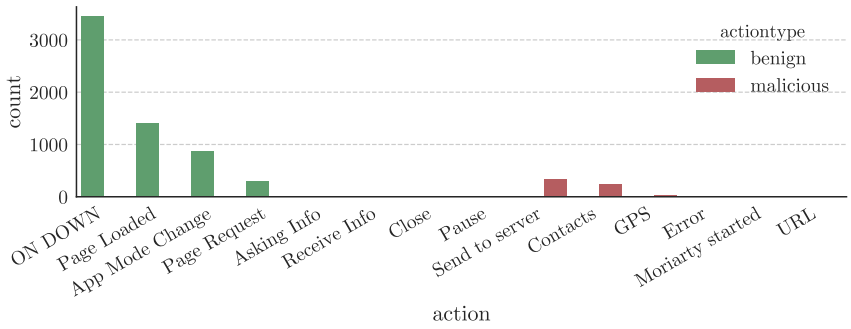


FIGURE 6.10: Distribution of actions of malware version 2 per actiontype

The performances of the classifiers on malware version 2 is shown in Table 6.13 for training mode *all* and testing mode *normal holdout*. This Table shows the following:

1. The RF classifier has a high f1 score (>0.94) for featuresets 2 and 3. The difference in performance between these two featuresets is not statistically significant, suggesting that adding global features to app features does not result in a different performance, compared to only using app features.
2. The difference between KNN with featureset 3 and RF with featureset 2 (or 3) are not statistically significant. This suggests that 8 app features are sufficient for detecting malicious actions of a (general) Spyware malware version. These features are shown in Table 6.14

3. All classifiers have a high f1 score for featureset 2, indicating that app features are useful in detecting malware version 3.
4. NB and MLP predict a benign label for all test instances with featureset 1, leading to an FNR of 1. This indicates that NB and MLP with global features are not useful in detecting malicious actions of malware version 3.

	Classifier	Ada			RF			KNN			NB			MLP		
Best model	Featureset	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
	Best nr. features	26	25	44	12	39	27	14	8	13	10	28	61	29	22	16
	Best accuracy	0.961	0.989	0.983	0.966	0.992	0.987	0.946	0.989	0.989	0.902	0.947	0.951	0.902	0.988	0.983
	Best f1 score	0.780	0.944	0.911	0.802	0.956	0.932	0.698	0.944	0.945	0.000	0.703	0.735	0.000	0.938	0.912
	Best FPR	0.012	0.003	0.004	0.005	0.001	0.003	0.019	0.001	0.003	0.000	0.019	0.021	0.000	0.007	0.005
	Best FNR	0.292	0.085	0.131	0.300	0.077	0.108	0.369	0.100	0.077	1.000	0.362	0.308	1.000	0.062	0.123
Feature caetgories in best features	Battery	X		X	X		X	X			X		X	X		
	CPU	X		X	X		X	X			X		X	X		X
	I/O Interrupts															
	Memory	X		X	X		X	X		X	X		X	X		X
	Network traff.	X		X	X		X	X			X		X	X		X
	Storage															
	Wifi															
	App CPU		X	X		X	X		X	X		X	X		X	X
	App memory		X	X		X	X		X	X		X	X		X	X
	App netw. traff.		X	X		X	X					X	X		X	
	App process		X	X		X	X		X	X		X	X		X	X

TABLE 6.13: Feature categories included in best models of classifiers for malware version 2

Features			
importance_mor_app	utime_mor_app	rss_mor_app	cpu_usage_mor_app
lru_mor_app	dalvikprivatedirty_mor_app	stime_mor_app	othersharedddirty_mor_app

TABLE 6.14: Features included in best KNN featureset 2 model. (*ordered by feature ranking from top to bottom, left to right*)

6.2.3 Version 3 - Spyware - photo theft

An overview with key details malware version 3 is shown in Table 6.15. Malware version 3 was a system monitor app that spied on the photos of a device. In a malicious session, the malware app checked every 5 minutes whether a new picture was taken and if so, sent it to a remote server. When the device was connected to wifi, the malware application checked for new photos and sent these photos every hour, regardless of whether it was in a malicious session. The malware application changed its session type every day.

Spyware – photo theft	
Benign mode	Utilization widget
Malware type	Spyware
Description	Steals photos that are taken and in storage, and takes candid photos of the user.
Total nr. rows	1715

TABLE 6.15: Overview malware version 3

Figure 6.11 shows the count of actions per actiontype. This Figure shows the differences between malicious actions and benign actions taken by malware type 3. As noted before, the action *App Mode Change* refers to the user pausing or resuming the application.

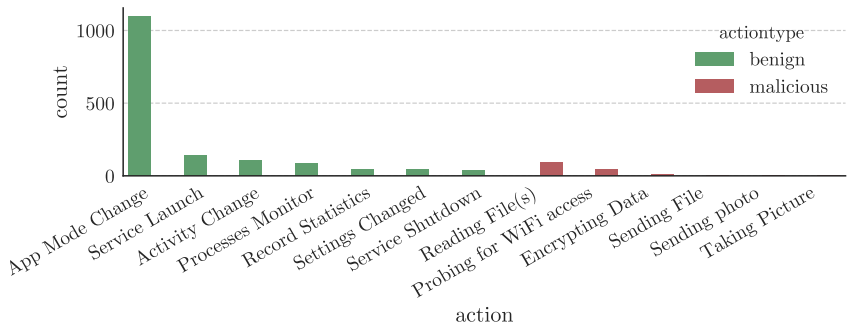


FIGURE 6.11: Distribution of actions of malware version 3 per actiontype

The performances of the classifiers on malware version 3 is shown in Table 6.16. This Table shows the following:

1. RF has a high f1 score (>0.76) for featuresets 2 and 3. The difference in performance between these two featuresets is not statistically significant, suggesting that adding global features to app features does not result in a different performance for RF, compared to only using app features.
2. Ada has a high f1 score (>0.71) for featuresets 2 and 3. The difference in performance between these two featuresets is not statistically significant, suggesting that adding global features to app features does not result in a different performance for Ada, compared to only using app features.
3. The difference between KNN featureset 3 and RF featureset 2 is not statistically significant, suggesting that 5 features are sufficient for detecting malicious actions of a mobile (photo) Spyware malware version. These features are shown in Table 6.17. This Table shows that only app features are used.
4. MLP has an f1 score of 0 for featureset 2 and 3 because it only predicts benign actions, resulting in an FNR of 1 and FPR of 1.
5. MLP has a low f1 score of (<0.38) for all features, suggesting that MLP is not useful in detecting malicious actions of malware version 3.
6. NB has a low f1 score (<0.0) for all featuresets, suggesting that the NB is not useful in detecting malicious actions of malware version 3.

	Classifier	Ada			RF			KNN			NB			MLP		
Best model	Featureset	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
	Best nr. features	24	16	50	21	32	45	31	24	5	10	25	58	29	38	25
	Best accuracy	0.930	0.953	0.962	0.933	0.965	0.962	0.904	0.950	0.956	0.901	0.898	0.898	0.913	0.901	0.901
	Best f1 score	0.455	0.714	0.764	0.511	0.793	0.764	0.267	0.679	0.727	0.000	0.000	0.000	0.375	0.000	0.000
	Best FPR	0.000	0.006	0.000	0.003	0.003	0.000	0.016	0.003	0.003	0.000	0.003	0.003	0.016	0.000	0.000
	Best FNR	0.706	0.412	0.382	0.647	0.324	0.382	0.824	0.471	0.412	1.000	1.000	1.000	0.735	1.000	1.000
Feature categories in best features	Battery	X		X	X		X	X			X		X	X		X
	CPU	X		X	X		X	X			X		X	X		X
	I/O Interrupts															
	Memory	X		X	X		X	X			X		X	X		X
	Network traff.	X		X	X		X	X			X		X	X		X
	Storage															
	Wifi															
	App CPU		X	X		X	X		X	X		X	X		X	X
	App memory		X	X		X	X		X	X		X	X		X	X
	App netw. traff.		X	X		X	X		X			X	X		X	X
	App process		X	X		X	X		X	X		X	X		X	X

TABLE 6.16: Feature categories included in best models of classifiers for malware version 3

Features		
otherpss_mor_app	dalvikprivatedirty_mor_app	otherprivatedirty_mor_app
start_time_mor_app	stime_mor_app	

TABLE 6.17: Features included in best KNN featureset 3 model.
(*ordered by feature ranking from top to bottom, left to right*)

6.2.4 Version 4 - Spyware - SMS

An overview with key details malware version 4 is shown in Table 6.18. Malware version 4 was a pedometer app that read SMSes in the background and sent the message to a remote server when it contained keywords such as *code*, *verification*, and *authentication*. This malware version was always in a malicious session type and listened to SMS messages in the background.

Figure 6.12 shows the count of actions per actiontype. This Figure shows the differences between malicious actions and benign actions taken by malware type 4. As noted before, the action *App Mode Change* refers to the user pausing or resuming the application.

Spyware - SMS	
Benign app	Sports app
Malware type	Spyware
Description	Captures and reports immediately on SMSs that contain codes and various keywords.
Total nr. rows	190

TABLE 6.18: Overview malware version 4

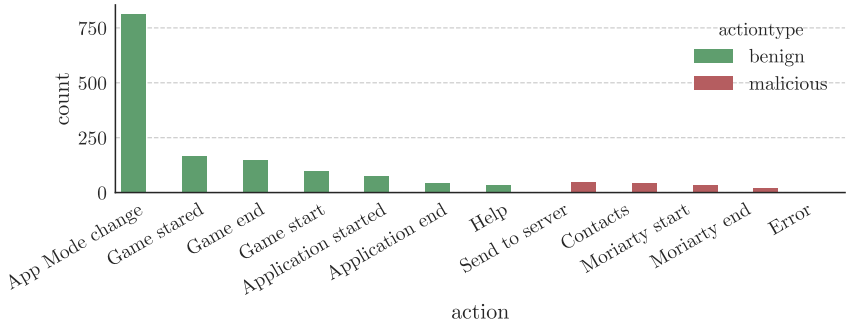


FIGURE 6.12: Distribution of actions of malware version 4 per actiontype

The performances of the classifiers on malware version 1 is shown in Table 6.19. This table shows the following:

1. RF with featureset 1 performs the best of all classifiers but has a medium f1 score of 0.5.
2. All other models have a low f1 score, suggesting that all models are ineffective at detecting malicious actions of malware version 4.
3. 12 out of 15 models have the same FPR and FNR.

	Classifier	Ada			RF			KNN			NB			MLP		
Best model	Featureset	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
	Best nr. features	5	10	11	28	22	17	5	1	1	28	14	10	33	17	14
	Best accuracy	0.921	0.921	0.921	0.947	0.921	0.921	0.921	0.921	0.921	0.921	0.921	0.921	0.921	0.921	0.921
	Best f1 score	0.400	0.400	0.400	0.500	0.400	0.400	0.400	0.400	0.400	0.000	0.400	0.400	0.000	0.400	0.400
	Best FPR	0.029	0.029	0.029	0.000	0.029	0.029	0.029	0.029	0.029	0.000	0.029	0.029	0.000	0.029	0.029
	Best FNR	0.667	0.667	0.667	0.667	0.667	0.667	0.667	0.667	0.667	1.000	0.667	0.667	1.000	0.667	0.667
Feature categories in best features	Battery				X						X			X		
	CPU	X			X			X			X			X		
	I/O Interrupts															
	Memory	X			X			X			X			X		
	Network traff.	X			X			X			X			X		
	Storage															
	Wifi															
	App CPU		X	X		X	X		X	X		X	X		X	X
	App memory		X	X		X	X					X	X		X	X
	App netw. traff.															
	App process		X	X		X	X					X	X		X	X

TABLE 6.19: Feature categories included in best models of classifiers for malware version 4

The best performing model, with the least number of features, is the RF classifier with featureset 1. The features included in this model are shown in Table 6.20. Due to the low performance of the model, no findings regarding relevant features for detecting malware version 4 can be drawn.

Features			
traffic_totalrxbytes	traffic_totalwifitxpackets	battery_plugged	cpu_2
traffic_totalrxpackets	battery_current_avg	battery_scale	cpu_3
traffic_totaltxbytes	battery_health	battery_status	total_cpu
traffic_totaltxpackets	battery_icon_small	battery_temperature	totalmemory_freesize
traffic_totalwifirxbytes	battery_invalid_charger	battery_voltage	totalmemory_max_size
traffic_totalwifirxpackets	battery_level	cpu_0	totalmemory_total_size
traffic_totalwifitxbytes	battery_online	cpu_1	totalmemory_used_size

TABLE 6.20: Features included in best RF featureset 1 model. (*ordered by feature ranking from top to bottom, left to right*)

6.2.5 Version 5 - Phishing

Phishing	
Benign app	Game
Malware type	Phishing
Description	Makes fake shortcuts and notifications to login to Facebook, Gmail, and Skype.
Total nr. rows	3465

TABLE 6.21: Overview malware version 5

An overview with key details malware version 5 is shown in Table 6.24. Malware version 5 was an Angry Birds game that posted phishing attempts. Every two days the application added different shortcuts (Facebook, Skype, and Gmail) on the home screen of the user its device. Each shortcut leads to an app which contained a fake login screen. When the user entered his username and password, the details were sent to a remote server. The malware application was always in a malicious session type. Figure 6.13 shows the differences between malicious actions and benign actions taken by malware type 5. Note here that the action *App Mode Change* can be either malicious or benign, in contrast to the other malware versions described before. The benign *App Mode Change* refers to a mode change within the benign version of the malware, i.e. the game. The malicious *App Mode Change* refers to a mode change within the malicious app of the malware, i.e. the installed fake Facebook, Gmail, or Skype. The action *View Mode Change* refers to the user changing the view within a game, e.g.

transitioning from viewing levels to viewing the actual game.

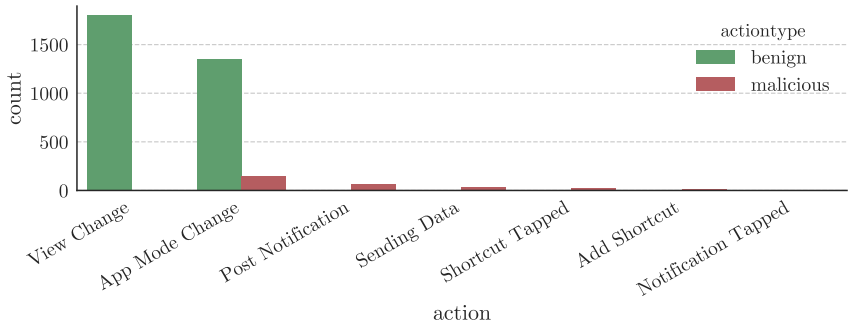


FIGURE 6.13: Distribution of actions of malware version 5 per actiontype

The performances of the classifiers on malware version 5 is shown in Table 6.22. This Table shows the following:

1. KNN with featureset 2 has the highest f1 score of >0.92 for featureset 2 and featureset 3. The difference in performance between these two featuresets is not statistically significant, suggesting that adding global features to app features does not result in a different performance for KNN, compared to only using app features.
2. All classifiers have a low f1 score with featureset 1, indicating that global features are not effective at detecting malicious actions of malware version 5.
3. All classifier have a higher f1 score with featureset 2 than with featureset 3, but this difference is not statistically significant. This suggests that adding global features to app features does not result in a performance difference, for all classifiers, compared to only using app features.
4. NB has a medium or low f1 score for all featuresets, indicating that NB is not useful in detecting malware version 5.

	Classifier	Ada			RF			KNN			NB			MLP		
Best model	Featureset	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
	Best nr. features	25	15	55	8	18	18	5	7	9	16	30	46	26	16	14
	Best accuracy	0.932	0.988	0.984	0.949	0.988	0.986	0.918	0.991	0.987	0.919	0.958	0.958	0.926	0.987	0.984
	Best f1 score	0.447	0.927	0.897	0.588	0.927	0.911	0.412	0.946	0.920	0.000	0.688	0.688	0.215	0.920	0.903
	Best FPR	0.016	0.005	0.005	0.006	0.005	0.008	0.033	0.005	0.008	0.000	0.008	0.008	0.003	0.008	0.009
	Best FNR	0.661	0.089	0.143	0.554	0.089	0.089	0.643	0.054	0.071	1.000	0.429	0.429	0.875	0.071	0.089
Feature caetgories in best features	Battery	X		X	X		X	X			X		X	X		
	CPU	X		X	X						X		X	X		
	I/O Interrupts															
	Memory	X		X	X		X	X			X		X	X		
	Network traff.	X		X	X			X			X		X	X		
	Storage															
	Wifi															
	App CPU		X	X		X	X		X	X		X	X		X	X
	App memory		X	X		X	X		X	X		X	X		X	X
	App netw. traff.		X	X		X	X					X	X		X	X
	App process		X	X		X	X					X	X		X	X

TABLE 6.22: Feature categories included in best models of classifiers for malware version 5

The best performing model, with the least number of features, is the KNN classifier with featureset 2. The features included in this model are shown in Table 6.23. This Table shows that 7 app features are sufficient for detecting malicious actions of a mobile Phishing malware version.

Features			
dalvikpss_mor_app	dalvikprivatedirty_mor_app	rss_mor_app	cpu_usage_mor_app
utime_mor_app	priority_mor_app	vsize_mor_app	

TABLE 6.23: Features included in best KNN featureset 2 model.
(*ordered by feature ranking from top to bottom, left to right*)

6.2.6 Version 6 - Adware

An overview with key details malware version 6 is shown in Table 6.24. Malware version 6 was a game (Chase Whisply). At the start of a malicious session, the malware application displayed advertisements and gathered the following information on device info, sim card details, and list of installed apps. Then, every 15 seconds the app sampled location coordinates from the network, mobile network information, or list of running application. The malware application changed its session type in the following sequence: benign, malicious, malicious.

Adware	
Benign app	Game
Malware type	Adware
Description	Gathers information and places ads, popups and banners.
Total nr. rows	7940

TABLE 6.24: Overview malware version 6

Figure 6.14 shows the count of actions per actiontype. This Figure shows the differences between malicious actions and benign actions taken by malware type 6.

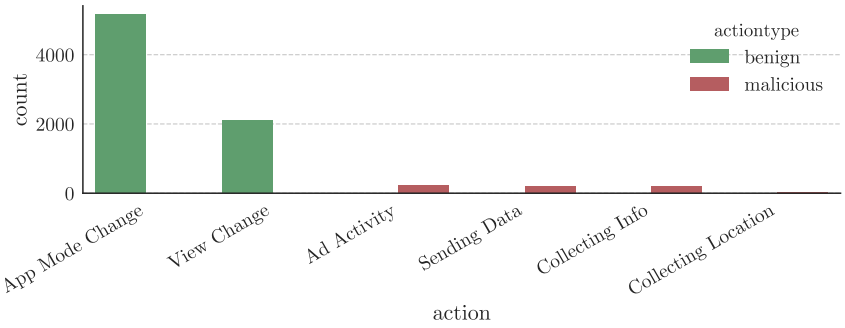


FIGURE 6.14: Distribution of actions of malware version 6 per actiontype

The performances of the classifiers on malware version 6 is shown in Table 6.25. This Table shows that all models have a low f1 score and a high FNR, meaning that all models mostly predict a benign label. This suggests that all models are not effective in detecting malware version 6.

	Classifier	Ada			RF			KNN			NB			MLP		
Best model	Featureset	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
	Best nr. features	24	18	36	23	7	55	1	11	6	27	30	66	33	30	62
	Best accuracy	0.908	0.911	0.906	0.901	0.895	0.901	0.887	0.889	0.887	0.909	0.911	0.908	0.911	0.911	0.911
	Best f1 score	0.076	0.194	0.186	0.233	0.271	0.270	0.268	0.352	0.297	0.000	0.027	0.027	0.000	0.000	0.000
	Best FPR	0.007	0.011	0.017	0.028	0.039	0.030	0.049	0.057	0.053	0.001	0.001	0.004	0.000	0.000	0.000
	Best FNR	0.958	0.880	0.880	0.831	0.782	0.796	0.768	0.662	0.732	1.000	0.986	0.986	1.000	1.000	1.000
Feature caetgories in best features	Battery	X		X	X		X				X		X	X		X
	CPU	X		X	X		X				X		X	X		X
	I/O Interrupts															
	Memory	X		X	X		X				X		X	X		X
	Network traff.	X		X	X		X	X		X	X		X	X		X
	Storage															
	Wifi															
	App CPU		X	X		X	X		X	X		X	X		X	X
	App memory		X	X		X	X		X	X		X	X		X	X
	App netw. traff.		X	X		X	X		X			X	X		X	X
	App process		X	X			X					X	X		X	X

TABLE 6.25: Feature categories included in best models of classifiers for malware version 6

The best performing model, with the least number of features, is the KNN classifier with featureset 2. The features included in this model are shown in Table 6.26. Due to the low performance of the model, no findings regarding relevant features for detecting malware version 6 can be drawn.

Features			
otherpss_mor_app	uidrxbytes_mor_app	num_threads_mor_app	uidtxbytes_mor_app
dalvikpss_mor_app	othershareddirty_mor_app	rss_mor_app	stime_mor_app
utime_mor_app	cpu_usage_mor_app	vsize_mor_app	

TABLE 6.26: Features included in best KNN featureset 2 model. (*ordered by feature ranking from top to bottom, left to right*)

6.2.7 Version 7 - Spyware, Adware, Hostile downloader

An overview with key details malware version 7 is shown in Table 6.27. Malware version 7 was a game (Collider), that had two malicious modes: malicious1, and malicious2. In malicious1, the app downloaded a file and then gathered information on account details, location coordinates, and phone number. In malicious2, the app posted advertisements in the notification bar, posted phishing shortcuts on the home screen, played a voice advertisement every 6 phone calls, and presented Google Play installation pages of random apps every 8th time the user interacted with its device.

Hostile downloader	
Benign app	Game
Malware type	Hostile downloader
Description	Gathers private information and places shortcuts, notifications, and attempts to install new applications.
Total nr. rows	5205

TABLE 6.27: Overview malware version 7

Figure 6.15 shows the count of actions per actiontype. This Figure shows the differences between malicious actions and benign actions taken by malware type 7. As noted before, the action *View Mode Change* refers to the user changing the view within a game, e.g. transitioning from viewing levels to viewing the actual game.

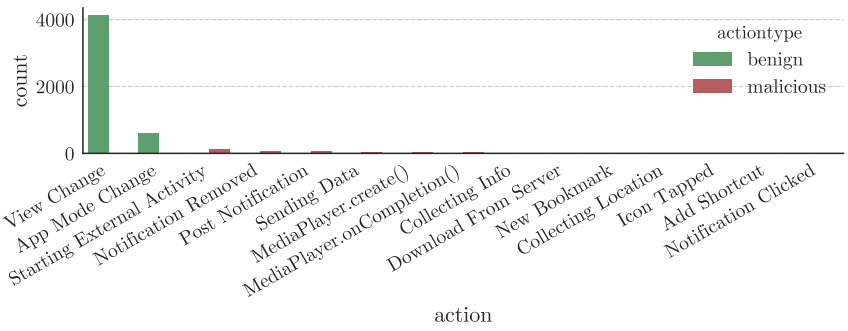


FIGURE 6.15: Distribution of actions of malware version 7 per actiontype

The performances of the classifiers on malware version 7 is shown in Table 6.28. This Table shows the following:

1. The RF classifier has the highest f1 score (>0.84) for featuresets 2 and 3. The difference in performance between these two featuresets is not statistically significant, suggesting that adding global features to app features does not result in a different performance for RF, compared to only using app features.
2. All classifiers have a low f1 score with featureset 1, indicating that global features are not useful in detecting malware type 7.
3. NB and MLP with featureset 3 have high f1 scores for featureset 3, suggesting that NB and MLP with global and app features combined, are useful in detecting malware version 7.
4. No statistical difference is found between featureset 2 and featureset 3 of classifiers (except for MLP), indicating that adding global features to app features does not result in a performance difference, for all classifiers, compared to including only app features.

	Classifier	Ada			RF			KNN			NB			MLP		
Best model	Featureset	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
	Best nr. features	23	24	14	20	9	18	33	9	9	30	16	17	28	9	10
	Best accuracy	0.908	0.966	0.965	0.908	0.976	0.973	0.888	0.964	0.964	0.906	0.944	0.948	0.907	0.907	0.967
	Best f1 score	0.333	0.813	0.804	0.351	0.857	0.843	0.339	0.793	0.793	0.020	0.678	0.700	0.000	0.000	0.813
	Best FPR	0.024	0.015	0.014	0.026	0.003	0.006	0.053	0.012	0.012	0.002	0.023	0.021	0.000	0.000	0.012
	Best FNR	0.753	0.216	0.237	0.732	0.227	0.227	0.691	0.268	0.268	0.990	0.371	0.351	1.000	1.000	0.237
Feature catgories in best features	Battery	X			X		X	X			X		X	X		
	CPU	X			X			X			X			X		
	I/O Interrupts															
	Memory	X			X		X	X			X		X	X		
	Network traff.	X			X		X	X			X		X	X		
	Storage															
	Wifi															
	App CPU		X	X		X	X		X	X		X	X		X	X
	App memory		X	X		X	X		X	X		X	X		X	X
	App netw. traff.		X	X			X					X	X			
	App process		X	X		X	X		X	X		X	X		X	X

TABLE 6.28: Feature categories included in best models of classifiers for malware version 7

The best performing model, with the least number of features, is the RF classifier with featureset 2. The features included in this model are shown in Table 6.29. This Table shows that 9 app features are useful in detecting malicious actions of a mobile Spyware / Adware / Hostile downloader malware version.

Features		
dalvikprivatedirty_mor_app	num_threads_mor_app	priority_mor_app
dalvikpss_mor_app	otherprivatedirty_mor_app	rss_mor_app
importance_mor_app	otherpss_mor_app	utime_mor_app

TABLE 6.29: Features included in best RF featureset 2 model. (*ordered by feature ranking from top to bottom, left to right*)

6.2.8 Version 8 - Ransomware

Ransomware	
Benign app	Lock-screen
Malware type	Ransomware
Description	Performs either: 1) lock screen ransomware, or 2) crypto ransomware.
Total nr. rows	170

TABLE 6.30: Overview malware version 8

An overview with key details malware version 8 is shown in Table 6.30. Malware version 8 was a lock screen application, that performed either i) lockscreen ransomware or ii) crypto ransomware. The documentation on this malware version is missing from the SherLock website. Therefore no exact details can be given on its behaviour. However, based on the specific actions, shown in Figure 6.16, we deduce that lockscreen ransomware refers to the malware application blocking the lockscreen unless a ransom (in this case, assumed fictional because the data collection was voluntary) is paid. The crypto ransomware is assumed to encrypt files or folders unless a (fictional) ransom is paid. As the malware application was based on the wild malware *Simplocker*, we also assume that the user was tricked into enabling accessibility permissions for the malware application.

Figure 6.16 shows the count of actions per actiontype. This Figure shows the differences between malicious actions and benign actions taken by malware type 8. As noted before, the action *View Mode Change* refers to the user changing the view within the application. We assume that this is, for example, the application transitioning from lock screen to home screen. However, due to the missing documentation, we are unable to verify this.

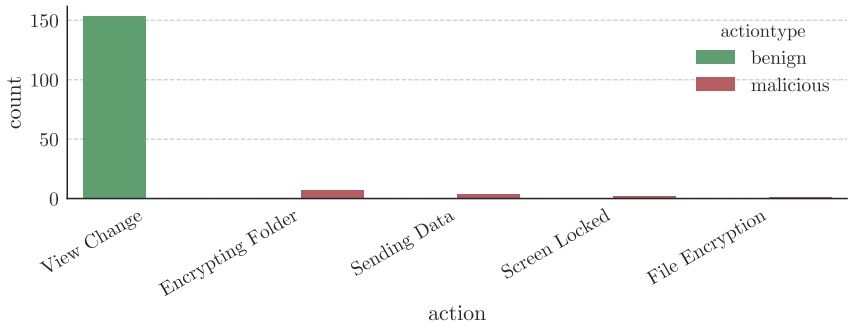


FIGURE 6.16: Distribution of actions of malware version 8 per actiontype

The performances of the classifiers on malware version 8 is shown in Table 6.31. This Table shows the following:

1. The best performing classifier is Ada with a perfect score for featureset 2 and 3.
2. All classifiers have a low f1 score with featureset 1, indicating that global features are not useful in detecting malware type 8.
3. Most classifier (3 out of 4) have a high f1 score with featureset 3, indicating that a combination of global and app features are useful in detecting malware type 8.
4. KNN has a higher f1 score than RF for all featuresets and uses fewer features. KNN uses only memory features of the application, indicating that this feature category is useful in detecting malware type 8.

	Classifier	Ada			RF			KNN			NB			MLP		
Best model	Featureset	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
	Best nr. features	13	19	14	26	33	28	5	4	4	10	25	33	11	34	38
	Best accuracy	0.912	1.000	1.000	0.912	0.971	0.971	0.941	0.971	0.971	0.912	0.912	0.941	0.088	0.971	0.971
	Best f1 score	0.400	1.000	1.000	0.000	0.800	0.800	0.500	0.857	0.857	0.000	0.000	0.500	0.162	0.800	0.800
	Best FPR	0.032	0.000	0.000	0.000	0.000	0.000	0.000	0.032	0.032	0.000	0.000	0.000	1.000	0.000	0.000
	Best FNR	0.667	0.000	0.000	1.000	0.333	0.333	0.667	0.000	0.000	1.000	1.000	0.667	0.000	0.333	0.333
Feature categories in best features	Battery	X			X			X			X			X		
	CPU	X			X			X			X		X	X		X
	I/O Interrupts															
	Memory	X			X		X	X			X		X	X		X
	Network traff.	X			X						X			X		
	Storage															
	Wifi															
	App CPU		X	X		X	X					X	X		X	X
	App memory		X	X		X	X		X	X		X	X		X	X
	App netw. traff.		X			X	X					X	X		X	X
	App process		X	X		X	X					X	X		X	X

TABLE 6.31: Feature categories included in best models of classifiers for malware version 8

The best performing model, with the least number of features, is the AdaBoost classifier with featureset 3. The features included in this model are shown in Table 6.32. This Table shows that solely using 14 app features can be sufficient in detecting malicious actions of a mobile Ransomware malware version.

Features			
rss_mor_app	dalvikpss_mor_app	start_time_mor_app	num_threads_mor_app
dalvikprivatedirty_mor_app	stime_mor_app	priority_mor_app	pid_mor_app
vsize_mor_app	otherprivatedirty_mor_app	ppid_mor_app	
otherpss_mor_app	utime_mor_app	dalvikshareddirty_mor_app	

TABLE 6.32: Features included in best Ada featureset 3 model. (*ordered by feature ranking from top to bottom, left to right*)

6.2.9 Version 9 - Privilege escalation, Spyware

An overview with key details malware version 9 is shown in Table 6.33. Malware version 9 was a game (React), that tricked the user to grant accessibility permissions to steal information about the actions of the users. When the accessibility permissions were granted, the application gathered data on the actions such as clicks and keystrokes. When connected to wifi, and in a malicious session, the malware application sent the gathered data to a remote server hourly. The malware application stayed in a malicious session once the accessibility permissions were granted.

Privilege escalation	
Benign mode	File Manager
Malware type	Privilege escalation
Description	Tricks the user to activate accessibility services to then hijack the user interface.
Total nr. rows	1345

TABLE 6.33: Overview malware version 9

Figure 6.17 shows the count of actions per actiontype. This Figure shows the differences between malicious actions and benign actions taken by malware type 9.

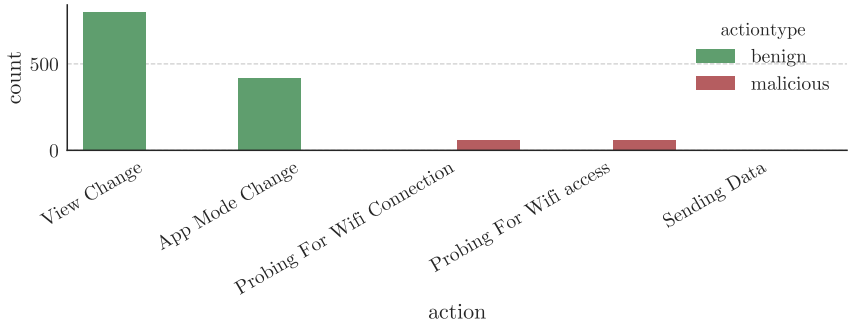


FIGURE 6.17: Distribution of actions of malware version 9 per actiontype

The performances of the classifiers on malware version 9 is shown in Table 6.34. This Table shows the following:

1. The best performing classifiers are RF with featureset 2 or 3, and KNN with featureset 2. The differences between these performances are not statistically significant.
2. All classifiers, except NB have a high f1 score with featureset 3, indicating that a combination of global and app features are useful in detecting malware type 9.
3. All classifiers have a low or medium f1 score with featureset 1, indicating that global features are not useful in detecting malware type 9.
4. Combining global and app features only leads to an improvement for the MLP classifier. This difference is statistically significant.

	Classifier	Ada			RF			KNN			NB			MLP		
Best model	Featureset	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
	Best nr. features	17	11	34	23	31	10	1	19	5	10	39	65	33	34	57
	Best accuracy	0.922	0.974	0.985	0.926	0.989	0.989	0.892	0.989	0.974	0.907	0.918	0.941	0.907	0.941	0.985
	Best f1 score	0.488	0.844	0.913	0.545	0.936	0.936	0.525	0.936	0.844	0.000	0.421	0.652	0.000	0.742	0.923
	Best FPR	0.025	0.004	0.000	0.029	0.000	0.000	0.082	0.000	0.004	0.000	0.020	0.025	0.000	0.057	0.012
	Best FNR	0.600	0.240	0.160	0.520	0.120	0.120	0.360	0.120	0.240	1.000	0.680	0.400	1.000	0.080	0.040
Feature caetgories in best features	Battery	X		X	X		X				X		X	X		X
	CPU	X		X	X		X				X		X	X		X
	I/O Interrupts															
	Memory	X		X	X			X			X		X	X		X
	Network traff.	X		X	X		X				X		X	X		X
	Storage															
	Wifi															
	App CPU		X	X		X	X		X	X		X	X		X	X
	App memory		X	X		X	X		X	X		X	X		X	X
	App netw. traff.					X						X	X		X	X
	App process		X	X		X			X			X	X		X	X

TABLE 6.34: Feature categories included in best models of classifiers for malware version 9

The best performing model, with the least number of features, is the RF classifier with featureset 3. The features included in this model are shown in Table 6.35. This Table shows that 10 app features are useful for detecting malicious actions of a mobile Privilege escalation / Spyware malware type.

Features		
traffic_mobiletxbytes	dalvikprivatedirty_mor_app	stime_mor_app
battery_level	dalvikpss_mor_app	utime_mor_app
battery_voltage	otherpss_mor_app	
total_cpu	rss_mor_app	

TABLE 6.35: Features included in best RF featureset 3 model. (*ordered by feature ranking from top to bottom, left to right*)

6.2.10 Version 11 - DOS

An overview with key details malware version 11 is shown in Table 6.36. Malware version 11 was a music player, that sent UDP packets based on instructions from a remote server. When the remote server sent a special message, the malware application sent UDP packets as fast as possible for a whole minute to the client it was instructed to send to. The malware session type thus depended on the instructions from the remote server.

DOS	
Benign mode	Music Player
Malware type	DOS
Description	Either performs: 1) DDoS attacks on command, or 2) SMS botnet activities
Total nr. rows	655

TABLE 6.36: Overview malware version 11

Figure 6.18 shows the count of actions per actiontype. This Figure shows the differences between malicious actions and benign actions taken by malware type 11.

FIGURE 6.18: Distribution of actions of malware version 11 per actiontype

The performances of the classifiers on malware version 9 is shown in Table 6.37. This Table shows the following:

1. The best performing classifiers are RF with featureset 1, 2 or 3, and KNN with featureset 1, 2, or 3. The differences among these performances are not statistically significant.
2. All classifiers, except MLP, have a high f1 score, indicating that all classifiers, except MLP, are useful in detecting malware type 11.
3. KNN has a high f1 score with only one network traffic feature (*traffic_totaltxpackets*), suggesting that this feature is useful in detecting malware version 11.

	Classifier	Ada			RF			KNN			NB			MLP		
Best model	Featureset	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
	Best nr. features	25	5	27	19	11	21	1	6	6	14	37	35	33	15	16
	Best accuracy	0.977	0.977	0.977	0.985	0.985	0.985	0.985	0.985	0.985	0.962	0.962	0.977	0.870	0.870	0.870
	Best f1 score	0.909	0.903	0.909	0.938	0.938	0.938	0.938	0.938	0.938	0.828	0.828	0.903	0.000	0.000	0.000
	Best FPR	0.009	0.000	0.009	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	Best FNR	0.118	0.176	0.118	0.118	0.118	0.118	0.118	0.118	0.118	0.294	0.294	0.176	1.000	1.000	1.000
Feature caetgories in best features	Battery	X		X	X		X				X		X	X		X
	CPU	X			X						X		X	X		
	I/O Interrupts															
	Memory	X		X	X						X		X	X		
	Network traff.	X		X	X		X	X		X	X		X	X		X
	Storage															
	Wifi															
	App CPU		X	X		X	X		X			X	X		X	X
	App memory		X	X		X	X		X	X		X	X		X	X
	App netw. traff.		X	X		X	X		X	X		X	X		X	X
	App process		X	X		X	X		X	X		X	X		X	X

TABLE 6.37: Feature categories included in best models of classifiers for malware version 11

The best performing model, with the least number of features, is the KNN classifier with featureset 1. The features included in this model are shown in Table 6.38. This Table shows that one feature, *traffic_totaltxpackets* is useful in detecting malicious actions of a mobile DOS application. This feature describes the number of transmitted packets.

Features
traffic_totaltxpackets

TABLE 6.38: Features included in best KNN featureset 1 model. (*ordered by feature ranking from top to bottom, left to right*)

6.2.11 Comparison classifiers per malware type

The performance of the classifiers per malware type, i.e. training mode *permwtype*, is described in this Section. Table 6.39 shows an overview of the performance per classifier per malware type and featureset. Note here that the performances are shown for the best number of features and the best hyperparameter settings per classifier. The number of features is based on the RFCV method described in Section 5.4.
Table 6.39 shows the following:

1. RF has the highest f1 score for 5 out of 10 malware types.
2. KNN has the highest f1 score for 5 out of 10 malware types.
3. Ada has the highest f1 score for 4 out of 10 malware types [3].
4. Ada has a high f1 score (>0.7) for out 8 of 10 malware types.
5. RF has a high f1 score for 7 out of 10 malware types.
6. KNN has a high f1 score for 7 out of 10 malware types.
7. NB has a high f1 score for 2 out of 10 malware types.
8. MLP has a high f1 score for 5 out of 10 malware types.
9. All classifiers have a low f1 score for malware type 4 (Spyware SMS) and 6 (Adware).

[3] some performances are the same leading to a draw, hence to more than 10 highest f1 scores.

Mw type	AdaBoost				Random Forest				K-nearest Neighbour				Naïve Bayes				Multilayer perceptron				Test set
	acc.	f1	fpr	fnr	acc.	f1	fpr	fnr	acc.	f1	fpr	fnr	acc.	f1	fpr	fnr	acc.	f1	fpr	fnr	nr. rows
1	**0.947**	**0.704**	**0.025**	**0.321**	0.931	0.571	0.025	0.500	0.901	0.500	0.062	0.464	0.888	0.320	0.051	0.714	0.908	0.000	0.000	1.000	**304**
2	0.983	0.911	0.004	0.131	0.987	0.932	0.003	0.108	**0.989**	**0.945**	**0.003**	**0.077**	0.951	0.735	0.021	0.308	0.983	0.912	0.005	0.123	**1327**
3	**0.962**	**0.764**	**0.000**	**0.382**	**0.962**	**0.764**	**0.000**	**0.382**	0.956	0.727	0.003	0.412	0.898	0.000	0.003	1.000	0.901	0.000	0.000	1.000	**343**
4	**0.921**	**0.400**	**0.029**	**0.667**	**0.921**	**0.400**	**0.029**	**0.667**	**0.921**	**0.400**	**0.029**	**0.667**	**0.921**	**0.400**	**0.029**	**0.667**	**0.921**	**0.400**	**0.029**	**0.667**	**38**
5	0.984	0.897	0.005	0.143	0.986	0.911	0.008	0.089	0.987	**0.920**	0.008	0.071	0.958	0.688	0.008	0.429	0.984	0.903	0.009	0.089	**693**
6	0.906	0.186	0.017	0.880	0.901	0.270	0.030	0.796	**0.887**	**0.297**	**0.053**	**0.732**	0.908	0.027	0.004	0.986	0.911	0.000	0.000	1.000	**1588**
7	0.965	0.804	0.014	0.237	**0.973**	**0.843**	**0.006**	**0.227**	0.964	0.793	0.012	0.268	0.948	0.700	0.021	0.351	0.967	0.813	0.012	0.237	**1041**
8	**1.000**	**1.000**	**0.000**	**0.000**	0.971	0.800	0.000	0.333	0.971	0.857	0.032	0.000	0.941	0.500	0.000	0.667	0.971	0.800	0.000	0.333	**34**
9	0.985	0.913	0.000	0.160	**0.989**	**0.936**	**0.000**	**0.120**	0.974	0.844	0.004	0.240	0.941	0.652	0.025	0.400	0.985	0.923	0.012	0.040	**269**
11	0.977	0.909	0.009	0.118	**0.985**	**0.938**	**0.000**	**0.118**	**0.985**	**0.938**	**0.000**	**0.118**	0.977	0.903	0.000	0.176	0.870	0.000	0.000	1.000	**131**
$\bar{x}$	**0.963**	**0.749**	**0.010**	**0.304**	0.961	0.736	0.010	0.334	0.954	0.722	0.021	0.305	0.933	0.492	0.016	0.570	0.940	0.475	0.007	0.549	**568**
s	0.030	0.260	0.011	0.275	0.032	0.241	0.013	0.251	0.037	0.237	0.022	0.256	0.028	0.304	0.016	0.282	0.042	0.435	0.009	0.425	**531**

TABLE 6.39: Performance of different classifiers
(*f1 score < 0.5 = red, 0.5 =< f1 score =< 0.7 = orange, f1 score > 0.7 = green,*
***bold text** = best classifier for that particular malware type*)

The feature categories included in the featuresets of the best models of classifiers per malware type are shown in Table 6.40. This Table shows whether one or more features of a feature category is present in the best featuresets. This Table shows the following:

1. 6 out of 10 malware versions are best detected using only app features.
2. Malware version 4 (Spyware SMS) is best detected using only global features. However, note that the f1 score for Spyware SMS is the lowest of all malware versions. Hence, best refers to the best classifier, and not the actual performance.
3. Malware version 11 (DOS) is best detecting using only one global feature regarding network traffic.
4. Malware version 1 and 9 (Spyware contacts theft and Privilege Escalation / Hostile downloader) is best detected using both the app and global features.
5. None of the classifier includes I/O interrupts, Storage, or Wifi features for any of the malware versions.

Note that Table 6.40 shows the feature categories present in the model with the least number of features of the best performing model per malware type, i.e. for a given malware version. From all classifiers with similar performances as the best classifier of that malware version, the classifier with the least number of features is chosen. Similar refers to performance results that do not differ statistically significant. The classifier with the least number of features is chosen because this reflects best which features are relevant for detecting specific malware types.

	Mwtype	1	2	3	4	5	6	7	8	9	11
Best model	**Best classifier**	Ada	KNN	RF	RF	KNN	KNN	RF	Ada	RF	KNN
	Best featureset	fs3	fs2	fs2	fs1	fs2	fs2	fs2	fs3	fs3	fs1
	Best nr. features	49	8	5	28	7	11	9	14	10	1
	Best f1 score	0.70	0.94	0.73	0.50	0.95	0.35	0.86	1.00	0.94	0.94
Features categories in best featureset	**Battery**	X			X					X	
	CPU	X			X					X	
	I/O Interrupts										
	Memory	X			X						
	Network traffic	X			X					X	X
	Storage										
	Wifi										
	App CPU	X	X	X		X	X	X	X	X	
	App memory	X	X	X		X	X	X	X	X	
	App network traffic	X					X				
	App process	X	X	X					X		

TABLE 6.40: Feature categories included in classifiers with the least number of features per malware type.

Chapter 7

Usability

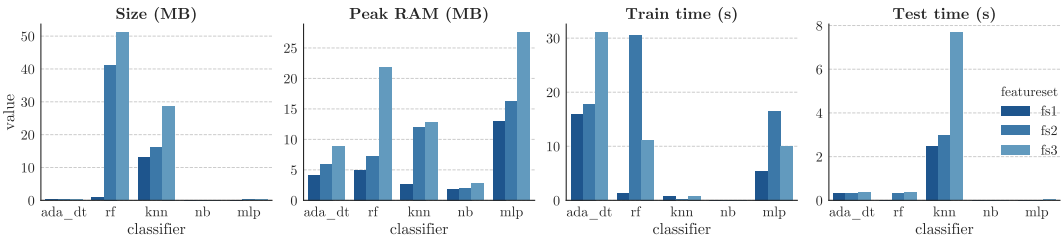
This Section analyses the usability of the classifiers on real devices. Usability in this Section refers to the extent in which the classifiers can be deployed on a real device. This Section does not examine whether the classifiers or detection methods are useful in detecting real-life malware on real devices; for this analysis see Section 8.2. The main factors influencing the usability of the classifiers are: i) model size, ii) train/test time, and iii) memory usage. Low metrics for these values reflect low storage requirements (model size) and computational requirements (train/test time and memory usage). In Section 7.1, we assume that models are deployed locally, i.e. the models are implemented on devices. Therefore, it requires lightweight methods regarding the aforementioned metrics. Additional design choices during the implementation of a classifier affect the usability. These are analysed in the last paragraph and include monitoring frequency, (re)training interval, and place of analysis.
Section 7.2 analyses the usability of self-made detection methods from a business perspective. This analysis examines multiple options for business regarding mobile security and the deployment of self-made detection methods.

7.1 Usability local deployment

The metrics of the best models of the classifiers are analysed on a PC with 2.3 G GHz, 2 cores, and 8GB RAM. The number of cores and RAM do not reflect the cores and RAM available on mobile devices. However the model metrics can be compared with each other by running them within the same environment. Consequently, any metric described in this Section does not describe actual model performance on a mobile device. These are solely used to estimate and compare the usability of the described detection models.

The model size, train/test time, and memory usage for training mode *all* and testing mode *normal holdout* are shown in Figure 7.1. When a model has relative low values for any of the metrics in comparison to other models we call it a lightweight model, hence requiring less resources. In this Section, the differences in resource consumptions are described and some suggestions are provided to reduce the resource consumptions of models. A deeper analysis might be performed on the optimization of resource consumption by altering the number of features, although for sake of simplicity and time we will keep these numbers fixed.

Lastly, in this Section we focus on the training mode *all*.



Metrics shown for classifiers with the following hyperparameter settings for respectively (fs1, fs3, fs3): Ada: n_estimators = (400, 400, 400), RF: n_estimators (5, 320, 320), max_depth = (320, 40, 20), max_features = (24, 10, 3), KNN: k = (1, 1, 1), NB: alpha = 0.5, 1.6, 0.4, MLP: nr_layers = (5, 10, 5), nr_nodes_per_layer = (25, 25, 15)

FIGURE 7.1: Resource metrics per classifier model for training mode *all* and testing mode *normal holdout*

Random Forest

The best performing RF classifiers are RF with featureset 2 (number of features: 29) and RF with featureset 3 (number of features 10). Figure 7.1 shows that the RF models have a relative large model size of >40 MB. The training time of RF is also relative high with >30 seconds for featureset 2 and >12 seconds for featureset 3.

The hyperparameters can be adjusted in order to decrease the space and time needed for the RF model. The hyperparameter analysis, presented in Section 6.1.1 suggests that the number of estimators has a small influence on the performance of RF. As each estimator represents a decision tree, reducing the number of estimators is expected to reduce the space and time requirement of the RF model. In order to quantify this, additional experiments were run with different numbers of estimators for featureset 2. These experiments keep the number of features, the maximum depth, and maximum features fixed. The impact of the number of features on the metrics is shown in Figure 7.2. This Figure shows that similar performances are achieved with less number of trees. This finding suggests that the RF classifier can be optimized to use less space and computational resources without loss of performance.

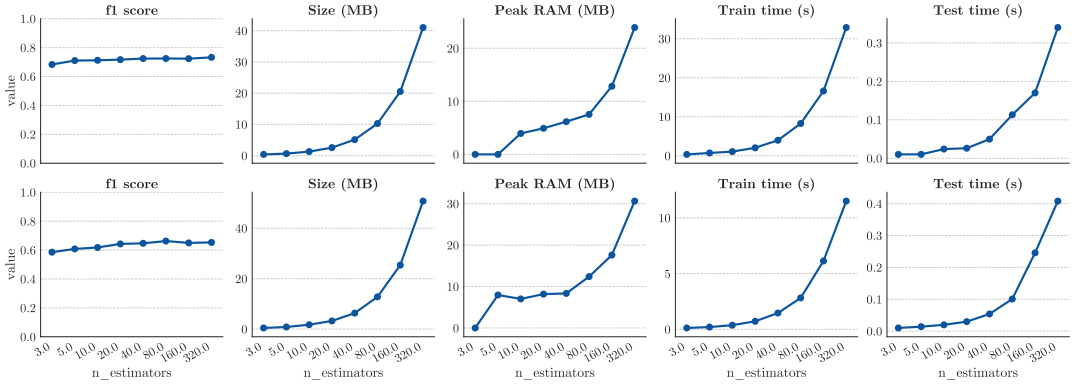


FIGURE 7.2: Influence of nr of estimators on resource metrics and f1 score. Top row = featureset 2, bottom row = featureset 3

K-nearest neighbour

The best performing KNN classifiers are KNN with featureset 2 (nr. of features: 13) and KNN with featureset 3 (nr. of features 9).
Figure 7.1 shows that the best performing KNN classifiers have a relatively large model size of >12 MB. The training time of KNN is relatively small compared to the other three classifiers. This can be explained by the fact that the model of KNN consists of: i) a search tree (KDTree) to search for the nearest neighbour in the training set, and ii) all training instances of the training set. This leads to a relatively large model size. Additionally, during testing, the nearest neighbour must be searched for in the complete training set leading to relative long testing times. An increase in the number of training instances will, therefore, coincide with a decrease in the usability of the KNN model.

To decrease the space and computational power needed for the KNN model, the hyperparameters cannot be adjusted. The number of neighbours to consider has the smallest value of 1 for all featuresets. The nearest neighbour computation algorithm can be adjusted to reduce the resource requirements of the KNN model. The influence of this algorithm has not been examined due to time constraints.

AdaBoost

The best performing Ada classifiers are Ada with featureset 2 (nr. of features 25) and Ada with featureset 3 (nr. of features 43). Figure 7.1 shows that the best performing Ada classifiers have relative small model sizes of <1 MB. Ada and RF are both classifiers that use decision trees, hence the relative small size of Ada suggest that it uses less complex decision trees than RF. The training time of Ada is similar to the training times of RF. This suggests that although the decision trees in Ada are smaller, the complexity of the boosting algorithm leads to higher training times. The testing times of Ada are relatively low compared to other classifiers and may be explained by the creation of small decision trees after boosting.

The hyperparameters can be adjusted to decrease the space and computational power needed for the AdaBoost model. However, the hyperparameter analyses, presented in Section

6.1.5 suggests that the number of estimators has a large influence on the performance of Ada. Reducing the number of estimators may reduce the resource consumption of the model, although at the expense of model performance. Therefore, no further analysis is performed on the reduction of resource consumption of AdaBoost.

NB and MLP

Figure 7.1 shows that the NB models are relatively small and have relative small training and testing times. NB and MLP are excluded from further analysis due to low performance.

Training *permwtype*

Figure 7.3 shows the resource metrics, summed over all 10 malware types, for training mode *permwtype* and testing mode *normal holdout*. For *Peak RAM(MB)* the average is shown instead of the sum. Comparing Figure 7.1 with Figure 7.3, we see that the total training and testing times for all models increase when 10 separate models are trained for each malware type, in comparison to when deploying one model for all 10 malware types. This suggests that deploying separate models per malware type decreases the usability of the detection method. This problem increases when more than 10 malware types are analysed and included in a detection method. Hence, detection methods with separate models per malware type are most likely unusable on real devices.

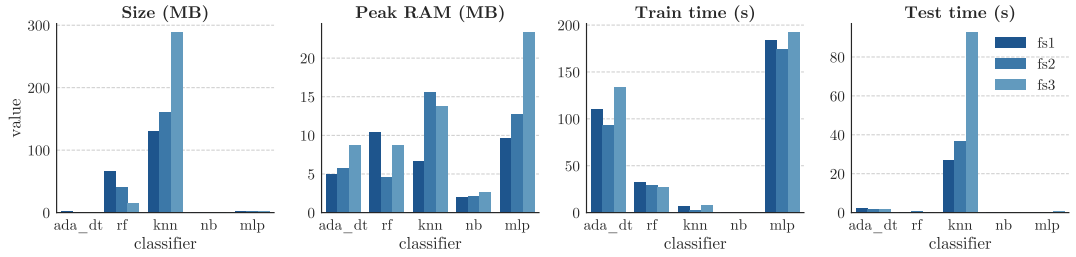


FIGURE 7.3: Resource metrics per classifier model summed over all malware types, for training mode *permwtype* and testing mode *normal holdout*. (Note: Peak RAM (MB) is shown as an average)

Other factors influencing usability

Other important factors influencing the usability of the detection methods are: i) place of analysis/ identification, ii) granularity of detection, iii) (re)training frequency, and iv) monitoring frequency.

In the analysis of the previous Section, we assume that models are deployed locally. However, the place of analysis and identification (as described in Section 2.4), can also be in the cloud. In that case, the training of models (analysis), and/or the detection of malware (identification) can take place in the cloud. This alleviates devices from resource-intensive computations, which are described in the previous Section.

The granularity of detection, i.e. per-app data monitoring or global device data monitoring, also influence the usability of the models. Featureset 1, throughout this research, examined the device globally, and featureset 2 examined the device per-app. Monitoring a device per-app requires more data retrieval than monitoring the device globally. Therefore, methods that monitor a device globally are more lightweight. This issue is described in more detail in Section 8.2.

Lastly, the monitoring frequency of features influences the usability of models. Continuous monitoring of features allows the real-time detection of malware. However, continuous monitoring is resource intensive. To increase the usability of models, an optimum threshold between timely detection and resource requirements must be made. This research's dataset contained data that was monitored at a 5 second time interval, so no analysis could be performed on this trade-off.

7.2 Cost-benefit analysis

As described in Section 2.3, mobile malware is an issue of increasing importance for businesses. This Section attempts to quantify this issue by examining the costs and benefits associated with mobile security. Three options are calculated for dealing with mobile security in businesses. These options are: i) doing nothing about mobile security, ii) developing in-house mobile security solutions, and iii) outsourcing mobile security. Other options such as the training of employees on mobile security, setting rules and policies regarding mobile device use, encrypting devices etc[1] are excluded from analysis.
The three options are simplified by excluding business factors such as the company's size, market, sector, strategy etc. All calculations below are assuming an average business of 250 employees, in which all employees in a business are using a mobile device for business purposes, e.g. calling, mailing, agenda, etc.

A recent whitepaper on mobile security [17] describes the current average situation of mobile security in businesses and the costs associated with mobile cyber attacks. This recent white paper is used as a starting point for the calculations of the cost and benefits of the three options. Note here that we only take into consideration the costs, as we assume that the benefits are avoidance of costs.

The next Sections describe the assumptions and calculations per option. Section 7.2.1 first describes the current average situation. The analysis of the current situation provides us with numbers on defence costs, direct costs, and indirect costs. Defence costs, in our calculations, are any costs associated with securing mobile devices. Direct costs and indirect costs are described in Section 7.2.1.

7.2.1 Average current situation

The numbers for calculating the current average situation are taken from a recent white paper called 'The economic risk of confidential data on mobile devices in the workplace' [17] by the Ponemon institute[2]. The findings of the white paper are based on a survey amongst 588 IT and IT security professionals employed in Global 2000 companies. The following numbers are drawn for the white paper:

1. 3% (1,723 devices) of the employees' mobile devices are believed to be infected with malware at any point in time" (the average company in the survey had 83,844 devices).
2. 26% of the infected devices are investigated (on average 448 devices).
3. The average costs per mobile malware attack is $9,485.

In the white paper above, the average costs per mobile malware attack are calculated as follows: first, the average total direct costs ($3,530,240) and indirect costs ($12,812,017), associated with the investigation of the 448 devices are summed. Then the sum is divided by the total 1,723 devices infected.

In the white paper, the average total direct costs are calculated by summing the following costs: i) IT helpdesk including replacement ii) diminished productivity or idle time, and iii) IT security support including investigation and forensics. These costs are extrapolated from the estimations given in the survey responds. The costs per infected device is shown in Figure 7.4.

[1] For additional options, refer to the literature on managing cybersecurity risks such as [71] and [72].

[2] The Ponemon institute refers to itself as "an independent research and education institute focused on issues affecting the management and security of sensitive information about people and organizations"

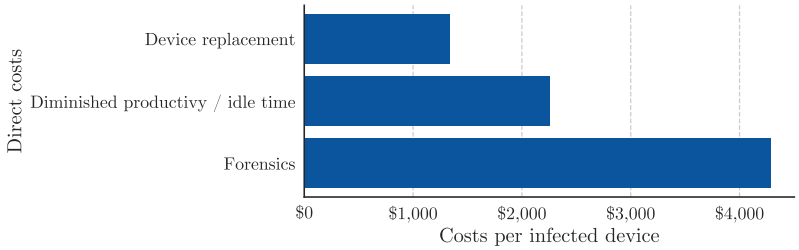


FIGURE 7.4: Direct costs per device associated with a mobile cyberattack

In the white paper, the average total indirect costs are calculated taking the potential maximum loss (PML), associated with a mobile cybersecurity attack, of the following costs: i) cost of data loss or breach ii) cost of non-compliance iii) lost or diminished reputation. The PMLs, in the white paper, are based on estimations given in the answers of a survey question that asked for an estimation of the aforementioned costs of a worst-case scenario. The PML is multiplied in the white paper by the perceived likelihood of occurrence (PLO), of the worst case scenario, per annum. The PLO is estimated at 9.8% and was extrapolated from the replies on the surveys. The multiplication of PML by PLO results in an estimated of $12,812,018 of indirect costs. Note here that the indirect costs are not related to the number of mobile malware attacks. Lastly, the average annual budget for mobile security is estimated at $4,368,000 in the white paper.

From the above calculations, we will use the following numbers for our calculations in the next Sections:

1. The average mobile devices being infected at any point in time is 3%.
2. The direct costs per mobile malware attack is $7,880.
3. The average likelihood of a worst-case scenario occurring is 9.8%.
4. The average annual budget for mobile security per employee is $52 ($4,368,000 average annual mobile security budget / 53,844 average employees in companies of the survey). Note here that in the white paper, only the total average annual mobile security budget is given. We calculate a per employee number ourself.
5. The average indirect costs, in a worst-case scenario, associated with at least one mobile malware attack is 2,428 dollars per employee ($130,734,870 / 53,844 employees[3]. Note here that in the white paper, only the total indirect costs are given. We calculate a per employee number ourself.

The average defence costs, i.e. costs associated with securing mobile devices, the direct costs, and indirect costs are shown in Table 7.1 per company size. The company sizes are shown for small, medium, and large-sized companies (respectively with less than 50, between 50 and 250, and more than 250 employees). The employee numbers per size are based on the taxonomy of the European Commission on internal market, industry, entrepreneurship and SMEs [73]

1. Total defence costs = $52 * number of employees
2. Total direct costs = (3% of the number of devices infected) * 7.880 direct costs per device
3. Total indirect costs = $2,428 * number of employees * 9.8% likelihood

Company (size)	Employees	Total defense costs ($)	Total direct costs ($)	Total indirect costs ($)	Total costs ($)
Small	<50	< 4,056	< 12,608	< 11,897	< **28,562**
Medium	50	4,056	12,608	11,897	**28,562**
	-	-	-	-	-
	250	20,281	63,040	59,487	**142,808**
Large	>250	> 20,281	> 63,040	> 59,487	> **142,808**
Avg. survey	53,844	4,368,000	13,577,303	12,812,017	**30,757,320**

TABLE 7.1: Cost overview current average situation

[3]For simplicity we assume that the number of employees is equal to the number of mobile devices used in a business

7.2.2 Option 1 - Do nothing

The first option consists of doing nothing with mobile security. The costs associated with this are shown in Table 7.2. The costs are calculated as follows:

1. Total defence costs = $0 * number of employees
2. Total direct costs = (30% of the number of devices infected) * 7,880 direct costs per device
3. Total indirect costs = $2,428 * number of employees * 18,6% likelihood

The defence costs are zero because this option implies doing nothing about mobile security. For the direct costs, we assume that the number of infected devices rises to 30%. This number is guesstimated, based on the high prevalence of mobile malware as described in Section 2.3. We assume that the that the likelihood of a worst-case scenario doubles when nothing is done about mobile security. This assumption is again based on the high prevalence of mobile malware.

Company (size)	Employees	Total defense costs ($)	Total direct costs ($)	Total indirect costs ($)	Total costs ($)
Small	<50	-	< 118,200	< 23,795	< **141,995**
Medium	50	-	118,200	23,795	**141,995**
	-		-	-	-
	250		591,000	118,973	**709,973**
Large	>250	-	> 591,000	> 118,973	> **709,973**
Avg. survey	53,844	-	127,287,216	25,624,035	**152,911,251**

TABLE 7.2: Cost overview option 1, doing nothing

7.2.3 Option 2 - In-house development

The second option consists of developing mobile security solutions in-house. The costs associated with this option are shown in Table 7.3. The costs are calculated as follows:

1. Total defense costs = 50% of investment in mobile security solution for maintenance.
2. Total direct costs = (15% of number of devices infected) * 7,880 direct costs per device
3. Total indirect costs = $2,428 * number of employees * 9.8% likelihood

The defence cost consists of maintaining an in-house security solution. We assume that the maintenance costs are 50% of the total investments costs of the in-house security solution. A rule of thumb for maintenance costs of mobile applications is 20% of the investment costs of creating the mobile application [74][75][76]. Given that a mobile security application is more complex than a normal application, we double this percentage to 40%. We add 10% additional maintenance costs for any server requirements to manage the devices remotely.

The investments costs are calculated by assuming that a team of 7 cybersecurity specialists are required to work 6 months on developing the mobile security solution. Given an average estimated salary of $58,08 dollar per hour [77], this results in $382,183 costs for the salary of the specialists. Lastly, we add 50% of the salary costs for any additional expenses such as dataset purchase, distributed data solutions, and implementation costs. The salary costs + additional expenses result in a total of $573,274 investments costs. Maintenance costs are thus estimated at $286,637 per year.

For the direct costs, the percentage of devices infected is estimated to be 15%. This number is based on the observation that most literature work described in Section 2.5.1 was able to create a detection model with a TPR of 0.85, thus an FNR (number of undetected malware) of 15%. We, therefore, assume that a group of 7 cybersecurity specialist achieve the same FNR in 6 months.

For the indirect costs, the likelihood of a worst-case scenario is assumed to be the same as the average solution. The percentage of devices is lower than doing nothing, but 15% of the malware is assumed to go undetected. Therefore we keep the likelihood the same as the current average situation.

Company (size)	Employees	Total defense costs ($)	Total direct costs ($)	Total indirect costs ($)	Total costs ($)
Small	<50	< 286,637	< 59,100	< 5,949	< **351,686**
Medium	50	286,637	59,100	5,949	**351,686**
	-	-	-	-	-
	250	404,562	295,500	29,743	**729,805**
Large	>250	> 404,562	> 295,500	> 29,743	> **729,805**
Avg. survey	53,844	798,200	63,643,608	6,406,009	**70,847,817**

TABLE 7.3: Cost overview option 2, in-house development mobile security solutions

7.2.4 Option 3 - Outsource

The costs associated with this are shown in Table 7.4. The costs are calculated as follows:

1. Total defense costs = $203 * number of employees
2. Total direct costs = (0.015% of number of devices infected) * 7,880 direct costs per device
3. Total indirect costs = $2,428 * number of employees * 4.5% likelihood

The defence costs are based on the yearly cost per device of a popular mobile device management service (MDM) [4]. With MDM, businesses can outsource their mobile security. These services provide software to monitor, manage, and securing employees' mobile devices [78]. To calculate the direct costs, we assume that the number of devices infected is at least 50% less than the current average situation. This number is based on the observation that 40% of the companies in the above white paper, use some kind of MDM. We assume that most malware attacks occur due to the lack of MDM. Therefore, we assume that if 100% of the companies used some kind of MDM, the number of infected devices would be at least 50% less. Using the same logic, the likelihood of the worst-case scenario is excepted to be at least 50% less than the current average situation.

Company (size)	Employees	Total defense costs ($)	Total direct costs ($)	Total indirect costs ($)	Total costs ($)
Small	<50	< 10,140	< 3,940	< 5,949	< **20,029**
Medium	50	10,140	3,940	5,949	**20,029**
	-	-	-	-	-
	250	50,700	19,700	29,743	**100,143**
Large	>250	> 50,700	> 19,700	> 29,743	> **100,143**
Avg. survey	53,844	10,919,563	4,242,907	6,406,009	**21,568,479**

TABLE 7.4: Cost overview option 3, outsource mobile security

7.2.5 *Conluding remarks*

Figure 7.5 shows an overview of the findings of the cost-benefit analysis. This Figure shows the costs associated with the current average situation and the three options described above. Note again that these numbers are shown for an average company of 250 employees and specifics of the sector, market, strategy etc. of the company are not included in the analysis.

Based on our cost-benefit analysis, in-house development of mobile security solutions has the highest cost compared to the other options. These costs are mainly due to the high defence costs coming from the maintenance of the mobile security solutions. Option 2 has similar high costs as option 1, due to the high direct and indirect costs from the increased likelihood of a mobile cyberattack. Lastly, option 3 has the lowest costs due to the relatively low defence costs (compared to maintaining an own mobile security solution), and the decreased likelihood of a mobile cyberattack.
Which option to choose depends on numerous factors such as:

1. Company size
2. Company sector/market

[4] https://www.air-watch.com/pricing

3. Confidentiality of information shared within business
4. Knowledge of employees on cybersecurity etc.

Based on our example of 250 employees, our analysis shows that outsourcing mobile security has the lowest total costs. In the cost-benefit analysis, we assumed an average company without taking into consideration the many factors such as a company's size and market. Each specific company should therefore carefully consider their decision based on these many factors.

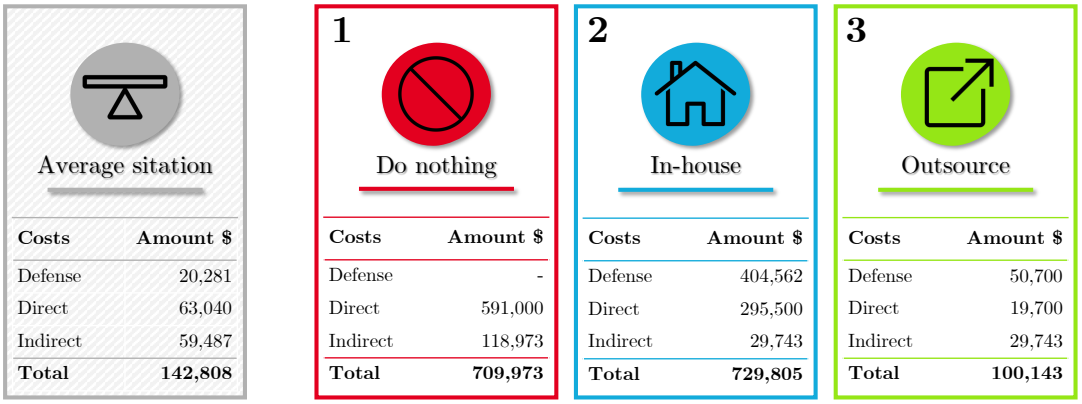


FIGURE 7.5: Costs-benefit analysis overview for a company with 250 employees

Chapter 8

Discussion

This Chapter discusses the main findings of the Results presented in Chapter 6 and this research's limitations.

8.1 Results discussion

This Section discusses the main findings of Chapter 6. The findings are reported as following i) classifier performance and ii) important features.

8.1.1 Classifier performance

High performance RF for training mode *all*
The best RF model with training mode *all* is RF with featureset 2, containing 29 app features, and showed an f1 score of 0.730, an FPR of 0.013, and an FNR of 0.346 (i.e. TPR of 0.654). Similar results, i.e. results with no statistically significant difference to the best RF model, are achieved with featureset 3 that showed an f1 score of 0.722, an FPR of 0.009, and an FNR of 0.380 (i.e. TPR of 0.620). A TPR of 0.62 is relatively low compared to other studies on the dynamic detecting of mobile malware, which showed TPRs between 0.61 and 1 (see Section 2.5.1 for a complete comparison).

In contrast to [24][58][60], with TPRs between 0.82 and 0.97, our study used data from real-life users instead of virtual environments for the training and testing of detection methods. The use of real-life data for the training of models may have led to lower performance due to the additional noise included in our dataset (see paragraph *Overall high FNR*). However, performances are hard to compare as [24][58][60] reflect performance results of their detection methods under lab-like environments, i.e. with the use virtual environments, and our performances reflect performances of detection methods under real-life circumstances.

Other studies, using real devices for training and testing of models, based their performance on apps running isolated for 10 minutes [24], device in idle state [38] or unknown circumstances [54][23]. One study that used real devices under real-life circumstances for the assessment of their detection method is [57]. In that study, the researchers created a multi-level framework (called MADAM) that showed high performance (TPR 0.97, FPR 0.005). MADAM requires root permissions as it used System Calls for the detection of malware. In contrast to MADAM, this present study's detection methods do not require any root permissions. Furthermore, MADAM used both dynamic and static features, in contrast to this present study that used only dynamic features. Lastly, MADAM is a detection method consisting of multiple architectural blocks that monitor different aspects of the device, in contrast to this present study that does not consist of a complex architecture.

High performance RF, KNN, Ada for training mode *permwtype*
In this research, RF and KNN, showed good performances (f1 scores on average above 0.7) when separate models were trained per malware versions (training mode *permwtype*). These findings are in line with earlier studies that found good performances for RF [26][54][60] and KNN [57] on the detection of mobile malware. The performance of Ada was not found in earlier studies.

MLP overall varying performance
In this research, MLP did not show good performances in training mode *all*. Additionally, it showed good performance (f1 score >0.7) for 50% of the malware versions, and bad performances (f1 score <0.5) for the remaining 50% of the malware version. Therefore, MLP had a varying performance and may be less useful in detecting mobile malware. This does not necessarily imply that Neural Networks, in general, are useless in detecting mobile malware. This present study only examined MLP with a limited search space. Tuning MLP or using more advanced Neural Networks may improve the performance.

NB overall relative low performance
In this research, NB showed overall low performance (f1 score <0.5). Other studies [53][24] showed good performances for NB, hence the results of this present study are inconsistent with earlier studies. This may be due to the additional noise included in our dataset, although further research is needed to examine the cause.

Similar results for Ada with different testing modes
All classifiers, except Ada, show worse performances when tested on new devices (testing mode *unknown device*), than if tested on the same devices (testing mode *normal holdout*). Most studies do not provide information on the testing mode, thus testing modes of other research cannot be compared to our study. Our results suggest that Ada may be more suitable when its purpose is to first build a detection method with a subset of users and subsequently use this detection method to protect other users. This approach is more scalable since new users can be protected with an already existing detection method. Otherwise, the model needs to be retrained constantly for every new user. Ada with featureset 2 and testing mode *unknown device* had an f1 score of 0.583 and an FPR of 0.013 and an FNR of 0.528. The relatively low performance of Ada calls for more research on improving models to detect malicious actions of new devices, given the larger scalability of these models. Some suggestions are given in Section 9.2.

Low performance on Spyware SMS and Adware
Performances on the detection of malware version 4 (Spyware SMS) and 6 (Adware) were low (f1 score < 0.5). The dataset used for training and testing contained 190 records for malware version 4. The low performance on the detection of Spyware SMS may, therefore, be due to the small number of training instances. The training and testing dataset contained 7940 records for malware version 6. Therefore, the low performance is most likely not a result of insufficient training instances. More research is needed to find the cause for the low performance of the classifiers on malware version 6.

Perfect score on Ransomware
Ada showed a perfect score for malware version 8 (Ransomware). The dataset used for training and testing contained 170 records for malware version 8. Ada did not show a low performance although it was trained on a low number of training instances. The perfect score on the Ransomware is hard to explain. Given the low number of testing instances, it is hard to estimate whether this performance is the same on larger test sets.

FNR higher than FPR
Ín this research, the FNR (undetected malicious actions) was overall higher than the FPR (benign actions labelled as malicious) . In other studies that we examined during our literature research, the FPR is overall higher than the FNR. This may be due to the difference in the distribution of malicious and benign datapoints. In our research, the dataset contained 90% benign datapoints and 10% malicious data points. As most models are biased towards the majority class, the FNR is expected to be higher than the FPR in our case. In other studies the distribution of malicious and benign datapoints used are often 50/50 or a majority of malicious datapoints, resulting in equal FPRs and FNRs, or higher FPRs than FNRs. This stresses the importance of presenting both FPR and FNR values in research, which are not presented in all examined studies.

Overall high FNR
In this research, all detection methods showed relatively high FNR (f1 score > 0.3). A high

FNR implies a high number of malicious actions are undetected. This high FNR may have different causes. Many features in the featureset are influenced by many factors, as the features describes devices that are running multiple applications simultaneously. For instance, the priority, the CPU allocation, and the memory allocation of the Malware app depend on other applications running parallel to it. Therefore, the features in the dataset do not solely reflect the (type of) action of the Malware app, but also the device's state at a given moment. This may result in excessive noise for the classifiers to accurately detect malicious actions of the Malware app. Another possible cause may be the due to similar influences of malicious actions and benign actions on the analysed features. Therefore, the classifiers may not be able to distinguish sufficiently between malicious actions and benign actions.

8.1.2 Important features

Global features relative low performance
In this research, all models with App features showed an overall high performance (>0.7 f1 scores) compared to Global features (<0.4 f1 scores). Therefore, only examining a device globally does not result in good detection performance. Other studies with features similar to our Global features [24][26][54][23][60] found high performances (TPR > 0.8) with Global features. All these studies used virtual environments instead of real devices. Additionally, the apps in [24][26][60] were run isolated for a maximum of 90 seconds. Therefore, our data contains more noise compared to the datasets used in [24][26][60]. The difference in performance results may be due to this additional noise. This suggests that the performance results of the research that used Global features may be too optimistic. The remaining studies examined in our research did not provide a clear description on whether Global or App features were used, so no comparison can be made.

No improvement of performance by combining App features and Global features
In this research, all models with a Combined featureset (App features and Global features) had similar results, i.e. no statistically significant difference, as the models with only App features. This suggests that Global features add little value to detection methods. This may be because Global features are influenced by many factors of the device at a given moment, e.g. running applications or processes. This may suggest that Global features do not capture the behaviour of the Malware app sufficiently.

App CPU, App Memory, and App Process important features
The RF model with featureset 3 and training mode *all* contained CPU, Memory, and Process features, suggesting that these feature categories are important in detecting malicious actions of malware. The same three categories were found in 7 out of the 10 best models for training mode *permwtype*.

8.2 Limitations

The limitations of this study are described below. They are divided as limitations imposed by the i) dataset, ii) detection method, and iii) statistical analysis.

8.2.1 Dataset

All detection methods in this research are created using the same dataset. The limitations caused by using this particular dataset are listed below:

1. All devices in the dataset are Samsungs Galaxy S5. Therefore, it is unknown how the models perform when used on other devices.
2. All malware types in the dataset were written for the purpose of this research. This limits the findings of the research due to two reasons. First of all the self-written malware was adjusted for this research, As described in Section 3.2.1, before the malware probe sent any data to a server, the data was scrambled to ensure the privacy of the volunteers

of the research. It is possible that this scrambling influenced the features analysed in the research. As a result, it is possible that this research's detection models are biased towards detecting scrambling actions, limiting its efficacy on real wild malware. Secondly, ii) it is unknown whether real malware executes the same way as the self-written malware. Although the behaviour of the self-written malware is based on wild malware types, the implementation of the self-written malware may differ from real wild malware. It is therefore unknown whether similar detection performances are achieved on wild malware.

3. The dataset contained a low number of data points for malware version 4 and 6. This limits the conclusions drawn from these malware version's results.

8.2.2 Detection method

The detection methods of this research share characteristics that may limit their performance or applicability. The limitations imposed by this is listed below:

1. All detection methods in this research are signature-based detection methods. As described in Section 2.4.1, signature-based models identify malware based on signatures, e.g. a pattern of behaviour. In this research, the signature of a malicious action is a record of feature values for some given featureset. This signature may be different for other (wild) malware types. Therefore it is unknown whether similar performances are achieved on other (wild) malware types or other versions of the same malware type.
2. All detection methods in this research use RFCV as a feature selection method. This method allows for feature reduction, but RFCV is a greedy search strategy. Therefore any solutions for the optimal number of features found are sub-optimal, as local optimums are found with RFCV.
3. All detection methods using App features require data collection of all applications running on an application. In our research, we discarded data from other applications as we knew which application was malicious. However, in real-life circumstances, the malware application is unknown. Therefore, although App features show better performance than Global features, they are more resource intensive than Global features as all applications need to be monitored. A suggestion for this issue is described in 9.2

8.2.3 Statistical analysis

All models in this research are tested once using cross-validation with a hold-out method. The statistical significance of differences between models is analysed using a McNemar test ($\alpha < 0.05$). This test is suitable for comparing machine learning performances when comparing test set performances [70] and has a low Type I error (false positive errors). However, as [70] noted this statistical analysis is limited by two issues. Both issues arise because only the test set is taken into consideration by the statistical test. This results in the following issues: i) any variability in the training set is not accounted for, and ii) we must assume that differences observed in the test set are similar to the differences observed in the training set.

Chapter 9

Conclusion

Section 9.1 concludes this research by providing the answers to the this research's research questions. Lastly Section 9.2 provides suggestions for future research to improve the performances of dynamic detection methods of mobile malware. Additionally, suggestions are provided to overcome this research's limitations.

9.1 Conclusion

This Section summarizes the answers to the sub-questions and main research question proposed in Section 1.1.

S.Q. 1 *How do different machine learning techniques such as Random Forest, K-Nearest Neighbour, Naïve Bayes, and Multilayer Perceptrons, perform in detecting Mobile Trojans?*

The performances of the AdaBoost, Random Forest, K-Nearest Neighbour, Naïve Bayes, and Multilayer Perceptrons are examined and described in Section 6.1. The results of this research show that the Random Forest classifier is most suitable for detecting Mobile Trojans when one model is used that is trained on multiple subtypes of Mobile Trojans. This classifier achieves an f1-score of 0.73 with an FPR of 0.009 and an FNR of 0.380. Random Forest, AdaBoost and K-nearest-neighbour show high performances when separate models are trained on each subtype of Mobile Trojan with an average f1-score of >0.72, FPR of <0.02 and FNR of <0.33. The remaining classifiers (Naïve Bayes and Multilayer Perceptron) showed relatively low performance overall.

S.Q. 2 *What software and/or hardware features, that do not require root permissions, are the most crucial for the detection of Mobile Trojans?*

Multiple featuresets are examined in this research, making a distinction between global device features and features related to an application. When using one model for the detection of multiple subtypes of Mobile Trojans, 10 app features related to the memory usage, process information, and CPU usage of the app, are sufficient for detecting malicious actions. Additionally, app features in general showed the best performance for detecting 7 out of 10 malware versions, compared to global device features.

S.Q. 3 *What is the usability of these different classifiers on a real device?*

The usability of the different classifiers is examined in Section 7. This Section showed that the best performing model, RF with featureset 2 is relatively large and has long train and test times. However, this Section showed that the usability of the RF model can be optimized by adjusting hyperparameters, e.g. number of estimators, without loss of performance. The KNN classifier is less scalable than other classifiers when the training set increases in size. Lastly, this Section showed that the usability depends on design choices during the implementation of these classifiers. The models can be implemented in the cloud, alleviating the mobile device of resource-intensive tasks compared to local deployment. Other design options, such as the (re)training interval and monitoring frequency, affecting the usability of the detection methods are also described in Section 7.

M.Q. 1 *How can we improve the dynamic detection of Mobile Trojans using hardware and software features (not requiring any root permissions), based on real-life data?*

Using the answers to the sub-questions, the main research question can be answered as follows. The dynamic detection of Mobile Trojans can be improved by:

1. focusing on AdaBoost, Random Forest or K-nearest neighbour classifiers,
2. with features related to applications, instead of global device features,
3. and by carefully considering different design choices during the implementation of the detection methods to optimize usability on real devices.

Additionally this research provided:

1. valuable knowledge on which features are important to detect different subtypes of Mobile Trojans,
2. an extensive analysis of the influence of hyperparameters on the performance of multiple classifiers,
3. and a global cost-benefit analysis of multiple mobile security options for businesses.

All findings of this research are based on real-life data. Therefore all assessment of detection methods is shown for real-life circumstances, increasing this research's contribution to practice.

9.2 Future work

Future research is necessary to i) improve the performances of this research's detection methods and ii) overcome this research's limitations. Some suggestions are provided below:

1. Future research can focus on improving False Negative Rates as this research's detection methods show relatively high FNR. As Random Forest and AdaBoost showed relative high performance, other (boosted) ensemble classifiers can be examined such as GradientBoosting [79].
2. More research is needed to improve the performance of detection methods on new devices, i.e. devices not included in the training set, as this increases the scalability of detection methods. Detection methods may be improved by dividing users into separate groups during cross-validation and to exclude (multiple) group(s) from the validation set. This can help the tuning of classifiers to detect new users since hyperparameters are selected for their performance on new users.
3. Future research can improve dynamic detection methods by analysing real wild malware samples to improve the efficacy of the methods on the detection of real malware in order to overcome the limitation imposed by the use self-written malware.
4. Further research is encouraged to increase the sample size to improve the efficacy and applicability of the methods. This research analysed 10 different samples of Mobile Trojans.
5. Future research can examine whether time series analysis may improve the detection methods. This research's detection methods analyse dynamic features by individually assessing these values, without considering these value prior in time. Therefore, the absolute values of features are analysed. With time series analysis, relative values can be used which may improve the detection methods, as is suggested by the results of [58].
6. Future research is encouraged to optimize the resource requirements of detection methods using App features. As described in Section 8, monitoring App features of all applications installed on a device is resource intensive. Research to decrease these resource requirements is needed. A suggestion may be to only monitor applications that require sensitive permissions, e.g. access to SD card or access to contacts, as was implemented in the framework of [57].

Appendices

Appendix A

System preprocessing

The dataset was preprocessed to allow the reading and analysis of the different probes. To read the dataset, PySpark requires a data schema with column names and column data type. The data schemes were provided with the dataset, however the data schema was not correct for all rows of the T4 probe. Therefore an additional preprocessing was needed for the T4 probe. The preprocessing steps are shown in Figure XX. This figure shows that in order to get the correct dataset *t4df_final*, the method *read_t4df* is called four times on four different combinations of input files. The *read_t4df* method uses two data schemas and the T4 probe data as input. One data schema is the data schema as provided with the dataset. The other data schema was found as a header in the csv file of quarter 4 2016. The *read_t4df* method then tries to first read the complete csv file with data schema 1, and drops all rows not conforming to this data schema. The resulting dataframe is called *part1_df*. Then it tries to read the complete csv files with the data schema 2, and again drops all rows not conforming to this dataschema. This resulting dataframe is called *part2_df*. The two partial dataframes (*part1_df & part2_df*) are then joined together and returned by the method *read_t4df*. The *read_t4df* method is called with all four csv files and these files are joined together to create the final T4 dataframe. This dataframe is exported as a csv file and used throughout the research as the T4 probe dataset.

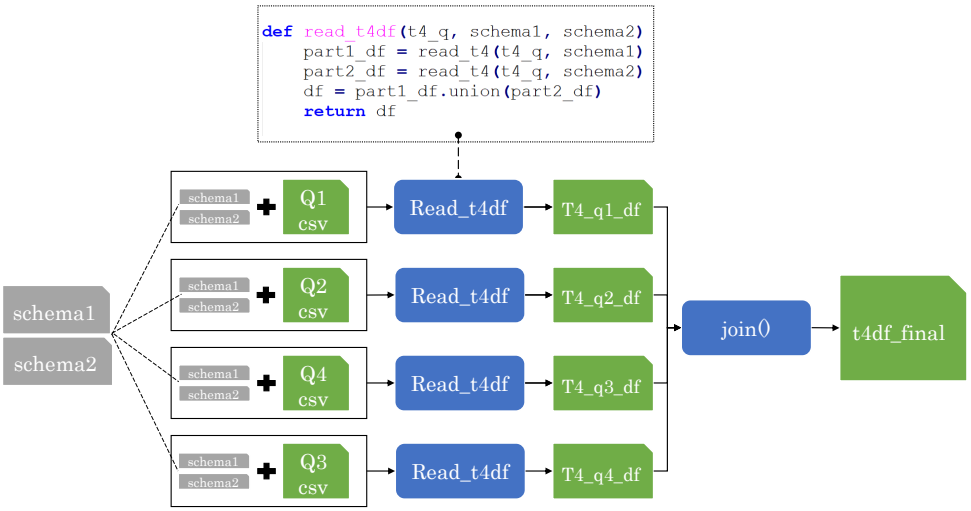


FIGURE A.1: T4 preprocessing steps

Appendix B

Literature review method

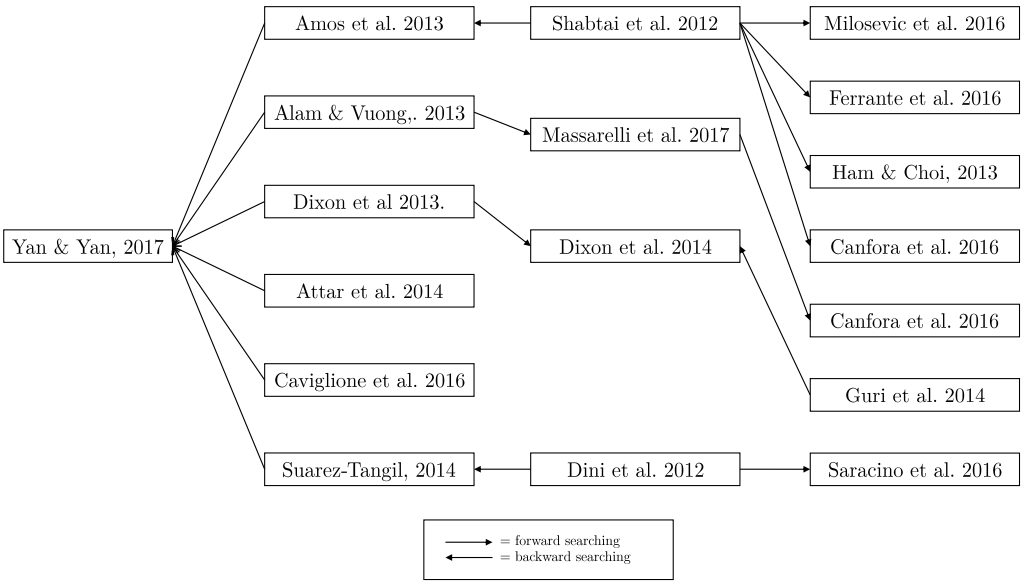


FIGURE B.1: Literature Review Method

Appendix C

Data exploration I

This appendix includes the findings of the first data exploration. First the general findings of all three probes used in this research are described. Then the probes are described seperately.

General findings

An important finding of all three probes, is the difference in distribution of data per user as can be seen in Figure C.1. Some users are overrepresented in the dataset. The ranking of the users based on the amount of data seems to be the same for the Malware and the System probe. However, this is not the case for the Apps probe. The cause for this is unknown. The distribution of the users for the final dataset is described in Section 3.3

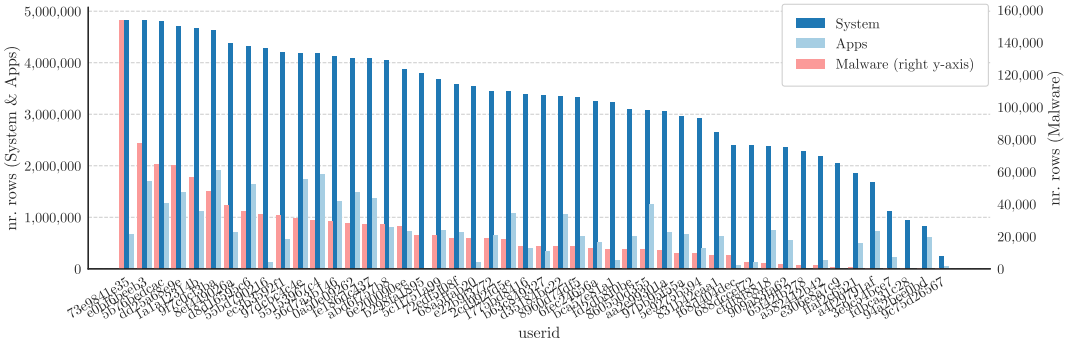


FIGURE C.1: Data distribution of users

Another important finding is the difference in distribution of data over time. As can be seen in Figure C.2, both the Malware probe and Apps probe have a peak around months 3 and 4, and months 9 and 10. This indicates that more actions were performed by malware versions 3, 4, 9, and 10, in comparison with the other malware versions.

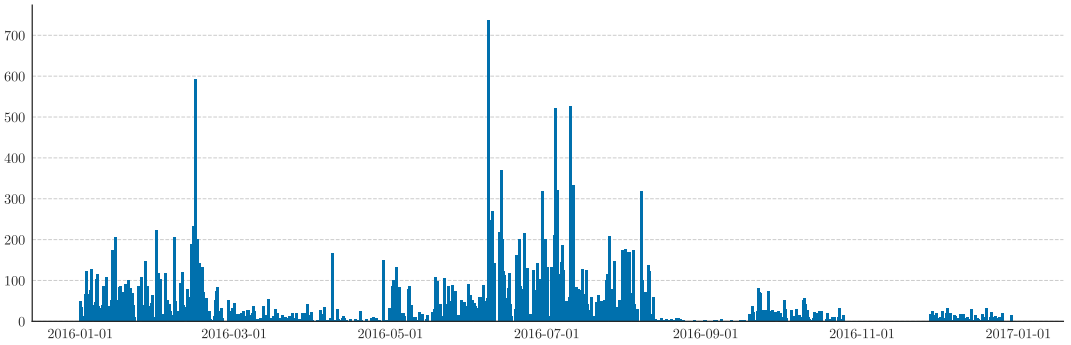


FIGURE C.2: Data distribution over time

Malware dataset findings

An important distribution in the Malware dataset is the distribution of data among the different malware types. As described in the previous Section, malware types 2, 3, 8 and 9, are overrepresented in the dataset. Figure C.3 shows the distribution of malicious and benign

actions and sessions, per malware type.
This Figure shows that malicious actions are overrepresented in the Malware dataset. Roughly 90 percent of the Malware data points are malicious actions and 10 percent are benign data points. To have a representative dataset, i.e. a dataset reflecting real-life circumstances, the dataset is balanced to include fewer malicious data points and more benign data points. This process is described in Section 4.4.
Appendix X describes the other findings of the exploration of the Malware dataset.

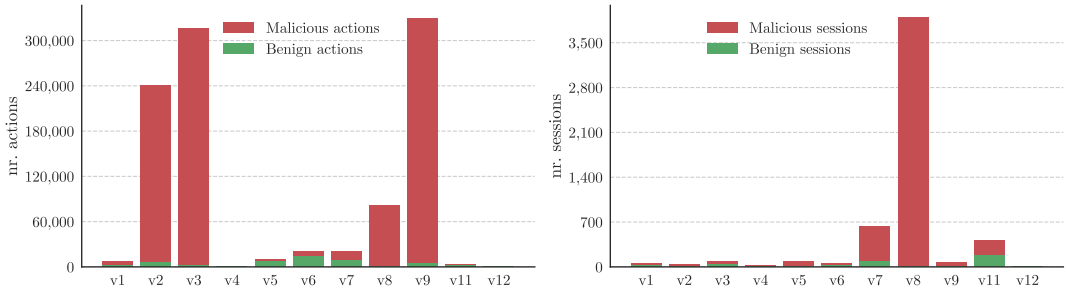


FIGURE C.3: Actions and sessions distribution per malware type

System and Apps dataset findings

The System and Apps dataset consist of more than 80% of numerical columns. The column descriptions indicate that some columns might be highly correlated as they describe the same value in another metric. For example: the *traffic_mobilerxbytes* column describes the network traffic received via a mobile network in bytes, and the *traffic_mobilerxpackets* describes the network traffic received via a mobile network in packets. Figure C.4 shows the correlation heatmaps for a subset of the Network and Memory feature categories. As expected, some columns are highly correlated due to the aforementioned reason. This might suggest that features can be removed without affecting the performance of classifiers. In the case of the Naïve Bayes classifier, the removal of features might improve the performance as the naivety of the classifier assumes no correlation amongst features. The correlation heatmaps of the other features and of the Apps probe are shown in Appendix X.

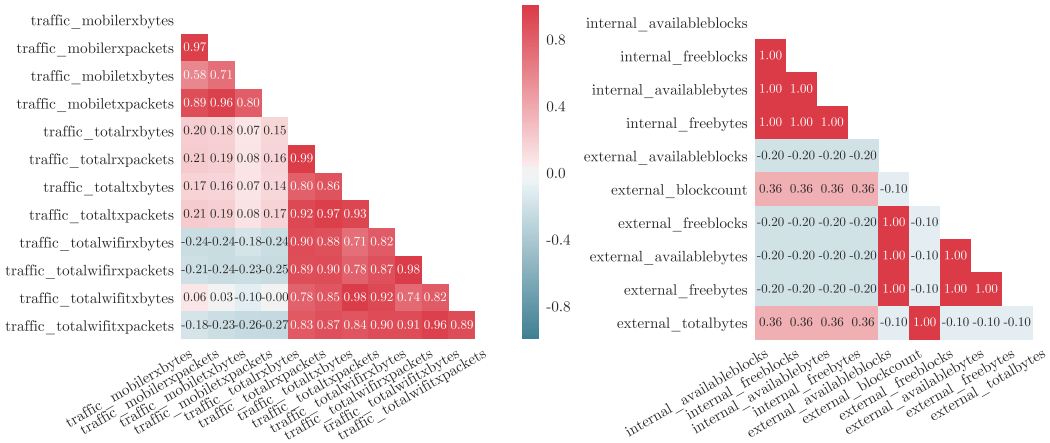


FIGURE C.4: Correlation heatmap of Network traffic features (left) and Memory features (right) in System probe

Appendix D

Android framework

Android is an open-source platform for mobile devices. The Android Framework can be seen in Figure D.1. The different layers are described below and are based on the publicly available information provided by Google regarding the Android platform [80][81][82] and a recent paper on Android security [2].

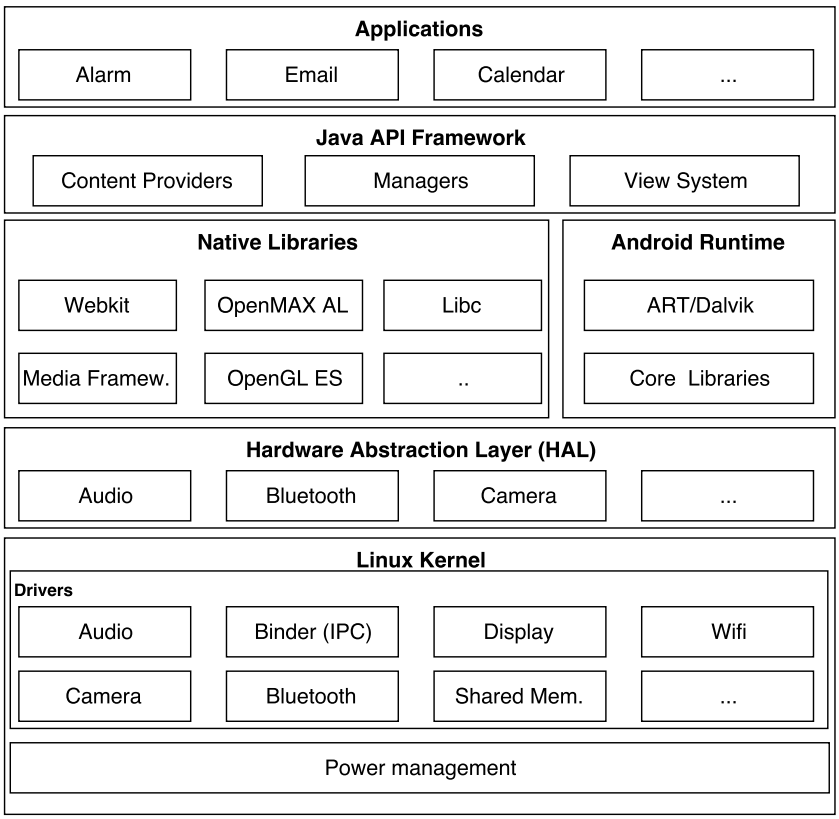


FIGURE D.1: Android Framework

Applications
This layer consists of the different applications on an Android device. These applications can either be pre-installed or user-installed.

Android Framework/ Application Framework / Java API Framework
This framework is used by applications developers and provides different APIs for applications to call.

Native Libraries
Native libraries writtin in C and C++ are used by many core android system components and services, such as ART and HAL. Some of these native libraries are accessible via Java framework APIs.

Android Runtime

This part consists of core libraries and either virtual machines in the form of Dalvik or ART. Applications are ran as separate processes in different virtual machines.

Hardware Abstraction Layer (HAL)
This layer provides a standard interface for the different hardware components of the device, such as the camera, speaker, keypad etc. This layer is used in order to get the hardware capabilities to the higher-level Java API framework.

Linux Kernel
The foundation of the Android platform is the Linux kernel. Android uses this kernel with a few adjustments for it to work effectively on a mobile embedded platform.

Android security mechanisms

Android provides different ways to secure its platform. The security features can be subdivided as security features in the Linux kernel, and application security features.

D.0.0.1 Kernel security features

The key security features of the Linux kernel are listed below with a short description.

The Application Sandbox
Different application resources are identified and isolated using the Linux user-based protection. This is done by running every Android application as a separate process and assigning it a unique user ID (UID). By default, the kernel-level Application Sandbox does not allow different applications to interact with each other. As this security feature is in the kernel, the layers above make use of this Application Sandbox.

System Partition and Safe mode
Android's kernel, operating system, libraries, application runtime, application framework and applications are stored in the system partition. This partition is read-only. Additionally a device can be booted in Safe Mode, allowing third-party applications only to launch when manually launched by the device owner.

Filesystem permission
This feature ensures that files cannot be altered or read by other users. As stated before every application is run as its own users, thus files created by one application cannot be read or altered by another application.

Verified boot
This security mechanism checks during booting whether the device still in the same state as it was last used.

D.0.0.2 Application security features

The key security features of Android applications are listed below with a short description.

Android Permission model
Applications can by default only access a limited range of system resources. Applications can access more system resources via APIs, however it needs permission for some sensitive APIs. This security mechanism is therefore also known as Permissions. The sensitive APIs being protected are camera functions, location data, bluetooth functions, telephone functions, SMS/MMS functions, and network/data connections. The permissions an application requires are stated in a file called the manifest file. This file is used during the installation of an application and users are prompted during the installation whether they want to accept an application accessing the defined resources.

Interprocess communication
Android allows for different ways of interprocess communications. The mechanisms for these communications are: binders, services, intents, and contentproviders.
Binders allow for remote procedure calls. In the case of applications it allows for applications to use methods of other applications.
Services are bodies of code running on the background. Other components can bind to a service and invoke methods on it via remote procedure calls. For example, the media player can be a service which keeps on running even when the music app is not longer running.
Intents are intention message objects. An application can have the intention of for example displaying a web page. Then an intent instance is created and handed to the system. The system can then locate which application or code can run this intent (in this case a browser application).
ContentProviders allow access to data on the device via data storehouses. Contentproviders is used for applications to expose data of its own or to access data of other applications.

Application signing
Applications need to be signed by the developer. The Package Manager verifies that an application being installed is properly signed with the certificate included in the installation package.

Application verification
This is an option on Android 4.2 and later for users to have applications be evaluated by an application verifier before installation.

Appendix E

System features

Feature	Type	Feature category	Description
userid	string		the user ID to whom this sample belongs to.
uuid	int		Unix timestamp in milliseconds of when this event occurred.
version	string		The current version of the SherLock collection agent running on the device.
traffic_ mobilerxbytes	int	Network	Number of Bytes received over mobile data since the last activation of the T4 probe (a value of -1 indicates that this is the first sample since boot).
traffic_ mobilerxpackets	int	Network	Number of Packets received over mobile data since the last activation of the T4 probe (a value of -1 indicates that this is the first sample since boot).
traffic_ mobiletxbytes	int	Network	Number of Bytes transmitted over mobile data since the last activation of the T4 probe (a value of -1 indicates that this is the first sample since boot).
traffic_ mobiletxpackets	int	Network	Number of Packets transmitted over mobile data since the last activation of the T4 probe (a value of -1 indicates that this is the first sample since boot).
traffic_ totalrxbytes	int	Network	Number of Bytes received over all networks since the last activation of the T4 probe (a value of -1 indicates that this is the first sample since boot).
traffic_ totalrxpackets	int	Network	Number of Packets received over all networks since the last activation of the T4 probe (a value of -1 indicates that this is the first sample since boot).
traffic_ totaltxbytes	int	Network	Number of Bytes transmitted over all networks since the last activation of the T4 probe (a value of -1 indicates that this is the first sample since boot).
traffic_ totaltxpackets	int	Network	Number of Packets transmitted over all networks since the last activation of the T4 probe (a value of -1 indicates that this is the first sample since boot).
traffic_ totalwifirxbytes	int	Network	Number of Bytes received over Wi-Fi since the last activation of the T4 probe (a value of -1 indicates that this is the first sample since boot).
traffic_ totalwifirxpackets	int	Network	Number of Packets received over all networks since the last activation of the T4 probe (a value of -1 indicates that this is the first sample since boot).
traffic_ totalwifitxbytes	int	Network	Number of Bytes transmitted over Wi-Fi since the last activation of the T4 probe (a value of -1 indicates that this is the first sample since boot).
traffic_ totalwifitxpackets	int	Network	Number of Packets transmitted over Wi-Fi since the last activation of the T4 probe (a value of -1 indicates that this is the first sample since boot).
traffic_ timestamp	string	Network	DateTime indicating when the traffic was calculated.
battery_ charge_type	int	Battery	A value indicating the method of charging.

battery_ current_avg	int	Battery	Average battery current in microamperes, as an integer. Positive values indicate net current entering the battery from a charge source, negative values indicate net current discharging from the battery. The time period over which the average is computed may depend on the fuel gauge hardware and its configuration.
battery_health	int	Battery	A value that indicated the current health of the battery (e.g., good, hot, over voltage,...)
battery_icon_ small	int	Battery	The resource ID of a small status bar icon indicating the current battery state
battery_ invalid_ charger	int	Battery	Indication if the charger is invalid.
battery_level	int	Battery	The current battery level (0-100)
battery_online	bool	Battery	An indication if the battery is operational.
battery_ plugged	bool	Battery	An indication if the battery is plugged in.
battery_present	string	Battery	An indication if the battery is in the device.
battery_scale	int	Battery	The maximum battery level.
battery_status	int	Battery	the current status constant.
battery_ technology	string	Battery	The technology of the current battery.
battery_ temperature	int	Battery	The current battery temperature in tenths of a degree Centigrade.
battery_ timestamp	string	Battery	A DateTime indicating when the battery statistics were sampled.
battery_voltage	int	Battery	current battery voltage in millivolts.
cpuhertz	int	CPU	the current clock speed of the CPU taken from proc/sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq
cpu_0	int	CPU	CPU utilization of core #0 in percentage.
cpu_1	int	CPU	CPU utilization of core #1 in percentage.
cpu_2	int	CPU	CPU utilization of core #2 in percentage.
cpu_3	int	CPU	CPU utilization of core #3 in percentage.
total_cpu	int	CPU	Total CPU utilization in percentage.
totalmemory_ freesize	int	Memory	Memory free in the Android heap.
totalmemory_ max_size	int	Memory	Max memory avalaible in the Android heap.
totalmemory_ total_size	int	Memory	Total memory in the Android heap.
totalmemory_ used_size	int	Memory	Total memory used in the Android heap.
memtotal	int	Memory	Total amount of usable RAM, in kibibytes, which is physical RAM minus a number of reserved bits and the kernel binary code.
Memfree	int	Memory	The amount of physical RAM, in kibibytes, left unused by the system.
buffers	int	Memory	The amount, in kibibytes, of temporary storage for raw disk blocks.
cached	int	Memory	The amount of physical RAM, in kibibytes, used as cache memory.
swapcached	int	Memory	The amount of memory, in kibibytes, that has once been moved into swap, then back into the main memory, but still also remains in the swapfile. This saves I/O, because the memory does not need to be moved into swap again.
active	int	Memory	The amount of memory, in kibibytes, that has been used more recently and is usually not reclaimed unless absolutely necessary.
inactive	int	Memory	The amount of memory, in kibibytes, that has been used less recently and is more eligible to be reclaimed for other purposes.
active_anon	int	Memory	The amount of anonymous and tmpfs/shmem memory, in kibibytes, that is in active use, or was in active use since the last time the system moved something to swap.

inactive_anon	int	Memory	The amount of anonymous and tmpfs/shmem memory, in kibibytes, that is a candidate for eviction.
active_file	int	Memory	The amount of file cache memory, in kibibytes, that is in active use, or was in active use since the last time the system reclaimed memory.
inactive_file	int	Memory	The amount of file cache memory, in kibibytes, that is newly loaded from the disk, or is a candidate for reclaiming.
unevictable	int	Memory	The amount of memory, in kibibytes, discovered by the pageout code, that is not evictable because it is locked into memory by user programs.
mlocked	int	Memory	The total amount of memory, in kibibytes, that is not evictable because it is locked into memory by user programs.
hightotal	int	Memory	The total amount of memory, in kilobytes, that is not directly mapped into kernel space.
highfree	int	Memory	The free memory, in kilobytes, that is not directly mapped into kernel space.
lowtotal	int	Memory	The total amount of memory, in kilobytes, that is directly mapped into kernel space.
lowfree	int	Memory	The free memory, in kilobytes, that is directly mapped into kernel space.
swaptotal	int	Memory	The total amount of swap available, in kibibytes.
swapfree	int	Memory	The total amount of swap free, in kibibytes.
dirty	int	Memory	The total amount of memory, in kibibytes, waiting to be written back to the disk.
writeback	int	Memory	The total amount of memory, in kibibytes, actively being written back to the disk.
anonpages	int	Memory	The total amount of memory, in kibibytes, used by pages that are not backed by files and are mapped into userspace page tables.
mapped	int	Memory	The memory, in kibibytes, used for files that have been mmaped, such as libraries.
shmem	int	Memory	The total amount of memory, in kibibytes, used by shared memory (shmem) and tmpfs.
slab	int	Memory	The total amount of memory, in kibibytes, used by the kernel to cache data structures for its own use.
sreclaimable	int	Memory	The part of Slab that can be reclaimed, such as caches.
sunreclaim	int	Memory	The part of Slab that cannot be reclaimed even when lacking memory.
kernelstack	int	Memory	The amount of memory, in kibibytes, used by the kernel stack allocations done for each task in the system.
pagetables	int	Memory	The total amount of memory, in kibibytes, dedicated to the lowest page table level.
commitlimit	int	Memory	The total amount of memory currently available to be allocated on the system based on the overcommit ratio.
committed_as	int	Memory	The total amount of memory, in kibibytes, estimated to complete the workload. This value represents the worst case scenario value, and also includes swap memory.
vmalloctotal	int	Memory	The total amount of memory, in kibibytes, of total allocated virtual address space.
vmallocused	int	Memory	The total amount of memory, in kibibytes, of used virtual address space.
vmallocchunk	int	Memory	The largest contiguous block of memory, in kibibytes, of available virtual address space.
msmgpio_cpu0	int	IO Interrupts	Accumulative interrupts for the msmgpio component. Interrupts on CPU #0.
msmgpio_sum_cpu123	int	IO Interrupts	Accumulative interrupts for the msmgpio component. Interrupts on CPUs #1, #2, #3.
wcd9xxx_cpu0	int	IO Interrupts	Accumulative interrupts for the wcd9xxx component. Interrupts on CPU #0.
wcd9xxx_sum_cpu123	int	IO Interrupts	Accumulative interrupts for the wcd9xxx component. Interrupts on CPUs #1, #2, #3.
pn547_cpu0	int	IO Interrupts	Accumulative interrupts for the pn547component. Interrupts on CPU #0.

pn547_sum_cpu123	int	IO Interrupts	Accumulative interrupts for the pn547component. Interrupts on CPUs #1, #2, #3.
cypress_touchkey_cpu0	int	IO Interrupts	Accumulative hardware interrupt count of back button presses. Interrupts on CPU #0.
cypress_touchkey_sum_cpu123	int	IO Interrupts	Accumulative hardware interrupt count of back button presses. Interrupts on CPUs #1, #2, #3.
synaptics_rmi4_i2c_cpu0	int	IO Interrupts	Accumulative hardware interrupt count for the touch screen (a single gesture may incur many interrupts –e.g., x y coordinate change). Interrupts on CPU #0.
synaptics_rmi4_i2c_sum_cpu123	int	IO Interrupts	Accumulative hardware interrupt count for the touch screen (a single gesture may incur many interrupts –e.g., x y coordinate change). Interrupts on CPUs #1, #2, #3.
sec_headset_detect_cpu0	int	IO Interrupts	Accumulative hardware interrupt count for head set detection. Interrupts on CPU #0.
sec_headset_detect_sum_cpu123	int	IO Interrupts	Accumulative hardware interrupt count for head set detection. Interrupts on CPUs #1, #2, #3.
flip_cover_cpu0	int	IO Interrupts	Accumulative hardware interrupt count for head set detection. Interrupts on CPU #0.
flip_cover_sum_cpu123	int	IO Interrupts	Accumulative hardware interrupt count for head set detection. Interrupts on CPUs #1, #2, #3.
home_key_cpu0	int	IO Interrupts	Accumulative hardware interrupt count of home key presses. Interrupts on CPU #0.
home_key_sum_cpu123	int	IO Interrupts	Accumulative hardware interrupt count of home key presses. Interrupts on CPUs #1, #2, #3.
volume_down_cpu0	int	IO Interrupts	Accumulative hardware interrupt count of volume down button presses. Interrupts on CPU #0.
volume_down_sum_cpu123	int	IO Interrupts	Accumulative hardware interrupt count of volume down button presses. Interrupts on CPUs #1, #2, #3.
volume_up_cpu0	int	IO Interrupts	Accumulative hardware interrupt count of volume up button presses. Interrupts on CPU #0.
volume_up_sum_cpu123	int	IO Interrupts	Accumulative hardware interrupt count of volume up button presses. Interrupts on CPUs #1, #2, #3.
companion_cpu0	int	IO Interrupts	Accumulative hardware interrupt count of companion occurrences. Interrupts on CPU #0.
companion_sum_cpu123	int	IO Interrupts	Accumulative hardware interrupt count of companion occurrences. Interrupts on CPUs #1, #2, #3.
slimbus_cpu0	int	IO Interrupts	Accumulative interrupt count on the slimbus. Interrupts on CPU #0.
slimbus_sum_cpu123	int	IO Interrupts	Accumulative interrupt count on the slimbus. Interrupts on CPUs #1, #2, #3.
function_call_interrupts_cpu0	int	IO Interrupts	Accumulative software interrupt count on function calls. Interrupts on CPU #0.
function_call_interrupts_sum_cpu123	int	IO Interrupts	Accumulative software interrupt count on function calls. Interrupts on CPUs #1, #2, #3.
cpu123_intr_prs	int	IO Interrupts	Accumulative interrupt count on the intr_prs element.
tot_user	int	CPU	The number of normal processes executing in user mode.
tot_nice	int	CPU	The number of niced processes executing in user mode.
tot_system	int	CPU	The number of processes executing in kernel mode.
tot_idle	int	CPU	The number of twiddling thumbs.
tot_iowait	int	CPU	The number of waiting for I/O to complete.
tot_irq	int	CPU	The number of servicing interrupts.
tot_softirq	int	CPU	The number of servicing softirqs.
ctxt	int	CPU	The total number of context switches across all CPUs.

btime	int	CPU	The time at which the system booted, in seconds since the Unix epoch.
processes	int	CPU	The number of processes and threads created, which includes (but is not limited to) those created by calls to the fork() and clone() system calls.
procs_running	int	CPU	The number of processes currently running on CPUs.
procs_blocked	int	CPU	The number of processes currently blocked, waiting for I/O to complete.
connectedwifi_ ssid	int	Wifi	The salted hash of the connected Wi-Fi access point's SSID.
connectedwifi_ level	int	Wifi	The reception level of the connected Wi-Fi access point (RSSI).
internal_ availableblocks	int	Storage	Avalaible blocks in internal storage.
internal_ blockcount	int	Storage	Number of blocks in internal storage.
internal_ freeblocks	int	Storage	Free blocks in internal storage.
internal_ blocksize	int	Storage	Block size in internal storage.
internal_ availablebytes	int	Storage	Avalaible Bytes in internal storage.
internal_ freebytes	int	Storage	Free Bytes in internal storage.
internal_ totalbytes	int	Storage	Total Bytes in external (SD card) storage.
external_ availableblocks	int	Storage	Avalaible blocks in external (SD card) storage.
external_ blockcount	int	Storage	Number of blocks in external (SD card) storage.
external_ freeblocks	int	Storage	Number of blocks in external (SD card) storage.
external_ blocksize	int	Storage	Block size in external (SD card) storage.
external_ availablebytes	int	Storage	Avalaible Bytes in external (SD card) storage.
external_ freebytes	int	Storage	Free Bytes in external (SD card) storage.
external_ totalbytes	int	Storage	Total Bytes in external (SD card) storage.

Appendix F

Apps features

Feature	type	Feature Category	description
userid	string	Metadata	the user ID to whom this sample belongs to.
uuid	int	Metadata	Unix timestamp in milliseconds of when this event occurred.
applicationname	string	Metadata	The name of the sampled application described in this record.
cpu_usage	double	App_CPU	The percent of CPU utilization normalized to a constant CPU clock speed. Note that this data field has been depreciated. It is recommended to use the stime, utime, cstime, cutime fields to measure the app's activity.
packagename	string	App_Info	The Android package name of this app (e.g., com.example.helloandroid)
packageuid	int	App_Info	The UID identifier of this app's package.
uidrxbytes	int	App_ Network	Bytes received by this application since the last time the T4 probe was activated (approximately 5 seconds on average –compare uuids for accuracy). If this is the first sample since boot, then the value is -1.
uidrxpackets	int	App_ Network	Packets received by this application since the last time the T4 probe was activated (approximately 5 seconds on average –compare uuids for accuracy). If this is the first sample since boot, then the value is -1.
uidtxbytes	int	App_ Network	Bytes transmitted by this application since the last time the T4 probe was activated (approximately 5 seconds on average –compare uuids for accuracy). If this is the first sample since boot, then the value is -1.
uidtxpackets	int	App_ Network	Packets transmitted by this application since the last time the T4 probe was activated (approximately 5 seconds on average –compare uuids for accuracy). If this is the first sample since boot, then the value is -1.
cguest_time	int	App_CPU	Guest time of the process's children, measured in clock ticks.
cmaj_flt	int	App_ Memory	The number of major faults that the process's waited-for children have made.
cstime	int	App_CPU	Amount of time that this process's waited-for children have been scheduled in kernel mode, measured in clock ticks.
cutime	int	App_CPU	Amount of time that this process's waited-for children have been scheduled in user mode, measured in clock ticks. This includes guest time, cguest_time (time spent running a virtual CPU).
dalvikprivatedirty	int	App_ Memory	The private dirty pages used by dalvik heap.
dalvikpss	int	App_ Memory	The proportional set size for dalvik heap.
dalvikshareddirty	int	App_ Memory	The shared dirty pages used by dalvik heap.
guest_time	int	App_CPU	Guest time of the process (time spent running a virtual CPU for a guest operating system), measured in clock ticks.
importance	int	App_ Process	The relative importance level that the system places on this process (details). For example, background, foreground, service, sleeping, ...etc.
importancereasoncode	int	App_ Process	The reason for importance, if any (details).
importancereasonpid	int	App_ Process	For the specified values of importanceReasonCode, this is the process ID of the other process that is a client of this process (details).

lru	int	App_ Memory	An additional ordering within a particular Android importance category, providing finer-grained information about the relative utility of processes within a category (details).
nativeprivatedirty	int	App_ Memory	The private dirty pages used by the native heap.
nativepss	int	App_ Memory	The proportional set size for the native heap.
nativeshareddirty	int	App_ Memory	The shared dirty pages used by the native heap.
num_threads	int	App_CPU	Number of threads in this process.
otherprivatedirty	int	App_ Memory	The private dirty pages used by everything else.
otherpss	int	App_ Memory	The proportional set size for everything else.
othershareddirty	int	App_ Memory	The shared dirty pages used by everything else.
pgid	int	App_ Process	
pid	int	App_ Process	The process ID of this process.
ppid	int	App_ Process	The PID of parent process.
priority	int	App_CPU	(Explanation for Linux 2.6) For processes running a real-time scheduling policy (policy below; see sched_setscheduler(2)), this is the negated scheduling priority, minus one; that is, a number in the range -2 to -100, corresponding to real-time priorities 1 to 99. For processes running under a non-real-time scheduling policy, this is the raw nice value (setpriority(2)) as represented in the kernel. The kernel stores nice values as numbers in the range 0 (high) to 39 (low), corresponding to the user-visible nice range of -20 to 19.
rss	int	App_ Memory	Resident Set Size: number of pages the process has in real memory. This is just the pages which count toward text, data, or stack space. This does not include pages which have not been demand-loaded in, or which are swapped out.
rsslim	int	App_ Memory	Current soft limit in bytes on the rss of the process.
sid	int	App_ Process	The process's session ID.
start_time	int	App_ Process	The time the process started after system boot. In kernels before Linux 2.6, this value was expressed in jiffies. Since Linux 2.6, the value is expressed in clock ticks.
state	string	App_ Process	Current state of the process. One of "R (running)", "S (sleeping)", "D (disk sleep)", "T (stopped)", "T (tracing stop)", "Z (zombie)", or "X (dead)".
stime	int	App_CPU	Amount of time that this process has been scheduled in kernel mode, measured in clock ticks.
tcomm	string	App_ Process	An associated string with the executable's name.
utime	int	App_CPU	Amount of time that this process has been scheduled in user mode, measured in clock ticks. This includes guest time, guest_time (time spent running a virtual CPU, see below), so that applications that are not aware of the guest time field do not lose that time from their calculations.
vsize	int	App_ Memory	Virtual memory size in bytes.
version_code	int	App_Info	An integer used as an internal version number for the Android app. This number is used only to determine whether one version is more recent than another, with higher numbers indicating more recent versions. This is not the version number shown to users (details).
version_name	string	App_Info	A string used as the version number shown to users. This setting can be specified as a raw string or as a reference to a string resource.
sherlock_ version	string	Metadata	The current version of the SherLock collection agent running on the device.

tgpid	int	App_ Process	The ID of the foreground process group of the controlling terminal of the process. -1 if the process is not connected to a terminal.
Flags	string	App_ Process	the internal kernel flags holding the status of the socket (e.g., 00010000).
Wchan	string	App_ Process	This is the "channel" in which the process is waiting. It is the address of a location in the kernel where the process is sleeping. The corresponding symbolic name can be found in /proc/[pid]/wchan.
exit_signal	int	App_ Process	Signal to be sent to parent when we die.
minflt	int	App_ Memory	The number of minor faults the process has made which have not required loading a memory page from disk.
cminflt	int	App_ Memory	The number of minor faults that the process's waited-for children have made.
majflt	int	App_ Memory	The number of major faults the process has made which have required loading a memory page from disk.
startcode	int	App_ Process	The address above which program text can run.
endcode	int	App_ Process	The address below which program text can run.
nice	int	App_CPU	The nice value a value in the range 19 (low priority) to -20 (high priority).
Itrealvalue	int	App_CPU	The time in jiffies before the next SIGALRM is sent to the process due to an interval timer. Since kernel 2.6.17, this field is no longer maintained, and is hard coded as 0.
Processor	int	App_CPU	CPU number last executed on.
rt_priority	int	App_CPU	Real-time scheduling priority, a number in the range 1 to 99 for processes scheduled under a real-time policy, or 0, for non-real-time processes.

Appendix G

Malware features

Feature	type	Feature Category	description
userid	string	metadata	the user ID to whom this sample belongs to.
uuid	int	metadata	Unix timestamp in milliseconds of when this event occurred.
version	string	metadata	The version of Moriarty this clue belongs to.
action	string	action	The general action being performed by the Moriarty agent.
actionType	benign, malicious	action	The intent of the action being performed.
details	string	action	Details on the action performed. This field may contain additional information. The format of this field is: <details> (<data field>,...);<data value>;... e.g.: "Successful send to server(duration
sessionID	int	session	The ID for the on going session to which this clue belongs.
sessionType	benign, malicious	session	The intent of the ongoing session.
behavior	string	session	Sometimes sessions may overlap (e.g. benign game playing and a spyware service). This field help segregate overlapping sessions and identify their intents.

Appendix H

Featureset overview

fs1	fs2	fs3
userid	userid	*fs1 + fs3*
uuid	uuid	
version	applicationname	
traffic_mobilerxbytes	cpu_usage	
traffic_mobilerxpackets	packagename	
traffic_mobiletxbytes	packageuid	
traffic_mobiletxpackets	uidrxbytes	
traffic_totalrxbytes	uidrxpackets	
traffic_totalrxpackets	uidtxbytes	
traffic_totaltxbytes	uidtxpackets	
traffic_totaltxpackets	cguest_time	
traffic_totalwifirxbytes	cmaj_flt	
traffic_totalwifirxpackets	cstime	
traffic_totalwifitxbytes	cutime	
traffic_totalwifitxpackets	dalvikprivatedirty	
traffic_timestamp	dalvikpss	
battery_charge_type	dalvikshareddirty	
battery_current_avg	guest_time	
battery_health	importance	
battery_icon_small	importancereasoncode	
battery_invalid_charger	importancereasonpid	
battery_level	lru	
battery_online	nativeprivatedirty	
battery_plugged	nativepss	
battery_present	nativeshareddirty	
battery_scale	num_threads	
battery_status	otherprivatedirty	
battery_technology	otherpss	
battery_temperature	othershareddirty	
battery_timestamp	pgid	
battery_voltage	pid	
cpuhertz	ppid	
cpu_0	priority	
cpu_1	rss	
cpu_2	rsslim	
cpu_3	sid	
total_cpu	start_time	
totalmemory_freesize	state	
totalmemory_max_size	stime	
totalmemory_total_size	tcomm	
totalmemory_used_size	utime	

FIGURE H.1: Features included per featureset (fs1 - fs3)

fs4			fs5		fs6
userid	swapcached	home_key_sum_cpu123	userid	tgpid	*fs4 + fs5*
uuid	active	volume_down_cpu0	uuid	Flags	
version	inactive	volume_down_sum_cpu 123	applicationname	Wchan	
traffic_mobilerxbytes	active_anon	volume_up_cpu0	cpu_usage	exit_signal	
traffic_mobilerxpackets	inactive_anon	volume_up_sum_cpu12 3	packagename	minflt	
traffic_mobiletxbytes	active_file	companion_cpu0	packageuid	cminflt	
traffic_mobiletxpackets	inactive_file	companion_sum_cpu123	uidrxbytes	majflt	
traffic_totalrxbytes	unevictable	slimbus_cpu0	uidrxpackets	startcode	
traffic_totalrxpackets	mlocked	slimbus_sum_cpu123	uidtxbytes	endcode	
traffic_totaltxbytes	hightotal	function_call_interrupts _cpu0	uidtxpackets	nice	
traffic_totaltxpackets	highfree	function_call_interrupts _sum_cpu123	cguest_time	Itrealvalue	
traffic_totalwifirxbytes	lowtotal	cpu123_intr_prs	cmaj_flt	Processor	
traffic_totalwifirxpackets	lowfree	tot_user	cstime	rt_priority	
traffic_totalwifitxbytes	swaptotal	tot_nice	cutime		
traffic_totalwifitxpackets	swapfree	tot_system	dalvikprivatedirty		
traffic_timestamp	dirty	tot_idle	dalvikpss		
battery_charge_type	writeback	tot_iowait	dalvikshareddirty		
battery_current_avg	anonpages	tot_irq	guest_time		
battery_health	mapped	tot_softirq	importance		
battery_icon_small	shmem	ctxt	importancereasoncode		
battery_invalid_charger	slab	btime	importancereasonpid		
battery_level	sreclaimable	processes	lru		
battery_online	sunreclaim	procs_running	nativeprivatedirty		
battery_plugged	kernelstack	procs_blocked	nativepss		
battery_present	pagetables	connectedwifi_ssid	nativeshareddirty		
battery_scale	commitlimit	connectedwifi_level	num_threads		
battery_status	committed_as	internal_availableblocks	otherprivatedirty		
battery_technology	vmalloctotal	internal_blockcount	otherpss		
battery_temperature	vmallocused	internal_freeblocks	othershareddirty		
battery_timestamp	vmallocchunk	internal_blocksize	pgid		
battery_voltage	msmgpio_cpu0	internal_availablebytes	pid		
cpuhertz	msmgpio_sum_cpu123	internal_freebytes	ppid		
cpu_0	wcd9xxx_cpu0	internal_totalbytes	priority		
cpu_1	wcd9xxx_sum_cpu123	external_availableblocks	rss		
cpu_2	pn547_cpu0	external_blockcount	rsslim		
cpu_3	pn547_sum_cpu123	external_freeblocks	sid		
total_cpu	cypress_touchkey_cpu0	external_blocksize	start_time		
totalmemory_freesize	cypress_touchkey_sum_ cpu123	external_availablebytes	state		
totalmemory_max_size	synaptics_rmi4_i2c_cpu 0	external_freebytes	stime		
totalmemory_total_size	synaptics_rmi4_i2c_su m_cpu123	external_totalbytes	tcomm		
totalmemory_used_size	sec_headset_detect_cpu 0		utime		
memtotal	sec_headset_detect_su m_cpu123		vsize		
Memfree	flip_cover_cpu0		version_code		
buffers	flip_cover_sum_cpu123		version_name		
cached	home_key_cpu0		sherlock_version		
cached	home_key_cpu0		sherlock_version		

FIGURE H.2: Features included per featureset (fs4 - fs6)

Appendix I

Features overview

Ada (training = all, testing = normal holdout)		
fs1	**fs2**	**fs3**
totalmemory_total_size	rss_mor_app	rss_mor_app
totalmemory_used_size	utime_mor_app	utime_mor_app
battery_voltage	otherprivatedirty_mor_app	dalvikpss_mor_app
battery_temperature	dalvikprivatedirty_mor_app	otherprivatedirty_mor_app
traffic_totaltxbytes	importance_mor_app	importance_mor_app
total_cpu	otherpss_mor_app	otherpss_mor_app
totalmemory_freesize	dalvikpss_mor_app	dalvikprivatedirty_mor_app
traffic_totalrxbytes	num_threads_mor_app	num_threads_mor_app
battery_level	vsize_mor_app	vsize_mor_app
cpu_0	stime_mor_app	stime_mor_app
traffic_totaltxpackets	dalvikshareddirty_mor_app	traffic_totalrxbytes
cpu_2	cpu_usage_mor_app	dalvikshareddirty_mor_app
cpu_3	uidtxbytes_mor_app	totalmemory_used_size
traffic_totalwifitxbytes	othershareddirty_mor_app	cpu_usage_mor_app
battery_current_avg	start_time_mor_app	othershareddirty_mor_app
traffic_totalrxpackets	pid_mor_app	total_cpu
cpu_1	uidrxbytes_mor_app	uidtxbytes_mor_app
traffic_mobilerxbytes	pgid_mor_app	totalmemory_freesize
traffic_totalwifirxbytes	lru_mor_app	traffic_totaltxbytes
traffic_mobiletxbytes	uidtxpackets_mor_app	start_time_mor_app
totalmemory_max_size	ppid_mor_app	battery_voltage
traffic_totalwifitxpackets	nativeprivatedirty_mor_app	totalmemory_total_size
battery_icon_small	priority_mor_app	battery_temperature
traffic_mobiletxpackets	uidrxpackets_mor_app	pid_mor_app
traffic_totalwifirxpackets	cstime_mor_app	uidrxbytes_mor_app
traffic_mobilerxpackets		ppid_mor_app
		battery_level
		cpu_0
		totalmemory_max_size
		traffic_totalwifitxbytes
		traffic_totaltxpackets
		cpu_2
		cpu_3
		lru_mor_app
		uidtxpackets_mor_app
		nativeprivatedirty_mor_app
		priority_mor_app
		cpu_1
		traffic_totalrxpackets
		pgid_mor_app
		traffic_mobiletxbytes
		cstime_mor_app
		traffic_totalwifirxbytes

FIGURE I.1: Features included in best performing Ada models

RF (training = all, testing = normal holdout)		
fs1	**fs2**	**fs3**
traffic_mobilerxbytes	cpu_usage_mor_app	dalvikprivatedirty_mor_app
traffic_mobiletxbytes	uidrxbytes_mor_app	dalvikpss_mor_app
traffic_mobiletxpackets	uidrxpackets_mor_app	importance_mor_app
traffic_totalrxbytes	uidtxbytes_mor_app	num_threads_mor_app
traffic_totalrxpackets	uidtxpackets_mor_app	otherprivatedirty_mor_app
traffic_totaltxbytes	cmaj_flt_mor_app	otherpss_mor_app
traffic_totaltxpackets	cstime_mor_app	rss_mor_app
traffic_totalwifirxbytes	dalvikprivatedirty_mor_app	stime_mor_app
traffic_totalwifitxbytes	dalvikpss_mor_app	utime_mor_app
traffic_totalwifitxpackets	dalvikshareddirty_mor_app	vsize_mor_app
battery_current_avg	importance_mor_app	
battery_icon_small	importancereasonpid_mor_app	
battery_level	lru_mor_app	
battery_temperature	nativeprivatedirty_mor_app	
battery_voltage	nativepss_mor_app	
cpu_0	nativeshareddirty_mor_app	
cpu_1	num_threads_mor_app	
cpu_2	otherprivatedirty_mor_app	
cpu_3	otherpss_mor_app	
total_cpu	othersharedirty_mor_app	
totalmemory_freesize	pgid_mor_app	
totalmemory_max_size	pid_mor_app	
totalmemory_total_size	ppid_mor_app	
totalmemory_used_size	priority_mor_app	
	rss_mor_app	
	start_time_mor_app	
	stime_mor_app	
	utime_mor_app	
	vsize_mor_app	

FIGURE I.2: Features included in best performing RF models

KNN (training = all, testing = normal holdout)		
fs1	**fs2**	**fs3**
totalmemory_total_size	rss_mor_app	rss_mor_app
totalmemory_used_size	utime_mor_app	utime_mor_app
battery_voltage	otherprivatedirty_mor_app	dalvikpss_mor_app
battery_temperature	dalvikprivatedirty_mor_app	otherprivatedirty_mor_app
traffic_totaltxbytes	importance_mor_app	importance_mor_app
total_cpu	otherpss_mor_app	otherpss_mor_app
totalmemory_freesize	dalvikpss_mor_app	dalvikprivatedirty_mor_app
traffic_totalrxbytes	num_threads_mor_app	num_threads_mor_app
battery_level	vsize_mor_app	vsize_mor_app
cpu_0	stime_mor_app	
traffic_totaltxpackets	dalvikshareddirty_mor_app	
cpu_2	cpu_usage_mor_app	
cpu_3	uidtxbytes_mor_app	
traffic_totalwifitxbytes		
battery_current_avg		
traffic_totalrxpackets		
cpu_1		
traffic_mobilerxbytes		
traffic_totalwifirxbytes		
traffic_mobiletxbytes		
totalmemory_max_size		
traffic_totalwifitxpackets		
battery_icon_small		
traffic_mobiletxpackets		
traffic_totalwifirxpackets		
traffic_mobilerxpackets		

FIGURE I.3: Features included in best performing KNN models

NB (training = all, testing = normal holdout)		
fs1	**fs2**	**fs3**
totalmemory_total_size	rss_mor_app	rss_mor_app
totalmemory_used_size	utime_mor_app	utime_mor_app
battery_voltage	otherprivatedirty_mor_app	dalvikpss_mor_app
battery_temperature	dalvikprivatedirty_mor_app	otherprivatedirty_mor_app
traffic_totaltxbytes	importance_mor_app	importance_mor_app
total_cpu	otherpss_mor_app	otherpss_mor_app
totalmemory_freesize	dalvikpss_mor_app	dalvikprivatedirty_mor_app
traffic_totalrxbytes	num_threads_mor_app	num_threads_mor_app
battery_level	vsize_mor_app	vsize_mor_app
cpu_0	stime_mor_app	stime_mor_app
traffic_totaltxpackets	dalviksharedirty_mor_app	traffic_totalrxbytes
cpu_2	cpu_usage_mor_app	dalviksharedirty_mor_app
cpu_3	uidtxbytes_mor_app	totalmemory_used_size
traffic_totalwifitxbytes	othersharedirty_mor_app	cpu_usage_mor_app
battery_current_avg	start_time_mor_app	othersharedirty_mor_app
traffic_totalrxpackets	pid_mor_app	total_cpu
cpu_1	uidrxbytes_mor_app	uidtxbytes_mor_app
traffic_mobilerxbytes	pgid_mor_app	totalmemory_freesize
traffic_totalwifirxbytes	lru_mor_app	traffic_totaltxbytes
traffic_mobiletxbytes	uidtxpackets_mor_app	start_time_mor_app
totalmemory_max_size	ppid_mor_app	battery_voltage
traffic_totalwifitxpackets	nativeprivatedirty_mor_app	totalmemory_total_size
battery_icon_small	priority_mor_app	battery_temperature
traffic_mobiletxpackets	uidrxpackets_mor_app	pid_mor_app
traffic_totalwifirxpackets	cstime_mor_app	uidrxbytes_mor_app
traffic_mobilerxpackets	nativepss_mor_app	ppid_mor_app
battery_online	cmaj_flt_mor_app	battery_level
battery_status	nativesharedirty_mor_app	cpu_0
battery_charge_type	importancereasonpid_mor_app	totalmemory_max_size
battery_plugged	cutime_mor_app	traffic_totalwifitxbytes
battery_health	state_mor_app_S	traffic_totaltxpackets
	importancereasoncode_mor_app	cpu_2
	state_mor_app_R	cpu_3
		lru_mor_app
		uidtxpackets_mor_app
		nativeprivatedirty_mor_app
		priority_mor_app
		cpu_1
		traffic_totalrxpackets
		pgid_mor_app
		traffic_mobiletxbytes
		cstime_mor_app
		traffic_totalwifirxbytes
		nativepss_mor_app
		traffic_mobilerxbytes
		uidrxpackets_mor_app
		traffic_totalwifitxpackets
		cmaj_flt_mor_app
		traffic_mobiletxpackets
		nativesharedirty_mor_app
		traffic_totalwifirxpackets
		battery_current_avg
		traffic_mobilerxpackets
		importancereasonpid_mor_app
		cutime_mor_app
		state_mor_app_S
		importancereasoncode_mor_app
		battery_icon_small
		battery_status
		battery_online
		state_mor_app_R
		battery_plugged
		battery_charge_type
		state_mor_app_D

FIGURE I.4: Features included in best performing NB models

MLP (training = all, testing = normal holdout)		
fs1	**fs2**	**fs3**
totalmemory_total_size	rss_mor_app	rss_mor_app
totalmemory_used_size	utime_mor_app	utime_mor_app
battery_voltage	otherprivatedirty_mor_app	dalvikpss_mor_app
battery_temperature	dalvikprivatedirty_mor_app	otherprivatedirty_mor_app
traffic_totaltxbytes	importance_mor_app	importance_mor_app
total_cpu	otherpss_mor_app	otherpss_mor_app
totalmemory_freesize	dalvikpss_mor_app	dalvikprivatedirty_mor_app
traffic_totalrxbytes	num_threads_mor_app	num_threads_mor_app
battery_level	vsize_mor_app	vsize_mor_app
cpu_0	stime_mor_app	stime_mor_app
traffic_totaltxpackets	dalvikshareddirty_mor_app	traffic_totalrxbytes
cpu_2	cpu_usage_mor_app	dalvikshareddirty_mor_app
cpu_3	uidtxbytes_mor_app	totalmemory_used_size
traffic_totalwifitxbytes	othershareddirty_mor_app	cpu_usage_mor_app
battery_current_avg	start_time_mor_app	othershareddirty_mor_app
traffic_totalrxpackets	pid_mor_app	total_cpu
cpu_1	uidrxbytes_mor_app	uidtxbytes_mor_app
traffic_mobilerxbytes	pgid_mor_app	totalmemory_freesize
traffic_totalwifirxbytes	lru_mor_app	traffic_totaltxbytes
traffic_mobiletxbytes	uidtxpackets_mor_app	start_time_mor_app
totalmemory_max_size	ppid_mor_app	battery_voltage
traffic_totalwifitxpackets	nativeprivatedirty_mor_app	totalmemory_total_size
battery_icon_small	priority_mor_app	battery_temperature
traffic_mobiletxpackets	uidrxpackets_mor_app	pid_mor_app
traffic_totalwifirxpackets	cstime_mor_app	uidrxbytes_mor_app
traffic_mobilerxpackets	nativepss_mor_app	ppid_mor_app
battery_online	cmaj_flt_mor_app	battery_level
battery_status	nativeshareddirty_mor_app	cpu_0
battery_charge_type		totalmemory_max_size
battery_plugged		traffic_totalwifitxbytes
battery_health		traffic_totaltxpackets
battery_scale		cpu_2
		cpu_3
		lru_mor_app
		uidtxpackets_mor_app
		nativeprivatedirty_mor_app
		priority_mor_app
		cpu_1
		traffic_totalrxpackets
		pgid_mor_app
		traffic_mobiletxbytes
		cstime_mor_app
		traffic_totalwifirxbytes
		nativepss_mor_app
		traffic_mobilerxbytes
		uidrxpackets_mor_app
		traffic_totalwifitxpackets
		cmaj_flt_mor_app
		traffic_mobiletxpackets
		nativeshareddirty_mor_app
		traffic_totalwifirxpackets
		battery_current_avg
		traffic_mobilerxpackets
		importancereasonpid_mor_app
		cutime_mor_app
		state_mor_app_S
		importancereasoncode_mor_app
		battery_icon_small
		battery_status
		battery_online
		state_mor_app_R
		battery_plugged

FIGURE I.5: Features included in best performing mlp models

Appendix J

McNemar test statistics

		Normal holdout			Unknown device		
fs	**Testing**	**fs1**	**fs2**	**fs3**	**fs1**	**fs2**	**fs3**
fs1	**Normal holdout**						
fs2		X					
fs3		X					
fs1	**Unknown device**		X	X			
fs2		X			X		
fs3			X			X	

FIGURE J.1: McNemar test results ($alpha < 0.05$) for Ada (X indicates statistical significant difference)

		Normal holdout			Unknown device		
fs	**Testing**	**fs1**	**fs2**	**fs3**	**fs1**	**fs2**	**fs3**
fs1	**Normal holdout**						
fs2		x					
fs3		x					
fs1	**Unknown device**		x	x			
fs2		x		x			
fs3			x		x	x	

FIGURE J.2: McNemar test results ($alpha < 0.05$) for RF (X indicates statistical significant difference)

		Normal holdout			Unknown device		
fs	**Testing**	**fs1**	**fs2**	**fs3**	**fs1**	**fs2**	**fs3**
fs1	**Normal holdout**						
fs2		X					
fs3		X					
fs1	**Unknown device**		X	X			
fs2		X		X	X		
fs3			X		X		

FIGURE J.3: McNemar test results ($alpha < 0.05$) for KNN (X indicates statistical significant difference)

		Normal holdout			Unknown device		
fs	**Testing**	**fs1**	**fs2**	**fs3**	**fs1**	**fs2**	**fs3**
fs1	**Normal holdout**						
fs2		X					
fs3		X					
fs1	**Unknown device**						
fs2				X	X		
fs3			X		X		

FIGURE J.4: McNemar test results ($alpha < 0.05$) for NB (X indicates statistical significant difference)

		Normal holdout			Unknown device		
fs	**Testing**	**fs1**	**fs2**	**fs3**	**fs1**	**fs2**	**fs3**
fs1	**Normal holdout**						
fs2		X					
fs3		X	X				
fs1	**Unknown device**		X				
fs2			X				
fs3			X				

FIGURE J.5: McNemar test results ($alpha < 0.05$) for MLP (X indicates statistical significant difference)

Class.	fs	Ada			RF			KNN			NB			MLP		
		fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
Ada	**fs1**															
	fs2															
	fs3	X														
RF	**fs1**															
	fs2															
	fs3															
KNN	**fs1**															
	fs2			X												
	fs3			X												
NB	**fs1**			X	X											
	fs2			X	X											
	fs3		X	X	X		X	X								
MLP	**fs1**															
	fs2			X										X		
	fs3		X	X	X		X	X								

FIGURE J.6: McNemar test results ($alpha < 0.05$) for malware version 1 (X indicates statistical significant difference)

Class.	fs	Ada			RF			KNN			NB			MLP		
		fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
Ada	**fs1**															
	fs2	X														
	fs3	X														
RF	**fs1**		X	X												
	fs2	X		X	X											
	fs3	X			X											
KNN	**fs1**	X	X	X	X	X	X									
	fs2	X			X			X								
	fs3	X			X			X								
NB	**fs1**	X	X	X	X	X	X	X	X	X						
	fs2	X	X	X	X	X	X		X	X	X					
	fs3		X	X		X	X		X	X	X					
MLP	**fs1**	X	X	X	X	X	X	X	X	X		X	X			
	fs2	X	X	X	X	X	X	X	X	X		X	X			
	fs3	X	X	X	X	X	X	X	X	X		X	X			

FIGURE J.7: McNemar test results ($alpha < 0.05$) for malware version 2 (X indicates statistical significant difference)

Class.	fs	Ada			RF			KNN			NB			MLP		
		fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
Ada	**fs1**															
	fs2															
	fs3	X														
RF	**fs1**															
	fs2	X														
	fs3	X														
KNN	**fs1**	X	X	X	X	X	X									
	fs2							X								
	fs3							X								
NB	**fs1**	X	X	X	X	X	X		X	X						
	fs2	X	X	X	X	X	X		X	X						
	fs3	X	X	X	X	X	X		X	X						
MLP	**fs1**	X	X	X	X	X	X		X	X						
	fs2	X	X	X	X	X	X		X	X				X		
	fs3	X	X	X	X	X	X		X	X						

FIGURE J.8: McNemar test results ($alpha < 0.05$) for malware version 3 (X indicates statistical significant difference)

Class.	fs	Ada			RF			KNN			NB			MLP		
		fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
Ada	**fs1**															
	fs2															
	fs3															
RF	**fs1**															
	fs2															
	fs3															
KNN	**fs1**															
	fs2															
	fs3															
NB	**fs1**															
	fs2															
	fs3															
MLP	**fs1**															
	fs2															
	fs3															

FIGURE J.9: McNemar test results ($alpha < 0.05$) for malware version 4 (X indicates statistical significant difference)

Class.	fs	Ada			RF			KNN			NB			MLP		
		fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
Ada	**fs1**															
	fs2	X														
	fs3	X														
RF	**fs1**	X	X	X												
	fs2	X			X											
	fs3	X			X											
KNN	**fs1**		X	X	X	X	X									
	fs2	X			X			X								
	fs3	X			X			X								
NB	**fs1**		X	X	X	X	X		X	X						
	fs2	X	X	X		X	X	X	X	X	X					
	fs3	X	X	X		X	X	X	X	X	X					
MLP	**fs1**		X	X	X	X	X		X	X		X	X			
	fs2		X	X	X	X	X		X	X		X	X	X		
	fs3	X			X			X			X	X	X	X	X	

FIGURE J.10: McNemar test results ($alpha < 0.05$) for malware version 5 (X indicates statistical significant difference)

Class.	fs	Ada			RF			KNN			NB			MLP		
		fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
Ada	**fs1**															
	fs2															
	fs3															
RF	**fs1**															
	fs2															
	fs3	X	X	X		X										
KNN	**fs1**	X	X	X	X	X										
	fs2	X	X	X	X	X										
	fs3	X	X	X	X	X										
NB	**fs1**						X	X	X	X						
	fs2						X	X	X	X						
	fs3						X	X	X	X						
MLP	**fs1**						X	X	X	X						
	fs2						X	X	X	X						
	fs3						X	X	X	X						

FIGURE J.11: McNemar test results ($alpha < 0.05$) for malware version 6 (X indicates statistical significant difference)

Class.	fs	Ada			RF			KNN			NB			MLP		
		fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
Ada	**fs1**															
	fs2	X														
	fs3	X														
RF	**fs1**		X	X												
	fs2	X	X	X	X											
	fs3	X			X											
KNN	**fs1**	X	X	X	X	X	X									
	fs2	X			X	X	X	X								
	fs3	X			X	X	X	X								
NB	**fs1**		X	X		X	X	X	X	X						
	fs2	X	X	X	X	X	X	X	X	X	X					
	fs3	X	X	X	X	X	X	X	X	X	X					
MLP	**fs1**		X	X		X	X	X	X	X		X	X			
	fs2	X			X	X		X			X	X	X	X		
	fs3	X			X	X	X	X			X	X	X	X		

FIGURE J.12: McNemar test results ($alpha < 0.05$) for malware version 7 (X indicates statistical significant difference)

Class.	fs	Ada			RF			KNN			NB			MLP		
		fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
Ada	**fs1**															
	fs2															
	fs3															
RF	**fs1**															
	fs2															
	fs3															
KNN	**fs1**															
	fs2															
	fs3															
NB	**fs1**															
	fs2															
	fs3															
MLP	**fs1**															
	fs2	X	X	X	X	X	X	X	X	X	X	X	X	X		
	fs3	X	X	X	X	X	X	X	X	X	X	X	X	X		

FIGURE J.13: McNemar test results ($alpha < 0.05$) for malware version 8 (X indicates statistical significant difference)

Class.	fs	Ada			RF			KNN			NB			MLP		
		fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3	fs1	fs2	fs3
Ada	**fs1**															
	fs2	X														
	fs3	X														
RF	**fs1**		X	X												
	fs2	X			X											
	fs3	X			X											
KNN	**fs1**		X	X		X	X									
	fs2	X			X			X								
	fs3	X			X			X								
NB	**fs1**		X	X		X	X		X	X						
	fs2		X	X		X	X		X	X						
	fs3			X		X	X	X	X							
MLP	**fs1**		X	X		X	X		X	X						
	fs2		X	X		X	X		X	X						
	fs3		X	X		X	X		X	X						

FIGURE J.14: McNemar test results ($alpha < 0.05$) for malware version 9 (X indicates statistical significant difference)

Bibliography

[1] T. Hagoort. (). Samsung is leading smartphone brand with 859 million active devices at end of 2016. Accessed at: 2017-10, [Online]. Available: https://newzoo.com/insights/articles/newzoo-global-mobile-market-report-samsung-is-leading-smartphone-brand-with-859-million-active-devices/.

[2] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques", *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, p. 76, 2017.

[3] P. Yan and Z. Yan, "A survey on dynamic mobile malware detection", *Software Quality Journal*, pp. 1–29, 2017.

[4] "Global smartphone market share 2016", Business Insider, 2016. [Online]. Available: http://www.businessinsider.com/smartphone-market-share-android-ios-windows-blackberry-2016-8?international=true&r=US&IR=T.

[5] Y. Mirsky, A. Shabtai, L. Rokach, B. Shapira, and Y. Elovici, "Sherlock vs moriarty: A smartphone dataset for cybersecurity research", in *Proceedings of the 2016 ACM workshop on Artificial intelligence and security*, ACM, 2016, pp. 1–12.

[6] Google. (). Android security 2016 year in review. Accessed at: 2017-11, [Online]. Available: https://source.android.com/security/reports/Google_Android_Security_2016_Report_Final.pdf.

[7] F. Martinelli, F. Marulli, and F. Mercaldo, "Evaluating convolutional neural network for effective mobile malware detection", *Procedia Computer Science*, vol. 112, pp. 2372–2381, 2017.

[8] G. Piatetsky. (). Crisp-dm, still the top methodology for analytics, data mining, or data science projects. Accessed at: 2017-10, [Online]. Available: https://www.kdnuggets.com/2014/10/crisp-dm-top-methodology-analytics-data-mining-data-science-projects.html.

[9] Google. (). Android security 2016 year in review. Accessed at: 2017-11, [Online]. Available: https://source.android.com/security/reports/Google_Android_Security_2016_Report_Final.pdf.

[10] Kaspersky. (). It threat evolution q1 2017. Accessed at: 2017-11, [Online]. Available: https://securelist.com/it-threat-evolution-q1-2017-statistics/78475/.

[11] S. Dua and X. Du, *Data mining and machine learning in cybersecurity*. CRC press, 2016.

[12] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*. Springer series in statistics New York, 2001, vol. 1.

[13] P. Flach, *Machine learning: the art and science of algorithms that make sense of data*. Cambridge University Press, 2012.

[14] S. Russell, P. Norvig, and A. Intelligence, "A modern approach", *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, vol. 25, p. 27, 1995.

[15] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro, "Transcend: Detecting concept drift in malware classification models", in *PROCEEDINGS OF THE 26TH USENIX SECURITY SYMPOSIUM (USENIX SECURITY'17)*, USENIX Association, 2017, pp. 625–642.

[16] "White paper: Maximize mobile value: Is byod holding you back?", Samsung, Tech. Rep., Jun. 2018, Accessed at: 2018-06. [Online]. Available: https://s7d2.scene7.com/is/content/SamsungUS/samsungbusiness/short-form/maximizing-mobile-value/WHP-HHP-MAXIMIZE-MOBILE-VALUE-JUN18.pdf.

[17] "The economic risk of confidential data on mobile devices in the workplace", Ponemon, Tech. Rep., Feb. 2016, Accessed at: 2018-06. [Online]. Available: https://info.lookout.com/rs/051-ESQ-475/images/Ponemon%20Report%20Enterprise%20FINAL.pdf.

[18] E. Parliament. (). What does the general data protection regulation (gdpr) govern? Accessed at: 2017-10, [Online]. Available: https://ec.europa.eu/info/law/law-topic/data-protection/reform/what-does-general-data-protection-regulation-gdpr-govern_en.

[19] "Revisit your enterprise mobility management practices to prepare for eu gdpr", Tech. Rep., Accessed at: 2017-10. [Online]. Available: https://www.gartner.com/doc/3709317/revisit-enterprise-mobility-management-practices.

[20] J. Aisien. (). He impact of gdpr on today's mobile enterprise. Accessed at: 2017-07, [Online]. Available: https://www.scmagazine.com/the-impact-of-gdpr-on-todays-mobile-enterprise/article/710019/.

[21] N. Keller, "Cybersecurity framework", 2013.

[22] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and A. Ribagorda, "Evolution, Detection and Analysis of Malware for Smart Devices", *IEEE Communications Surveys Tutorials*, vol. 16, no. 2, pp. 961–987, 2014, ISSN: 1553-877X. DOI: 10.1109/SURV.2013.101613.00077.

[23] A. E. Attar, R. Khatoun, and M. Lemercier, "A Gaussian mixture model for dynamic detection of abnormal behavior in smartphone applications", in *2014 Global Information Infrastructure and Networking Symposium (GIIS)*, Sep. 2014, pp. 1–6. DOI: 10.1109/GIIS.2014.6934278.

[24] B. Amos, H. Turner, and J. White, "Applying machine learning classifiers to dynamic Android malware detection at scale", in *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, Jul. 2013, pp. 1666–1671. DOI: 10.1109/IWCMC.2013.6583806.

[25] B. Dixon and S. Mishra, "Power Based Malicious Code Detection Techniques for Smartphones", in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, Jul. 2013, pp. 142–149. DOI: 10.1109/TrustCom.2013.22.

[26] M. S. Alam and S. T. Vuong, "Random Forest Classification for Detecting Android Malware", in *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, Aug. 2013, pp. 663–669. DOI: 10.1109/GreenCom-iThings-CPSCom.2013.122.

[27] S. B. Almin and M. Chatterjee, "A Novel Approach to Detect Android Malware", *Procedia Computer Science*, International Conference on Advanced Computing Technologies and Applications (ICACTA), vol. 45, no. Supplement C, pp. 407–417, Jan. 2015, ISSN: 1877-0509. DOI: 10.1016/j.procs.2015.03.170. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050915004135 (visited on 10/19/2017).

[28] S. Sheen, R. Anitha, and V. Natarajan, "Android based malware detection using a multifeature collaborative decision fusion approach", *Neurocomputing*, vol. 151, no. Part 2, pp. 905–912, Mar. 2015, ISSN: 0925-2312. DOI: 10.1016/j.neucom.2014.10.004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231214012739 (visited on 10/19/2017).

[29] L. Yang, V. Ganapathy, and L. Iftode, "Enhancing Mobile Malware Detection with Social Collaboration", in *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*, Oct. 2011, pp. 572–576. DOI: 10.1109/PASSAT/SocialCom.2011.176.

[30] J. Abawajy and A. Kelarev, "Iterative Classifier Fusion System for the Detection of Android Malware", *IEEE Transactions on Big Data*, vol. PP, no. 99, pp. 1–1, 2017. DOI: 10.1109/TBDATA.2017.2676100.

[31] B. Holland, T. Deering, S. Kothari, J. Mathews, and N. Ranade, "Security Toolbox for Detecting Novel and Sophisticated Android Malware", in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15, Piscataway, NJ, USA: IEEE Press, 2015, pp. 733–736. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819151 (visited on 10/19/2017).

[32] T. E. Wei, C. H. Mao, A. B. Jeng, H. M. Lee, H. T. Wang, and D. J. Wu, "Android Malware Detection via a Latent Network Behavior Analysis", in *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, Jun. 2012, pp. 1251–1258. DOI: 10.1109/TrustCom.2012.91.

[33] D.-F. Guo, A.-F. Sui, Y.-J. Shi, J.-J. Hu, G.-Z. Lin, and T. Guo, "Behavior Classification based Self-learning Mobile Malware Detection.", *JCP*, vol. 9, no. 4, pp. 851–858, 2014.

[34] B. Cui, H. Jin, G. Carullo, and Z. Liu, "Service-oriented mobile malware detection system based on mining strategies", *Pervasive and Mobile Computing*, Special Issue on Secure Ubiquitous Computing, vol. 24, no. Supplement C, pp. 101–116, Dec. 2015, ISSN: 1574-1192. DOI: 10.1016/j.pmcj.2015.06.006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574119215001133 (visited on 10/19/2017).

[35] P. S. Chen, S.-C. Lin, and C.-H. Sun, "Simple and effective method for detecting abnormal internet behaviors of mobile devices", *Information Sciences*, Security and privacy information technologies and applications for wireless pervasive computing environments, vol. 321, no. Supplement C, pp. 193–204, Nov. 2015, ISSN: 0020-0255. DOI: 10.1016/j.ins.2015.04.035. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025515003175 (visited on 10/19/2017).

[36] R. Andriatsimandefitra and V. V. T. Tong, "Detection and Identification of Android Malware Based on Information Flow Monitoring", in *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*, Nov. 2015, pp. 200–203. DOI: 10.1109/CSCloud.2015.27.

[37] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones", *ACM Trans. Comput. Syst.*, vol. 32, no. 2, 5:1–5:29, Jun. 2014, ISSN: 0734-2071. DOI: 10.1145/2619091. [Online]. Available: http://doi.acm.org/10.1145/2619091 (visited on 10/19/2017).

[38] L. Caviglione, M. Gaggero, J. F. Lalande, W. Mazurczyk, and M. Urbański, "Seeing the Unseen: Revealing Mobile Malware Hidden Communications via Energy Consumption and Artificial Intelligence", *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 4, pp. 799–810, Apr. 2016, ISSN: 1556-6013. DOI: 10.1109/TIFS.2015.2510825.

[39] S. H. Hung, C. H. Tu, and C. W. Yeh, "A Cloud-Assisted Malware Detection Framework for Mobile Devices", in *2016 International Computer Symposium (ICS)*, Dec. 2016, pp. 537–542. DOI: 10.1109/ICS.2016.0112.

[40] F. Tong and Z. Yan, "A hybrid approach of mobile malware detection in Android", *Journal of Parallel and Distributed Computing*, Special Issue on Scalable Cyber-Physical Systems, vol. 103, no. Supplement C, pp. 22–31, May 2017, ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2016.10.012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S074373151630140X (visited on 10/19/2017).

[41] D. V. Ng and J. I. G. Hwang, "Android malware detection using the dendritic cell algorithm", in *2014 International Conference on Machine Learning and Cybernetics*, vol. 1, Jul. 2014, pp. 257–262. DOI: 10.1109/ICMLC.2014.7009126.

[42] F. Martinelli, F. Mercaldo, and A. Saracino, "BRIDEMAID: An Hybrid Tool for Accurate Detection of Android Malware", in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17, New York, NY, USA: ACM, 2017, pp. 899–901, ISBN: 978-1-4503-4944-4. DOI: 10.1145/3052973.3055156. [Online]. Available: http://doi.acm.org/10.1145/3052973.3055156 (visited on 10/19/2017).

[43] D. Quan, L. Zhai, F. Yang, and P. Wang, "Detection of Android Malicious Apps Based on the Sensitive Behaviors", in *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, Sep. 2014, pp. 877–883. DOI: 10.1109/TrustCom.2014.115.

[44] T. Wuechner, A. Cislak, M. Ochoa, and A. Pretschner, "Leveraging Compression-based Graph Mining for Behavior-based Malware Detection", *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99, pp. 1–1, 2017, ISSN: 1545-5971. DOI: 10.1109/TDSC.2017.2675881.

[45] M. Sun, X. Li, J. C. S. Lui, R. T. B. Ma, and Z. Liang, "Monet: A User-Oriented Behavior-Based Malware Variants Detection System for Android", *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 5, pp. 1103–1112, May 2017, ISSN: 1556-6013. DOI: 10.1109/TIFS.2016.2646641.

[46] 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*, vol. 11, no. 2, pp. 289–302, Feb. 2016, ISSN: 1556-6013. DOI: 10.1109/TIFS.2015.2491300.

[47] M. Rahman, "DroidMLN: A Markov Logic Network Approach to Detect Android Malware", in *2013 12th International Conference on Machine Learning and Applications*, vol. 2, Dec. 2013, pp. 166–169. DOI: 10.1109/ICMLA.2013.184.

[48] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou, "Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS", in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 357–376. DOI: 10.1109/SP.2016.29.

[49] X. Liao, K. Yuan, X. Wang, Z. Pei, H. Yang, J. Chen, H. Duan, K. Du, E. Alowaisheq, S. Alrwais, L. Xing, and R. Beyah, "Seeking Nonsense, Looking for Trouble: Efficient Promotional-Infection Detection through Semantic Inconsistency Search", in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 707–723. DOI: 10.1109/SP.2016.48.

[50] T. Shen, Y. Zhongyang, Z. Xin, B. Mao, and H. Huang, "Detect Android Malware Variants Using Component Based Topology Graph", in *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, Sep. 2014, pp. 406–413. DOI: 10.1109/TrustCom.2014.52.

[51] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "Emulator vs real phone: Android malware detection using machine learning", in *Proceedings of the 3rd ACM on International Workshop on Security And Privacy Analytics*, ACM, 2017, pp. 65–72.

[52] A. Feizollah, N. B. Anuar, R. Salleh, and A. W. A. Wahab, "A review on feature selection in mobile malware detection", *Digital Investigation*, vol. 13, pp. 22–37, 2015.

[53] 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*, vol. 38, no. 1, pp. 161–190, 2012.

[54] H.-S. Ham and M.-J. Choi, "Analysis of android malware detection performance using machine learning classifiers", in *ICT Convergence (ICTC), 2013 International Conference on*, IEEE, 2013, pp. 490–495.

[55] B. Dixon, S. Mishra, and J. Pepin, "Time and location power based malicious code detection techniques for smartphones", in *Network Computing and Applications (NCA), 2014 IEEE 13th International Symposium on*, IEEE, 2014, pp. 261–268.

[56] M. Guri, G. Kedma, B. Zadov, and Y. Elovici, "Trusted detection of sensitive activities on mobile phones using power consumption measurements", in *Intelligence and Security Informatics Conference (JISIC), 2014 IEEE Joint*, IEEE, 2014, pp. 145–151.

[57] 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*, 2016.

[58] J. Milosevic, A. Ferrante, and M. Malek, "Malaware: Effective and efficient run-time mobile malware detector", in *Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), 2016 IEEE 14th Intl C*, IEEE, 2016, pp. 270–277.

[59] A. Ferrante, E. Medvet, F. Mercaldo, J. Milosevic, and C. A. Visaggio, "Spotting the malicious moment: Characterizing malware behavior using dynamic features", in *Availability, Reliability and Security (ARES), 2016 11th International Conference on*, IEEE, 2016, pp. 372–381.

[60] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Acquiring and analyzing app metrics for effective mobile malware detection", in *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, ACM, 2016, pp. 50–57.

[61] L. Massarelli, L. Aniello, C. Ciccotelli, L. Querzoni, D. Ucci, and R. Baldoni, "Android malware family classification based on resource consumption over time", *arXiv preprint arXiv:1709.00875*, 2017.

[62] 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*, ACM, 2011, pp. 15–26.

[63] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket.", in *NDSS*, 2014.

[64] (). Malware genome project. Accessed at: 2017-10, [Online]. Available: http://www.malgenomeproject.org/.

[65] (). Virusshare. Accessed at: 2017-10, [Online]. Available: http://virusshare.com.

[66] (). Virustotal database. Unaccessible at: 2017-10, [Online]. Available: https://secure.vt-mis/.

[67] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, "Madam: A multi-level anomaly detector for android malware", *Computer network security*, pp. 240–253, 2012.

[68] (). Contagio. Accessed at: 2017-10, [Online]. Available: http://contagiominidump.blogspot.com.

[69] AVTest. (). The best antivirus software for android september 2017. Accessed at: 2017-11, [Online]. Available: https://www.av-test.org/en/antivirus/mobile-devices/.

[70] T. G. Dietterich, "Approximate statistical tests for comparing supervised classification learning algorithms", *Neural computation*, vol. 10, no. 7, pp. 1895–1923, 1998.

[71] J. Drew, "Managing cybersecurity risks", *Journal of accountancy*, vol. 214, no. 2, pp. 44–48, 2012.

[72] S. A. Deutscher, W. Bohmayr, and A. Asen, *Building a cyberresilient organization*, 2017.

[73] (). Entrepreneurship and small and medium-sized enterprises (smes). Accessed at: 2018-07, [Online]. Available: http://ec.europa.eu/growth/smes_en.

[74] R. Chomko. (). Maintaining an app is critical to its overall success. Accessed at: 2018-07, [Online]. Available: https://www.fiercewireless.com/developer/maintaining-app-critical-to-its-overall-success.

[75] R. Patil. (). What affects the cost of mobile app maintenance services? Accessed at: 2018-07, [Online]. Available: https://medium.com/@ritesh.patil732/what-affects-the-cost-of-mobile-app-maintenance-services-4fa330c9278d.

[76] D. Diehl. (). How much does a mobile app maintenance contract cost? Accessed at: 2018-07, [Online]. Available: https://www.seguetech.com/how-much-mobile-app-maintenance-contract-cost/.

[77] (). Cyber security specialist salary. Accessed at: 2018-07, [Online]. Available: https://www.ziprecruiter.com/Salaries/Cyber-Security-Specialist-Salary.

[78] V. Beal. (). Webopedia - what is mobile device management? Accessed at: 2018-07, [Online]. Available: https://www.webopedia.com/TERM/M/mobile_device_management.html.

[79] J. H. Friedman, "Greedy function approximation: A gradient boosting machine", *Annals of statistics*, pp. 1189–1232, 2001.

[80] "Android application security", Google, 2017. [Online]. Available: https://source.android.com/security/overview/app-security.

[81] "Android platform security", Google, 2017. [Online]. Available: https://source.android.com/security/#platform-security-architecture.

[82] "Android platform architecture", Google, 2017. [Online]. Available: https://developer.android.com/guide/platform/index.html.