



# DEVELOPING A NEXT-GENERATION MOBILE SECURITY SOLUTION FOR ANDROID

**Paolo Rovelli**

Master of Science

Software Engineering

April 2014

School of Computer Science

Reykjavík University

**M.Sc. RESEARCH THESIS**





# **Developing a next-generation Mobile Security solution for Android**

by

Paolo Rovelli

Research thesis submitted to the School of Computer Science  
at Reykjavík University in partial fulfillment of  
the requirements for the degree of  
**Master of Science in Software Engineering**

April 2014

Research Thesis Committee:

Ýmir Vigfússon, Supervisor  
PhD, Assistant Professor, Reykjavík University

Gianfranco Tonello, Co-Supervisor  
MSc, CEO, TG Soft S.a.s.

Marjan Sirjani  
PhD, Associate Professor, Reykjavík University

Copyright  
Paolo Rovelli  
April 2014

# Developing a next-generation Mobile Security solution for Android

Paolo Rovelli

April 2014

## Abstract

The exponential growth of the Android platform in the recent years has made it a main target of cyber-criminals. As a result, the amount of malware for Android is constant and rapidly growing ([F-Secure, 2013](#); [Panda Security, 2013b](#); [Sophos, 2013b](#); [G Data Software, 2013](#)). This exponential growth of malware given, there is a need for new detection models designed to specifically target Android malware in order to better protect the end-users and, eventually, to counter the rise of Android malware itself.

We strongly believe that, before starting to address a problem, we firstly need to understand it deeply. Thus, in this work of thesis, we firstly investigate the current state-of-the-art of Android and Android malware, presenting a classification and characterisation of the current “in the wild” Android malware. Afterwards, we investigate possible detection models that can be applied to secure Android devices from the major classes of threats. As a result, we present *VirIT Mobile Security*, a mobile security solution specifically designed and developed to counter Android malware. VirIT Mobile Security has been designed and developed in collaboration with and commercialised by TG Soft, an Italian antivirus firm. In particular, we propose two different approaches together, the first being a reactive approach done with the use of a signature-based detection model and used to detect whether a mobile device is infected or not. The latter one, instead, is a proactive approach used to spot zero-day or next-generation malware as they emerge.

As to test VirIT Mobile Security, we use different experiments, one for each detection mechanism developed. Experimental results show that our signature-based detection system was able to properly detect and remove 95.95% of the malware in our test, while our Permission-based Malware Detection System (PMDS) was able to detect more than 94% of previously unseen malware. Finally, our behavioural detection system was able to spot and report back to the Anti-Malware Research Center of TG Soft (CRAM) 75% of the zero-day or next-generation malware in our test.



*A Andrea, mi musa y amor.*





# Acknowledgements

I do not remember when exactly I started to get interested in computer security. Perhaps, I always was. I think that, at the very beginning, it was the idea of protecting the weak and the helpless. Only after, I have discovered how deep and challenging is the “rabbit hole”.

I do not see this work of thesis as the end of anything, but rather as a first step and achievement in the field of computer security.

All this would not have been possible without the wonderful people that have filled my life and that I wish to thank.

The first thanks are for my parents, for giving me the amazing opportunity of studying in a sparkling country as Iceland, and my girlfriend, for unconditionally supporting and encouraging me.

A special thanks to all the friends who walked alongside me, the childhood ones from Monza as well as the ones I have met during my studies both in Italy and in Iceland, and the ones I have met during my period in TG Soft.

Thanks to Ýmir Vigfússon and to Gianfranco and Enrico Tonello for giving me the opportunity of working in a university-industry joint work.

The last thanks is for all the researchers who have enlightened my path towards computer security.

This is just the beginning.

*Paolo*



# Publications

During the period of this work of thesis, the author was a guest speaker in the “*Conto corrente sotto attacco: come l’evoluzione dei Trojan Banker minaccia i nostri soldi*” workshop at **SMAU Padova 2014**, where he spoke about Android Trojan-Banker, repackaging (see Chapter 2) and possible countermeasures. (CRAM, 2014; Tonello, 2014)

Furthermore, the following articles and malware analysis have been published on the web:

- Analysis of *Trojan://Android/FakeMarket.A* and on the safety of Google Play store: (Rovelli, 2014)
- Analysis of *Trojan-Spy://Android/SMSAgent.C*: (Rovelli, 2013b)
- Analysis of *Trojan-SMS://Android/Agent.B*: (Rovelli, 2013a)
- Analysis of *Trojan-Banker://Android/ZitMo*: (Rovelli, 2013e)
- Analysis of *Rogue-AV://Android/AndroidDefeder.A*: (Rovelli, 2013c)
- Analysis of *Trojan://Android/SMSAgent.A*: (Rovelli & Tonello, 2013)
- Wi-Fi Security: (Rovelli, 2013d)



# Contents

<b>List of Figures</b>	<b>xiv</b>
<b>List of Tables</b>	<b>xxiv</b>
<b>List of Abbreviations</b>	<b>xxvii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Android and Android malware</b>	<b>7</b>
2.1 Android . . . . .	7
2.1.1 The Android Operating System . . . . .	7
2.1.2 The APK File Format . . . . .	9
2.1.3 Google Play Store and Third-Party Markets . . . . .	10
2.2 Android Malware Taxonomy . . . . .	11
2.2.1 Malware Naming Scheme . . . . .	11
2.2.2 Malware Types . . . . .	12
2.2.3 Malware Propagation Methodologies . . . . .	17
2.3 Related Work . . . . .	22
2.4 Conclusion . . . . .	23
<b>3 Implementing a Signature-based Malware Detection System in Android</b>	<b>25</b>
3.1 Introduction . . . . .	25
3.2 Related Work . . . . .	27
3.3 Design . . . . .	28
3.3.1 Malware Signatures . . . . .	29
3.3.2 Signatures Database . . . . .	30
3.3.3 Signatures Extraction and Signature Matching Engine . . . . .	31
3.4 Implementation . . . . .	32
3.4.1 Signature Matching Engine . . . . .	33
3.4.2 On-Demand Scanner . . . . .	37

3.4.3	On-Install Scanner . . . . .	40
3.4.4	On-Access Scanner . . . . .	43
3.5	Evaluation . . . . .	45
3.5.1	Environment . . . . .	46
3.5.2	Results . . . . .	46
3.6	Conclusion . . . . .	47
<b>4</b>	<b>Implementing a Real-Time Monitor in Android</b>	<b>49</b>
4.1	Introduction . . . . .	49
4.2	Design and Implementation . . . . .	50
4.2.1	On-Execution Scanner . . . . .	50
4.2.2	On-Update Scanner . . . . .	52
4.2.3	Live Scanner . . . . .	52
4.2.4	Cloud Scanner . . . . .	54
4.3	Evaluation . . . . .	56
4.3.1	Environment . . . . .	57
4.3.2	Results . . . . .	58
4.4	Related Work . . . . .	60
4.5	Conclusion . . . . .	61
<b>5</b>	<b>PMDS: Permission-based Malware Detection System for Android</b>	<b>63</b>
5.1	Introduction . . . . .	63
5.2	Related Work . . . . .	64
5.3	Design . . . . .	65
5.3.1	Permissions as (possible) behavioural markers . . . . .	65
5.3.2	Custom Permissions . . . . .	67
5.3.3	PMDS Classifier . . . . .	67
5.4	Implementation . . . . .	68
5.5	Evaluation . . . . .	71
5.5.1	Environment . . . . .	71
5.5.2	Classification with standard machine learning algorithms . . . . .	72
5.5.3	Classification with boosted machine learning algorithms . . . . .	75
5.6	Conclusions . . . . .	76
<b>6</b>	<b>Implementing Network Protection and a Behavioural Detection System in Android</b>	<b>79</b>
6.1	Introduction . . . . .	79
6.2	Web Filter . . . . .	80

6.3	Network Monitor . . . . .	82
6.4	Community Network and Behavioural Detection . . . . .	83
6.5	Evaluation . . . . .	84
6.5.1	Results . . . . .	85
6.6	Conclusion . . . . .	86
7	Conclusions and Future Work	89
	Bibliography	93





# List of Figures

1.1	Top mobile Operating Systems on September 2013 (StatCounter.com). As one can see, Android is the most widespread mobile platform in the world. . . . .	1
1.2	New mobile threat families and variants from Q1 2012 to Q1 2013 (F-Secure) (F-Secure, 2013). The exponential growth in mobile devices adoption, together with the wealth of sensitive information carried with them (e.g. personal and bank information, GPS location, emails, etc...), have inevitably driven to the evolution of malware targeting mobile platforms. Many other factors have contributed to make mobile malware are one of the greatest computer threats in our time. . . . .	2
1.3	Number of Android malware samples in the AV-TEST database between January 2011 and June 2013 (AV-TEST) (AV-TEST, 2013). The total number of Android malware samples was expected to exceed 1 million in the Summer of 2013. . . . .	3
1.4	Android Threat Exposure Rate (TER) in 2013 (Sophos) (Sophos, 2013b). The TER is measured as the percentage of PCs and Android devices that experienced a malware attack, whether successful or failed. The experiment was run over a period of three month. . . . .	3
2.1	Android Operating System architecture. Android is a multi-user Linux system where each application is a different user and runs in its own Linux process, with its own instance of the Dalvik Virtual Machine. Therefore, an application's code runs in isolation from other applications. The system architecture is composed of five abstract layers: Linux kernel, native libraries, Android runtime, application framework and applications. . . .	8

2.2	The APK package structure. Android applications use the Android package (APK) file format, which is an archive file built on the ZIP file format and contains: the <i>classes.dex</i> executable file, which represents the set of all the Java classes compiled in the Dalvik EXecutable (dex) file format understandable by the Dalvik Virtual Machine, the <i>AndroidManifest.xml</i> file, which provides semantic-rich information about the application itself and its components, the <i>resources.arsc</i> file, which describes the pre-compiled resources, the <i>lib</i> directory, which contains the compiled code that is specific to a software layer of a processor, the <i>assets</i> and <i>res</i> directories, which contains the application assets and resources respectively, and the <i>META-INF</i> directory, which contains the digital certificate with which the application was signed and the signatures of all the files within the APK package. (The Android Open Source Project, n.d.) . . . . .	9
2.3	The list of files within the APK package of a malicious application ( <i>Trojan-Banker://Android/ZitMo.B</i> ). . . . .	10
2.4	The Android malware types. Most of the existing Android malware types are directly inherited from the desktop space (e.g. Adware or Rogue-AV), even if some of them have additional capabilities due to the mobile space. However, there are also others Android malware types that are unique to mobile space (e.g. Trojan-SMS). . . . .	12
2.5	The Android malware types in percentage, for both the number of families (on the left) and variants (on the right). The data refer to a dataset of 1500 samples, taken from the Android Malware Genome Project (Y. Zhou & Jiang, 2012), Contagio Mobile (Mila, 2013) and other on-line sources. The collected samples were later subdivided by type, obtaining a number of 826 samples of Trojan (divided into 33 families), 359 samples of Backdoor (divided into 17 families), 98 samples of Trojan-Spy (divided into 25 families) and 95 samples of Trojan-SMS (divided into 24 families) among the others. . . . .	13

2.6	Ad libraries' behaviours and their levels of severity in 2013 (Symantec). In Android, many applications are distributed or have a free version which includes in-app advertisements. Some ad libraries (defined as "aggressive"), however, are able to leak personal and sensitive information or exhibit annoying behaviours. In this graph, the ad libraries' behaviours are divided accordingly to their aggressiveness. In <i>low severity</i> (green) are grouped all those ad libraries which display ads inside the applications and do not leak any data. In <i>medium severity</i> (yellow) are grouped all those ad libraries which leaks non-harmful data (e.g. location or mobile network). In <i>high severity</i> (red) are grouped all those ad libraries which leak private data (e.g. phone number or user account) or annoy/lure the user (e.g. show ads in the notification bar or play a voice ad when making a phone call). According to Symantec, in 2013, there were 65 known ad libraries of which over 50% of them were classified as aggressive. (Uscilowski, 2013) . . . . .	14
2.7	On the left a screenshot of <i>Rogue-AV://Android/AndroidDefeder.A</i> , which claims to have found 6 malware on the device and asks to purchase a licence in order to remove them, while on the right a screenshot of <i>Trojan-SMS://Android/Agent.B</i> , which pretends to be a version of Microsoft Internet Explorer for Android but subscribes the user to paid services automatically answering to specific SMS messages. . . . .	15
2.8	An example of repackaging. A benign application is downloaded from an Android market (usually Google Play Store) and disassembled. A malicious payload is then injected into the application, the package is re-assembled and, finally, the new "Trojanised" version of the application is uploaded back into an alternative, third-party Android market. . . . .	20
2.9	The Android "Master Key" vulnerability allows attackers to inject malicious code into an APK package without having to digitally re-sign the application. Here, two <i>classes.dex</i> and two <i>AndroidManifest.xml</i> files are putted in the same APK package. The Android system verifies the digital signature of the first (the legitimate one) but installs the latter (the malicious one). . . . .	21
2.10	An example of <i>AndroidManifest.xml</i> file in which it is included the developer's publisher IDs given by an ad network ( <i>MiniMob</i> in this example). Cyber-criminals can replace the original publisher IDs with their own in order to steal the ad revenues. . . . .	22

3.1	The home screen of VirIT Mobile Security. The Activity changes dynamically, displaying the relative security-related messages (picture on the left).	26
3.2	The architecture of the signature-based detection system in VirIT Mobile Security. Thanks to the on-install scanner, the applications installed from Google Play or other third-party markets (see Seciton 2.1.3) are automatically scanned (after the installation) by the signatures matching engine (see Seciton 3.4.3). The application installed from the SD Card, instead, can be also scanned before the installation. Thanks to the on-demand scanner, the user can always decide to launch a scan on the installed applications and on the External Storage (see Seciton 3.4.2). Finally, thanks to the on-access scanner, each file on the External Storage is scanned every time it is written (see Seciton 3.4.4). For the overall VirIT Mobile Security architecture see Figure 6.1 . . . . .	29
3.3	The on-demand scanner of VirIT Mobile Security. A scan is performed before on the installed applications and after on the External Storage. In case some malicious applications are found, the “Solve” button will appear in order to show to the user the list of threats found (see Figure 3.4) .	38
3.4	The threats report of the on-demand scanner of VirIT Mobile Security (see Figure 3.3). Before are listed the (malicious) installed applications then the files in the External Storage. By tapping on a specific threat, an user can see more information about that particular threat or signal it to the Anti-Malware Research Center of TG Soft (CRAM) as a (possible) false positive. . . . .	40
3.5	The on-install scanner of VirIT Mobile Security, which scans the applications before they are installed from the External Storage. In the pictures, the on-install scanner detects the installation of the <i>Trojan://Android/FakeMarket.A</i> , which pretends to be the Google Play store application but runs silently in the background to perform click fraud. (Rovelli, 2014; AndroTotal, 2014)	41
3.6	The on-install scanner of VirIT Mobile Security, which scans the applications at installation. In the picture on the right, the on-install scanner detects the installation of the <i>Trojan://Android/FakeMarket.A</i> , which pretends to be the Google Play store application but runs silently in the background to perform click fraud. (Rovelli, 2014; AndroTotal, 2014) . . . . .	43

3.7	The on-access scanner of VirIT Mobile Security, which scans the files in the External Storage at creation and any modification. In the picture on the right, the on-access scanner detects that the <i>Trojan://Android/SpySMS.A</i> has been written in the External Storage (precisely in the SD Card root directory: <i>/storage/sdcard</i> ). . . . .	45
3.8	Testing the robustness of the signature-based detection system in VirIT Mobile Security. With the term robustness we mean the ability of our solution to prevent Android malware from penetrating a device at all. We tested VirIT Mobile Security with a dataset of 1200 samples, 148 labelled as malicious applications and 1052 labelled as benign ones. . . . .	47
4.1	A possible cloud scanner implementation is the one of dividing the so called Cloud Device Simulator (CDS), which will contain the virtual representations of the subscribed devices, and the Cloud App Repository (CAR), which will contain all the APK packages of known Android applications. Basically, a CDS instance contains the list of references of the applications installed on the real device at any moment in time. The scanner will scan all the applications installed by each CDS instance, retrieving the actual APK packages from the CAR. When a malware is detected on a CDS instance, this will be immediately reported to the corresponding real device. . . . .	55
4.2	Live Scanner: Number of updates needed to catch one malware in the 100.000 simulations with the configuration of the live scanner at 20%. In a situation in which we release one update a day, this is equal to the number of days needed to catch one malware. The complete results are shown in Table 4.2. . . . .	58
4.3	Live Scanner: Average number of updates/scans needed to catch at least one malware in relation to the percentage of application scanned for each update. The various lines show the number of malware in the device, from 1 (blue line) to 5 (light blue line). In a situation in which we release one update a day, the number of updates is equal to the number of days needed to catch one malware. The complete results are shown in Table 4.2. . . . .	59
4.4	Live Scanner: Highest number of updates/scans needed to catch at least one malware in relation to the percentage of application scanned for each update. The various lines show the number of malware in the device, from 1 (blue line) to 5 (light blue line). In a situation in which we release one update a day, the number of updates is equal to the number of days needed to catch one malware. The complete results are shown in Table 4.2. . . . .	60

- 5.1 The Privacy Advisor of VirIT Mobile Security, which makes use of the PMDS rule-based (RIPPER) classifier in order to retrieve and display the potentially dangerous applications installed on the device. As shown in the picture on the right, an application can be manually whitelisted (so that it will not display as potentially dangerous in future scans), scanned with the on-demand scanner (see Section 3.4.2) and/or sent to the Anti-Malware Research Center of TG Soft (CRAM) for the analysis. In the picture at the centre, the Privacy Advisor reports both the *Trojan-Spy://Android/Wapsx.A* (the first of the list) and the *Trojan-SMS://Android/FakeFlappyBird.A* (the second of the list) as possibly dangerous. . . . . 64
- 5.2 Example of a malicious application (*Trojan-Banker://Android/ZitMo.B*) which requires for a specific group of permissions. On the right the permissions are required during the installation process, while on the left the *AndroidManifest.xml* file in which the required permissions are declared. . 66
- 5.3 The architecture of the antivirus in VirIT Mobile Security. The applications are firstly scanned by the signature-based detection system and afterwards, if no signature matches them, they are scanned by the Permission-based Malware Detection System (PMDS) rule-based classifier in order to detect possible zero-day or next-generation malware. For the overall VirIT Mobile Security architecture see Figure 6.1 . . . . . 67
- 5.4 The “heuristic analysis on permissions” of the on-install scanner of VirIT Mobile Security (see Figure 3.6), which makes use of the PMDS rule-based (RIPPER) classifier in order to alert if potentially dangerous applications are installed on the device. In the first two pictures (on the left), the on-install scanner detects the installation of the *Trojan-Spy://Android/Wapsx.A* and the *Trojan-SMS://Android/FakeFlappyBird.A* respectively. In the last picture (on the right) it is shown that applications can be manually whitelisted, so that they will not display as potentially dangerous in future scans. . . . 68
- 5.5 The Permission-based Malware Detection System (PMDS) architecture. The permissions declared in the *AndroidManifest.xml* file of an application are automatically extracted using the Android Asset Packaging Tool (aapt). Then, the classifier automatically labels the application behaviour, as either benign or (potentially) malicious, according to the combination of permissions the application requires. . . . . 69

- 5.6 Count of the most frequently requested permissions by the samples (both benign and malicious) in our dataset of 2950 samples - 1500 benign and 1450 malicious. The blue lines show the number of times the specific permissions have been requested by benign applications, while the red lines show the number of times they have been requested by malicious applications. . . . . 72
- 5.7 The Receiver Operating Characteristic (ROC) Curve of our J48 (left) and K\* (right) classifiers respectively. The Area Under the Curve (AUC) represents the probability that a classifier will rank a randomly chosen *positive* instance higher than a randomly chosen *negative* one. The colour of the curve depicts the value of the threshold (i.e. closer to blue corresponds to the lower threshold value). Indeed, each point in the curve illustrates a prediction tradeoff that can be obtained by varying the threshold value between classes. Or, in other words, every point corresponds to setting a threshold on the probability assigned to the *positive* class. The typical threshold value of 0.5 means the predicted probability of *positive* must be higher than 0.5 for the instance to be predicted as *positive*. . . . . 73
- 5.8 The Receiver Operating Characteristic (ROC) Curve of our RIPPER (left) and Näive Bayes (right) classifiers respectively. The Area Under the Curve (AUC) represents the probability that a classifier will rank a randomly chosen *positive* instance higher than a randomly chosen *negative* one. The colour of the curve depicts the value of the threshold (i.e. closer to blue corresponds to the lower threshold value). Indeed, each point in the curve illustrates a prediction tradeoff that can be obtained by varying the threshold value between classes. Or, in other words, every point corresponds to setting a threshold on the probability assigned to the *positive* class. The typical threshold value of 0.5 means the predicted probability of *positive* must be higher than 0.5 for the instance to be predicted as *positive*. . . . . 73

- 5.9 The Receiver Operating Characteristic (ROC) Curve of our AdaBoost classifier using J48 as base classifier. The Area Under the Curve (AUC) represents the probability that a classifier will rank a randomly chosen *positive* instance higher than a randomly chosen *negative* one. The colour of the curve depicts the value of the threshold (i.e. closer to blue corresponds to the lower threshold value). Indeed, each point in the curve illustrates a prediction tradeoff that can be obtained by varying the threshold value between classes. Or, in other words, every point corresponds to setting a threshold on the probability assigned to the *positive* class. The typical threshold value of 0.5 means the predicted probability of *positive* must be higher than 0.5 for the instance to be predicted as *positive*. . . . . 76
- 5.10 The Receiver Operating Characteristic (ROC) Curve of our AdaBoost classifier using RIPPER (left) and Naïve Bayes (right) as base classifiers respectively. The Area Under the Curve (AUC) represents the probability that a classifier will rank a randomly chosen *positive* instance higher than a randomly chosen *negative* one. The colour of the curve depicts the value of the threshold (i.e. closer to blue corresponds to the lower threshold value). Indeed, each point in the curve illustrates a prediction tradeoff that can be obtained by varying the threshold value between classes. Or, in other words, every point corresponds to setting a threshold on the probability assigned to the *positive* class. The typical threshold value of 0.5 means the predicted probability of *positive* must be higher than 0.5 for the instance to be predicted as *positive*. . . . . 77
- 6.1 The overall architecture of VirIT Mobile Security. Suspicious applications detected by the behavioural detection system are forwarded to the Anti-Malware Research Center of TG Soft (CRAM). If it is the case of a zero-day or next generation malware - i.e. a malicious application which is not detected by the antivirus - a proper signature is extracted and, in turn, an update of the signatures database is released. . . . . 80
- 6.2 The Web Filter of VirIT Mobile Security, which scans the URLs when surfing the web with the default Android browser and Google Chrome. Thanks to the Web Filter we can alert the user when a malicious or phishing website is opened. . . . . 82



6.3 The Network Monitor of VirIT Mobile Security, which monitors the network usage of each application. Thanks to the Network Monitor we can have an overview of the incoming and outgoing network traffic for each application and, possibly, decide if it is the case of a zero-day or next-generation malware - i.e. a malicious application which is not detected by the antivirus. In the picture, we can see the daily network usage of both the *Trojan://Android/FakeMarket.A*, which pretends to be the Google Play store application but runs silently in the background to perform click fraud (Rovelli, 2014; AndroTotal, 2014), and the *Rogue-AV://Android/AndroidDefeder.A*, which pretends to be an antivirus solution and claims to detect some threats on the victims’ devices in order to lure the victims to pay to remove the non-existent threats (Rovelli, 2013c; Ducklin, 2013). . . . . 84

6.4 Testing the resilience of VirIT Mobile Security. With the term reactive detection rate we mean the number of malware properly detected at the first scan (i.e. the robustness of the antivirus), while with the term proactive detection rate we mean the number of malware detected in a second scan (i.e. the resilience of the antivirus). . . . . 86



# List of Tables

2.1	An overview of some of the most known Android malware. For each malware, it is shown: its type, its family, the propagation method, if it tries a privilege escalation (root the device), if it connects to a Command and Control (C&C) server and in that case with which channel (e.g. SMS, HTTP or SMTP), if it leads to direct financial costs (e.g. by sending SMS messages or performing phone calls) and if it steal personal information (e.g. mobile operator, telephone number, IMEI, contacts, web history, geographical location, SMS messages and phone calls logs). . . . .	18
3.1	Results of the robustness experiments on VirIT Mobile Security. With the term robustness we mean the ability of our solution to prevent Android malware from penetrating a device at all. In the table, for all the antivirus solutions tested, we point out the True Positives (TP) (i.e. the number of malware detected out of a total of 148 malware on the device) and the corresponding detection rate. . . . .	46
4.1	An overview of the presented real-time protection mechanisms. For each model, it is shown: whether it requires significant system resources or time (i.e. resource-intensive), whether it performs more scans than the ones needed (i.e. redundancy), the detection rate on the known malware (i.e. effectiveness) and the maximum number of days a malware will last after a proper malware signature update has been released (i.e. maximum infection time frame). . . . .	57

4.2	Results of the live scanner experiments. In order to evaluate the efficiency of the live scanner, we simulate a real device with a total number of 265 installed applications. We set the live scanner in order to scan 10, 20, 30, 40 and 50% of the application after every update respectively. Then, we run the simulation 100.000 times and, finally, we calculate the average (approximated) and the highest number of scans/updates needed to catch the malware. In a situation in which we release one update a day, this is equal to the number of days needed to catch one malware. . . . .	59
5.1	Permission-based Malware Detection System (PMDS): Experimental results using four different classifiers - i.e. a <i>Decision Tree-based learner</i> (J48), a <i>Lazy (Instance-based) learner</i> (K*), a <i>Rule-based learner</i> (RIPPER) and a <i>Bayesian learner</i> (Näive Bayes) - in order to automatically label the behaviour of previously unseen applications (as either benign or malicious). The experiments are performed using the standard tenfold cross-validation, which takes 90% of the dataset for training and 10% for testing, repeating the test 10 times. . . . .	73
5.2	Permission-based Malware Detection System (PMDS): Experimental results using AdaBoost in conjunction with the previous machine learning algorithms - i.e. J48, RIPPER and Näive Bayes - in order to automatically label the behaviour of previously unseen applications (as either benign or malicious). The experiments are performed using the standard tenfold cross-validation, which takes 90% of the dataset for training and 10% for testing, repeating the test 10 times. . . . .	76
6.1	Results of the resilience experiments on VirIT Mobile Security. With the term resilience we mean the ability of our solution to recover from zero-day or next-generation malware which do manage to get installed into a device. In the table, for all the antivirus solutions tested, we point out the True Positives (TP) (i.e. the number of malware detected out of a total of 8 malware on the device) and the corresponding detection rate. . . . .	85

# List of Abbreviations

**aapt** Android Asset Packaging Tool

**ACC** Accuracy

**AdaBoost** Adaptive Boosting

**AOT** Ahead-Of-Time

**API** Application Programming Interface

**APK** Android package

**ARFF** Attribute-Relation File Format

**AUC** Area Under the Curve

**C&C** Command and Control

**CAR** Cloud App Repository

**CDS** Cloud Device Simulator

**CPU** Central Processing Unit

**CRAM** Anti-Malware Research Center of TG Soft

**dex** Dalvik EXecutable

**EoF** End of File

**ER** Error Rate

**FN** False Negatives

**FP** False Positives

**FPR** False Positives Rate

**GPS** Global Positioning System

**HTTP** Hypertext Transfer Protocol

**JIT** Just-In-Time

**UID** kernel User-ID

**MIME** Multipurpose Internet Mail Extensions

**OS** Operating System

**PC** Personal Computer

**PMDS** Permission-based Malware Detection System

**RAT** Remote Administration Tool

**RIPPER** Repeated Incremental Pruning to Produce Error Reduction

**ROC** Receiver Operating Characteristic

**SDK** Software Development Kit

**SMS** Short Message Service

**SMTP** Simple Mail Transfer Protocol

**TCP** Transmission Control Protocol

**TER** Threat Exposure Rate

**TN** True Negatives

**ToS** Term Of Service

**TP** True Positives

**TPR** True Positives Rate

**VM** Virtual Machine

**UDP** User Datagram Protocol

**URL** Uniform Resource Locator

xxx

**USB** Universal Serial Bus

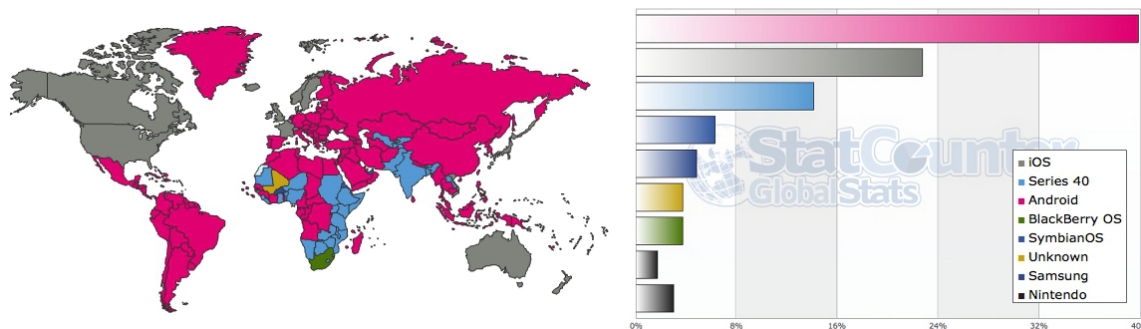
**XML** eXtensible Markup Language



# Chapter 1

## Introduction

In recent years, we have witnessed an exponential growth in mobile devices adoption. According to CNN (CNN, 2011), from 2008 to 2011 the number of smartphone shipments has tripled, and this number is still increasing. According to a report of Strategy Analytics (Strategy Analytics, 2012), in the third quarter of 2012, the number of smartphones in use worldwide have surpassed one billion-unit for the first time ever. Furthermore, according to a report of the International Data Corporation (IDC, 2013), in the first quarter of 2013 the total number of smartphones shipped has exceeded the one of feature phones.



**Figure 1.1:** Top mobile Operating Systems on September 2013 (StatCounter.com). As one can see, Android is the most widespread mobile platform in the world.

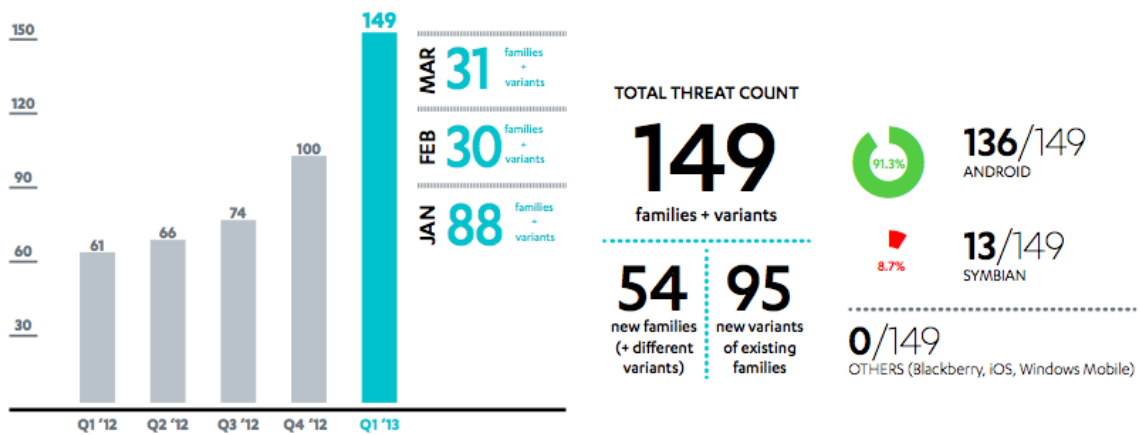
All these mobile devices, together with the wealth of sensitive information carried with them (e.g. personal and bank information, GPS location, SMS messages and emails), have inevitably driven to the evolution of malware targeting mobile platforms.

Since the very beginning of the mobile adoption, many papers have been carried out exploring possible threats to the mobile space (Leavitt, 2000; Foley & Dumigan, 2001; Dagon, Martin, & Starner, 2004; Hypponen, 2007; Lawton, 2008). Most of those papers agreed that the mobile space would have become a major security issue. Nevertheless,

we were not able to prevent mobile malware from their rising, rising which we are facing today.

Cyber-security firms have firstly addressed the mobile space as if it were an extension of the desktop space (and some of them are still persisting on doing so). However, there are key differences between the mobile and desktop spaces (Dagon et al., 2004; Lawton, 2008). Academic security researchers, on the other hand, have focused more on hypothetical, complex threats rather than on “in the wild”, often simpler ones (Spreitzenbarth & Freiling, 2012; Y. Zhou & Jiang, 2012). But attackers tends to choose the shortest path to exploitation, favouring over elegance and challenge.

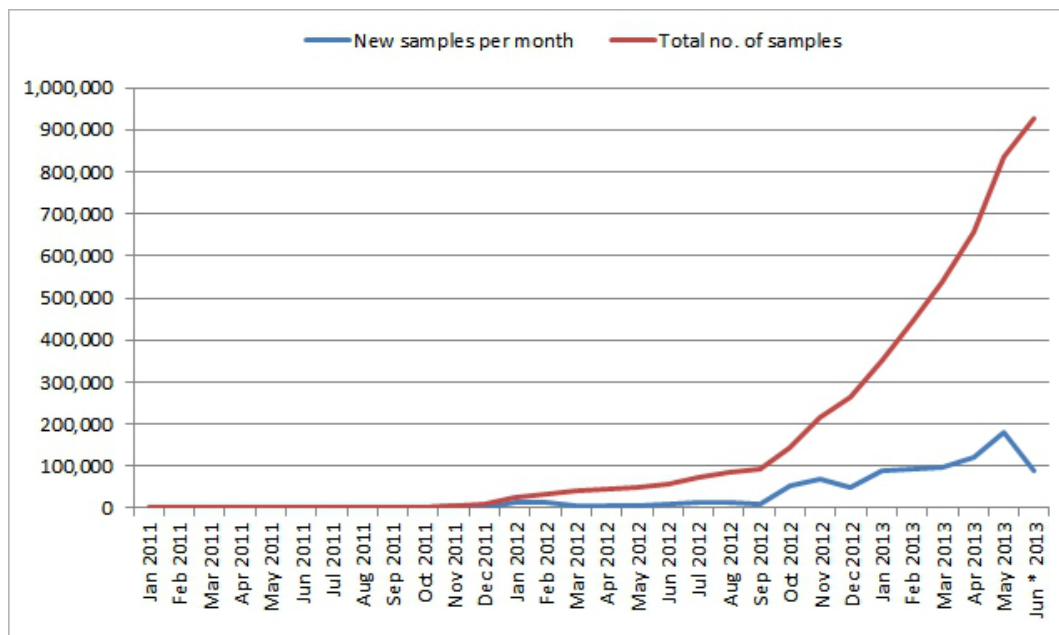
To all these factors, other and more influential ones have been added, contributing to make mobile malware one of the greatest computer threats in our time: for example, the usually slow patch cycle of the mobile platforms<sup>1</sup>, the fact that malware can be successful even without exploiting system vulnerabilities (especially Rogue-AVs and Trojans) and the drastic change of malware landscape from a for-fun activity to a profit-driven criminal business. Furthermore, many companies still do not provide protection for their employees’ mobile devices (Lawton, 2008).



**Figure 1.2:** New mobile threat families and variants from Q1 2012 to Q1 2013 (F-Secure) (F-Secure, 2013). The exponential growth in mobile devices adoption, together with the wealth of sensitive information carried with them (e.g. personal and bank information, GPS location, emails, etc...), have inevitably driven to the evolution of malware targeting mobile platforms. Many other factors have contributed to make mobile malware are one of the greatest computer threats in our time.

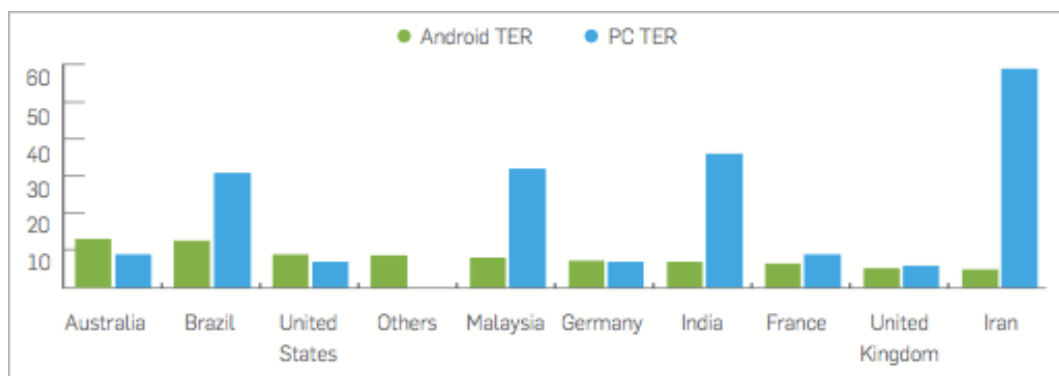
Among all the mobile platforms available, Android has become the more targeted one (see Figures 1.2 and 1.3). This is probably due to the fact that Android is the most widespread platform and to some technical particularities, such as the fact that Android applications are really easy to reverse engineer and to modify/repackage (see *Repackaging* in Section 2.2.3).

<sup>1</sup> According to Lookout, it took 42 weeks to the *Exploit://Android/Exploid* to reach its vulnerability half-life and even more to the *Exploit://Android/RageAgainstTheCage* (Wyatt, 2011).



**Figure 1.3:** Number of Android malware samples in the AV-TEST database between January 2011 and June 2013 (AV-TEST, 2013). The total number of Android malware samples was expected to exceed 1 million in the Summer of 2013.

Over the course of 2012, Lookout has estimated the likelihood of an Android user clicking on an unsafe link at 36% (Lookout, 2012). In 2013, Sophos has reported that, in some countries (i.e. Australia, Germany and the USA), the Android Threat Exposure Rate (TER) has exceeded those of PCs (Sophos, 2013b) (see Figure 1.4). In the same year, Panda Security has predicted that “Android will continue to be the number one mobile target for cyber-crooks, and the coming year will set a new record for the number of threats targeting this platform” (Panda Security, 2013a).



**Figure 1.4:** Android Threat Exposure Rate (TER) in 2013 (Sophos) (Sophos, 2013b). The TER is measured as the percentage of PCs and Android devices that experienced a malware attack, whether successful or failed. The experiment was run over a period of three months.

It seems current approaches are failing to consider key differences between mobile and desktop environments. As Zhou and Jiang point out in (Y. Zhou & Jiang, 2012), this

exponential growth of Android malware given, there is an actual call for the need of next-generation mobile security solutions.

We strongly believe that, before starting to address a problem, we firstly need to understand it deeply. Thus, in this work of thesis, we firstly investigate the current state-of-the-art of Android and Android malware. Afterwards, we introduce *VirIT Mobile Security*, a mobile security solution specifically designed and developed to counter Android malware. VirIT Mobile Security has been designed and developed in collaboration with and commercialised by TG Soft, an Italian antivirus firm.

The contributions presented in this work of thesis can be summarized as follows:

- We present a classification and characterisation of the current “in the wild” Android malware;
- we investigate possible detection models, either proactive or reactive, that can be applied to secure Android devices from the major classes of threats;
- we introduce the Android real-time malware detection problem and we discuss different possible solutions;
- we present a novel Android malware detection technique, called Permission-based Malware Detection System (PMDS), which is based on machine learning algorithms and, therefore, has the potential to detect previously unknown malware;
- we discuss the Android network protection problem.

The document is organized as follows.

In Chapter 2, we firstly introduce the Android Operating System and we present a classification and characterisation of the current “in the wild” Android malware.

In Chapter 3, we introduce the malware detection problem and we present a signature-based detection system specifically designed for Android.

In Chapter 4, we introduce the Android real-time malware detection problem and we discuss different possible solutions, with their benefits and drawbacks.

In Chapter 5, we propose a novel malware detection technique, called Permission-based Malware Detection System (PMDS), which is based on machine learning analysis of the Android permissions that an application requests.

In Chapter 6, we present the Android network protection problem and propose a Web Filter, which alerts the user about malicious and phishing websites, and a Network Monitor, which monitors the network usage of each application. Furthermore, we present a simple but effective behavioural detection system, known as Community Network, which collects and signals (potentially) dangerous applications to the Anti-Malware Research Center of TG Soft (CRAM) in order to spot zero-day or next-generation malware - i.e.

malicious applications which are not detected by the antivirus - as they emerge.

Finally, in Chapter 7, we draw the conclusions on our work of thesis and we discuss about the Android API, from a security-features prospective, and the future work.



# Chapter 2

## Android and Android malware

In this chapter we introduce the Android Operating System (see Section 2.1) and present a classification and characterisation of the current “in the wild” Android malware (see Section 2.2).

### 2.1 Android

In order to understand Android malware, and accordingly our design choices, it is firstly important to understand the Android architecture.

#### 2.1.1 The Android Operating System

Android is an Operating System (OS) primarily designed for touchscreen mobile devices (i.e. smartphones and tablets). It was initially designed and developed by the namesake *Android Inc.*, a startup bought by Google in 2005. Android was officially unveiled in 2007, and its source code was released by Google under the Apache License. (Google, 2013)

The Android OS is built on top of a modified version of the Linux kernel (see Figure 2.1). In particular, Android is a multi-user Linux system where each application is a different user and runs in its own Linux process. (The Android Open Source Project, 2013d)

On top of the Linux kernel there are the native libraries, such as *OpenGL* and *WebKit*, and the Dalvik Virtual Machine (VM), an open source Virtual Machine originally written



**Figure 2.1:** Android Operating System architecture. Android is a multi-user Linux system where each application is a different user and runs in its own Linux process, with its own instance of the Dalvik Virtual Machine. Therefore, an application's code runs in isolation from other applications. The system architecture is composed of five abstract layers: Linux kernel, native libraries, Android runtime, application framework and applications.

by Dan Bornstein (who named it after the Icelandic village of Dalvík) and optimized to run Java applications in mobile devices. The Dalvik VM uses a register-based architecture (with its own 16-bit instruction set) and Just-In-Time (JIT) compilation to run Dalvik Executable (dex) files. From the version 4.4, Android also supports ART Virtual Machine, a new experimental Virtual Machine that uses Ahead-Of-Time (AOT) compilation to run *oat* executable files. ([The Android Open Source Project, 2013m](#))

On top of the Android system architecture there are the applications (commonly referred as *apps*), which run on an application framework made of Java-compatible libraries based on Apache Harmony.

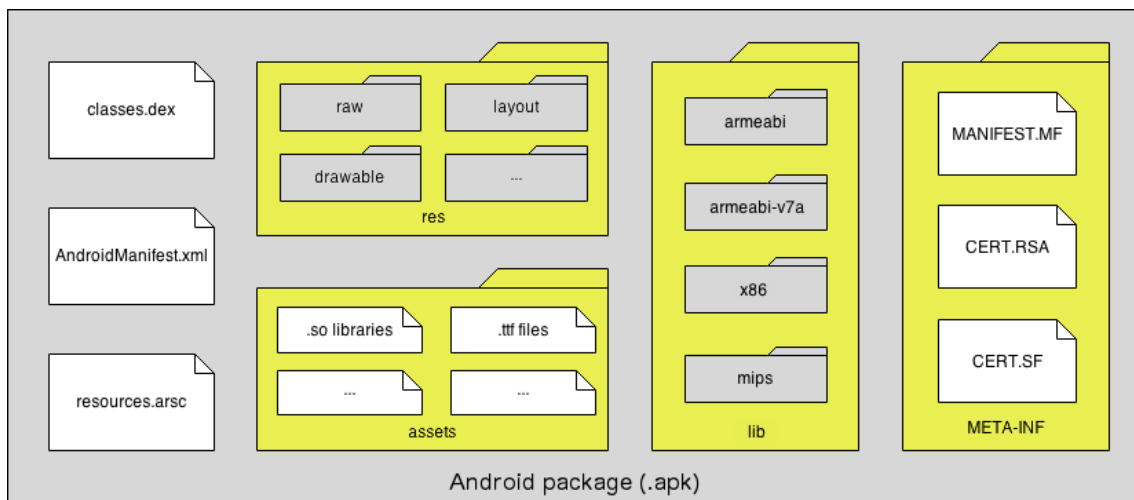
User applications are installed in the `/data/app` directory and their data are saved into the `/data/data` directory, while the system applications are installed in the `/system/app` and, starting from Android 4.4 KitKat, in the `/system/priv-app` ones. Without the root privileges, a third-party application cannot access to the other applications' `/data/data` directory. Furthermore, each application in Android runs in its own process, with its own instance of the Dalvik VM. Therefore, an application's code runs in isolation from other applications.



## 2.1.2 The APK File Format

Android applications are distributed and installed using the Android package (APK) file format. This is an archive file, built on the ZIP file format, and have the *.apk* file extension. Since it is basically a ZIP file, also the header signature of an APK package will be *0x504B0304* (big-endian). The MIME type associated with APK packages is: *application/vnd.android.package-archive*.

As shown in Figure 2.2, an APK package contains all the application's code, its resources

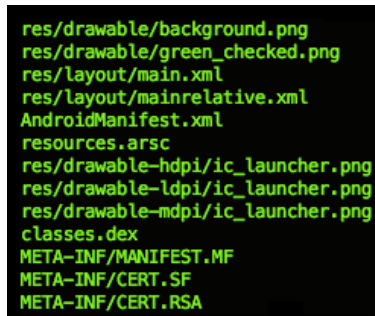


**Figure 2.2:** The APK package structure. Android applications use the Android package (APK) file format, which is an archive file built on the ZIP file format and contains: the *classes.dex* executable file, which represents the set of all the Java classes compiled in the Dalvik EXecutable (dex) file format understandable by the Dalvik Virtual Machine, the *AndroidManifest.xml* file, which provides semantic-rich information about the application itself and its components, the *resources.arsc* file, which describes the pre-compiled resources, the *lib* directory, which contains the compiled code that is specific to a software layer of a processor, the *assets* and *res* directories, which contains the application assets and resources respectively, and the *META-INF* directory, which contains the digital certificate with which the application was signed and the signatures of all the files within the APK package. ([The Android Open Source Project](#), n.d.)

and assets, the digital certificate and a manifest file. In particular it contains:

- *classes.dex*: it is the Dalvik Virtual Machine executable file. It contains the set of all the Java classes compiled in the Dalvik EXecutable (dex) file format ([The Android Open Source Project](#), 2013h);
- *AndroidManifest.xml*: it is a *binary XML* file, a compact representation of XML, that provides semantic-rich information about the application itself (e.g. name, version and permissions required) and its components (e.g. Activities, Services and BroadcastReceivers) ([The Android Open Source Project](#), 2013f);
- *resources.arsc*: it describes the pre-compiled resources (e.g. *binary XML*);
- *assets*: it is a directory which contains the application assets;

- *res*: it is a directory which contains all the resources (e.g. layouts, menu, images, strings, ...) not compiled into the *resources.arsc* file;
- *lib*: it is a directory which contains the compiled code that is specific to a software layer of a processor;
- *META-INF*: it is a directory which contains the digital certificate with which the application was signed (the *CERT.RSA*<sup>1</sup> file), and the signatures of all the files within the APK package (the *MANIFEST.MF* and the *CERT.SF* files contain the list of files and the SHA-1 hashes, of the files and of their declarations in the *Android-Manifest.xml* file respectively).



```
res/drawable/background.png
res/drawable/green_checked.png
res/layout/main.xml
res/layout/mainrelative.xml
AndroidManifest.xml
resources.arsc
res/drawable-hdpi/ic_launcher.png
res/drawable-ldpi/ic_launcher.png
res/drawable-mdpi/ic_launcher.png
classes.dex
META-INF/MANIFEST.MF
META-INF/CERT.SF
META-INF/CERT.RSA
```

Figure 2.3: The list of files within the APK package of a malicious application (*Trojan-Banker://Android/ZitMo.B*).

### 2.1.3 Google Play Store and Third-Party Markets

Google Play Store, formerly Android Market, is the main digital distribution platform for Android applications and media contents. Applications can be downloaded directly to an Android device through the Play Store application or can be deployed to an Android device from the Google Play website.

In addition to Google Play Store, tens of alternative, third-party Android markets are born in the last years (e.g. Amazon Appstore, AndroLibs, AppBrain, Aptoide, F-Droid, LG World, Opera Mobile Store, Samsung Apps and many more). These third-party Android markets offer either exclusive apps and apps that were banned from Google Play Store.

<sup>1</sup> In the first versions of Android the name of the digital certificate might be different than *CERT.RSA*, such as: *PACKAGENAME.DSA/RSA*.

## 2.2 Android Malware Taxonomy

The importance of a malware taxonomy is twofold. On one hand, it characterises the current known threats, helping us to better understand their behaviour and to generate effective detection models. On the other hand, it helps us to anticipate what kinds of threats will come and, consequently, to response faster to new threats.

Thus, in order to generate effective Android malware detection models, our first step is to identify and classify the different kinds of “in the wild” Android malware.

### 2.2.1 Malware Naming Scheme

Malware is an umbrella-term for various types of unwanted piece of software and/or executable codes that are used to perform unauthorized, often harmful, actions on computing devices.

A malware naming scheme has been proposed by Skúlason and Bontchev back in 1991 (Skulason & Bontchev, 1991). Even if this naming scheme is nowadays slightly outdated, it still remains the only existing standard that most computer security companies and researchers ever attempted to adopt. (Szor, 2005)

Nowadays, the malware naming is often abbreviated in its minimum form:

`<malware_type>://<platform>/<family_name>.<variant>`

e.g. *Trojan://Android/DroidKungFu.A*

In practice, many slightly different variants of this naming scheme has been adopted by the various computer security companies and researchers, such as:

*Trojan:Android/DroidKungFu.A*

*Android/Trojan.DroidKungFu.A*

*Android.Trojan.DroidKungFu.A*

*Android.DroidKungFu.A [Trojan]*

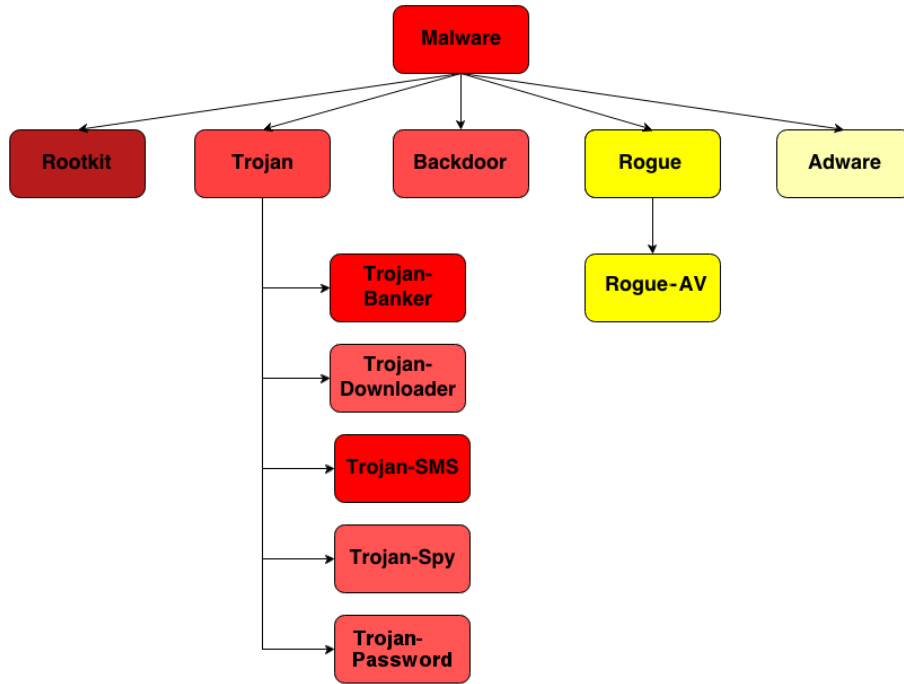
In this work of thesis, we decided to use the standard naming scheme proposed by Skúlason and Bontchev.

## 2.2.2 Malware Types

Malware can be firstly classified according to their behaviour, i.e. their malicious activities.

Note that due to architecture design/limits (see Section 2.1.1), under Android there are no computer viruses in the strict sense of the term (i.e. “a code that recursively replicates a possibly evolved copy of itself” (Szor, 2005)).

In Android most of the existing malware types are directly inherited from the desktop

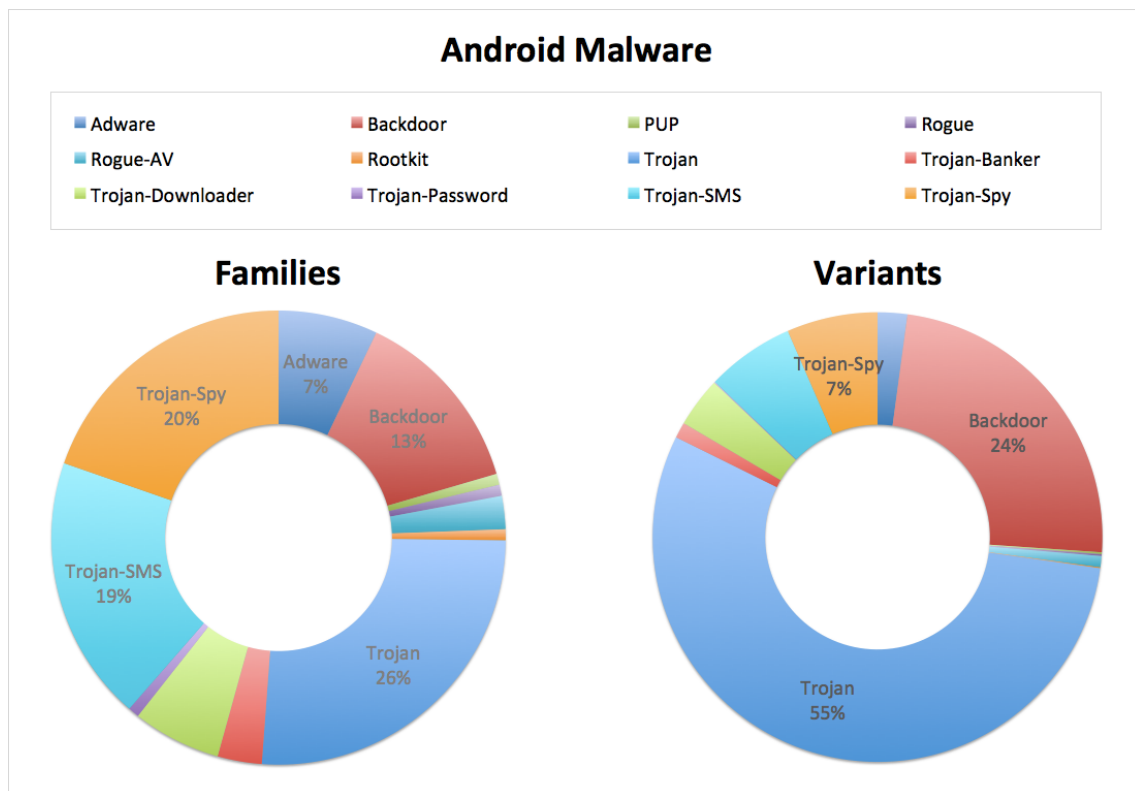


**Figure 2.4:** The Android malware types. Most of the existing Android malware types are directly inherited from the desktop space (e.g. Adware or Rogue-AV), even if some of them have additional capabilities due to the mobile space. However, there are also others Android malware types that are unique to mobile space (e.g. Trojan-SMS).

space (e.g. Adware or Rogue-AV), even if some of them have additional capabilities due to the mobile space (e.g. all the contacts information stored in one place, the ability to send SMS messages and to perform phone calls). However, there are also others malware types that are unique to mobile space (e.g. Trojan-SMS).

Most of the Android malware are actually “Trojanised” versions of legitimate applications. In particular, we collected a dataset of 1500 samples, taken from the Android Malware Genome Project (Y. Zhou & Jiang, 2012), Contagio Mobile (Mila, 2013) and other on-line sources. Subdividing the collected samples by type, we got a number of 826 samples of Trojan (divided into 33 families), 359 samples of Backdoor (divided into 17 families), 98 samples of Trojan-Spy (divided into 25 families) and 95 samples of Trojan-SMS (divided into 24 families) among the others (see Figure 2.5).

In particular, the most common types are:



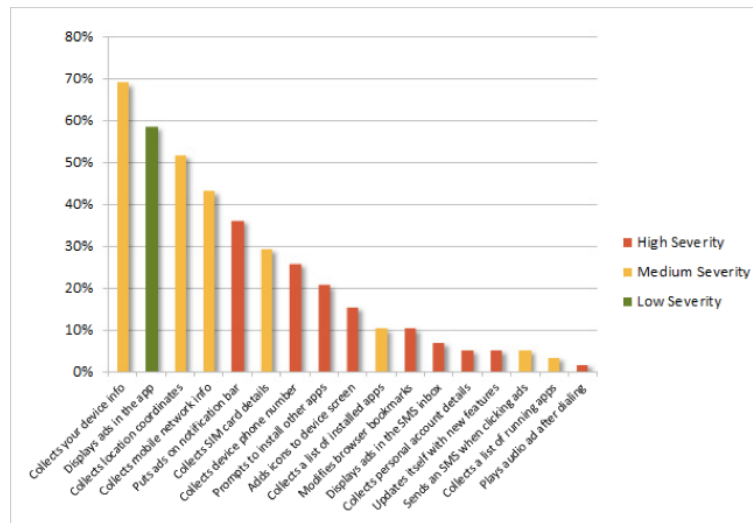
**Figure 2.5:** The Android malware types in percentage, for both the number of families (on the left) and variants (on the right). The data refer to a dataset of 1500 samples, taken from the Android Malware Genome Project (Y. Zhou & Jiang, 2012), Contagio Mobile (Mila, 2013) and other on-line sources. The collected samples were later subdivided by type, obtaining a number of 826 samples of Trojan (divided into 33 families), 359 samples of Backdoor (divided into 17 families), 98 samples of Trojan-Spy (divided into 25 families) and 95 samples of Trojan-SMS (divided into 24 families) among the others.

- **Adware:** it is a particular application that automatically displays annoying, often misleading, advertisements in order to generate revenue for its author.

In Android, many applications are distributed or have a free version which includes in-app advertisements. Some ad libraries (defined as “aggressive”), however, are able to leak personal and sensitive information or exhibit annoying behaviours, such as: displaying ads in the notification bar or full screen, creating ad icons or changing web browser bookmarks. According to Symantec, in 2013, there were 65 known ad libraries of which over 50% of them were classified as aggressive. (Uscilowski, 2013)

Furthermore, the very mild policies (or non-policies) of Google in the field of in-app advertising are contributing to the significant rising of the number of Adware hosted in Google Play store. *Trojan://Android/FakeMarket.A*

- **Backdoor:** it is a particular application - also known as Remote Administration Tool (RAT) - that allows an attacker to take control of the device (without the user consent or knowledge) and perform various malicious activities from a remote lo-



**Figure 2.6:** Ad libraries' behaviours and their levels of severity in 2013 (Symantec). In Android, many applications are distributed or have a free version which includes in-app advertisements. Some ad libraries (defined as "aggressive"), however, are able to leak personal and sensitive information or exhibit annoying behaviours. In this graph, the ad libraries' behaviours are divided accordingly to their aggressiveness. In *low severity* (green) are grouped all those ad libraries which display ads inside the applications and do not leak any data. In *medium severity* (yellow) are grouped all those ad libraries which leaks non-harmful data (e.g. location or mobile network). In *high severity* (red) are grouped all those ad libraries which leak private data (e.g. phone number or user account) or annoy/lure the user (e.g. show ads in the notification bar or play a voice ad when making a phone call). According to Symantec, in 2013, there were 65 known ad libraries of which over 50% of them were classified as aggressive. (Uscilowski, 2013)

cation, such as: intercept phone calls and SMS messages, make phone calls, send SMS messages, open websites, download and install other applications, delete files, collect and send information back to the attacker.

The victim devices (called *bots*) receive commands from the attacker (called *bot-master*) through a Command and Control (C&C) server and perform corresponding actions.

An example of a Backdoor is the *Backdoor://Android/OBad.A*, which is considered one of the most complicated existing Android malware. This malware is capable of downloading and installing other malware, re-sending the downloaded malware via Bluetooth, sending SMS messages to premium-rate numbers, connecting to specific web pages, opening a remote shell, collecting and forwarding to the C&C server various private information of its victims. These private information includes: Bluetooth MAC address, mobile operator, telephone number, IMEI and user account balance. (Unuchek, 2013)

- **Rogue** (also known as FraudTool): it is a deceptive application that pretends to be a well-known or trusted software in order to steal money and/or confidential data. An example of a Rogue is the *Rogue://Android/FakeFlash.A*, which tries to lure its victims to pay 5 Euro to download the Adobe Flash Player app. Adobe does not offer the "standalone" app on Google Play Store anymore. However, the Flash

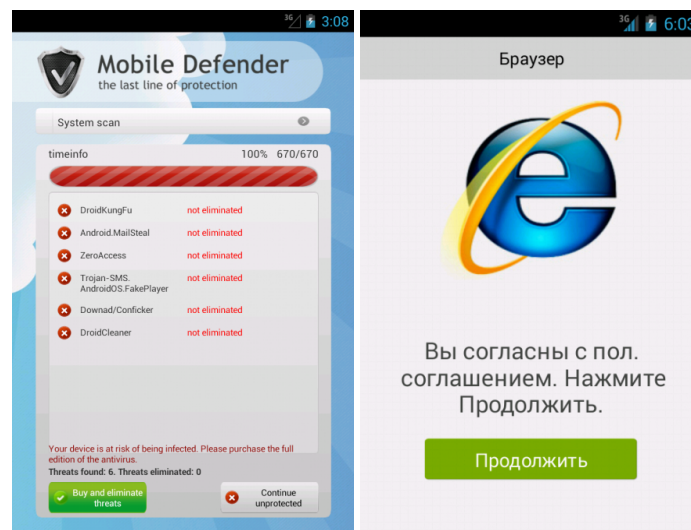
Player functionality has been integrated into the Adobe AIR app, which can be downloaded for free on Google Play Store. (SB, 2014)

- **Rogue-AV** (also known as FakeAV): it is the main and mostly unique sub-type of Rogue. This kind of Rogue pretends to be an antivirus solution and claims to detect some threats on the victims’ devices in order to lure the victims to pay to remove the non-existent threats. (Corrons & Correll, 2010; Stone-Gross et al., 2013)

An example of such a malware is the *Rogue-AV://Android/AndroidDefeder.A* (see Figure 2.7). (Rovelli, 2013c; Ducklin, 2013)

- **Rootkit**: it is a particular malware that operates at the kernel level of the Android Operating System and, thus, it is difficult to detect and require special operations to be removed.

Albeit various researches have been carried out on the topic (Oi, 2011; Jiang, 2012a; Brodbeck, 2012; Ayer, n.d.), we have found only one family of “in the wild” Rootkits so far, known as *Rootkit://Android/Oldboot*. (Xiao, Dong, Zhang, & Jiang, 2014)



**Figure 2.7:** On the left a screenshot of *Rogue-AV://Android/AndroidDefeder.A*, which claims to have found 6 malware on the device and asks to purchase a licence in order to remove them, while on the right a screenshot of *Trojan-SMS://Android/Agent.B*, which pretends to be a version of Microsoft Internet Explorer for Android but subscribes the user to paid services automatically answering to specific SMS messages.

- **Trojan** (or Trojan Horse): it is a deceptive application that intentionally conceals its malicious actions, pretending to perform other functionalities. In order to persuade the user to install it, a Trojan generally disguises itself as an attractive application, such as videogame or an application to improve the performance or the safety of the device. Thanks to repackaging (see Section 2.2.3), in Android, it is really common to see “Trojanised” versions of legitimate applications.

Examples of “in the wild” Trojans are the *Trojan://Android/FakeMarket.A* and the



*Trojan://Android/SMSAgent.A*. The first one was distributed on Google Play store as a videogame application. Once installed, the malware pretends to be the Google Play store application, but it runs silently in the background to perform click fraud (Rovelli, 2014; AndroTotal, 2014). The latter, instead, tries to hidden itself from the user in order to steal its personal information (such as the phone number and the IMEI) and to take control of SMS (Rovelli & Tonello, 2013).

Some Trojans are further subdivided accordingly to their malicious actions. For example, if a Trojan's financial revenues and malicious actions are only related to send SMS messages to premium-rate numbers, then this Trojan is classified as a Trojan-SMS. Similarly, a Trojan which aims only to spy its victims and to steal their personal and sensitive information is classified as a Trojan-Spy. However, not all the Trojans can be sub-classified. Indeed, most of the Trojans use perform multiple malicious actions together and, thus, cannot be specifically labelled. Furthermore, in Android, most of the Trojans have botnet capabilities - i.e. they are able to open a backdoor from which send and receive commands.

- **Trojan-Banker:** it is a particular sub-type of Trojan that aims to steal its victims' online banking account credentials (i.e. username and password) and/or other data related to online payments (e.g. credit card details or SMS one-time passwords). The collected data are then forwarded to remote Command and Control (C&C) server via a communication channel, such as: HTTP, SMTP or SMS.

Examples of such a malware are the various *Trojan-Banker://Android/ZitMo* variants, which intercept all incoming SMS messages and forward them to a C&C server, via HTTP POST requests or SMS messages depending on the variant. The *Trojan-Banker://Android/ZitMo* represents a classic example of *Man-in-the-Mobile* attack which is able to defeat the SMS-based two-factor authentication. (Rovelli, 2013e; Castillo, 2011; Maslennikov, 2011; Barroso, 2010)

- **Trojan-Downloader:** it is a particular sub-type of Trojan that, unbeknownst to its victims, downloads and installs other malicious applications on the device. The information about which applications have to be downloaded and where to find them can be hard-coded in the Trojan-Downloader itself or retrieved from the web.

Examples are the *Trojan-Downloader://Android/DroidDream.A* and the *Trojan-Downloader://Android/RootSmart.A*. In order to gain the root privileges, the first one makes use of the *Exploit://Android/Exploid* and, in case of failure, of the *Exploit://Android/RageAgainstTheCage*. In case of success, the malware



installs (in the system folder) a second application that will start and manage the downloads of other applications (Lookout, 2011; Svajcer, 2011). The latter one, instead, checks if the device is exploitable (i.e. Android version less or equal to 2.3.4) and if it has not been exploited before. If that is the case, the malware downloads and executes the *Exploit://Android/GingerBreak* in order to gain the root privileges. Then, it downloads a Remote Administration Tool (RAT) and starts installing other malicious applications. (Mullaney, 2012; Jiang, 2012b; Spreitzenbarth, 2012)

- **Trojan-Password:** it is a particular sub-type of Trojan that aims to steal its victims' account credentials (i.e. username and password).  
An example of such a malware is the *Trojan-Password://Android/FakeNetflix.A*, which tries to steal the Netflix account credentials and to send them via HTTP. (Asrar & Imano, 2011)
- **Trojan-SMS:** it is a particular sub-type of Trojan in which the malicious actions and the financial revenues are strictly related to SMS messages (e.g. send SMS messages to premium-rate numbers). An example of an such a malware is the *Trojan-SMS://Android/Agent.B* (see Figure 2.7), which tries to subscribe the user (without his/her knowledge) to paid services automatically answering to specific SMS messages. (Rovelli, 2013a)
- **Trojan-Spy** (also known as Spyware): it is a particular sub-type of Trojan which strictly aims to spy its victims' actions (e.g. sent/received SMS messages or performed/received phone calls) and to collect their personal and sensitive information (e.g. mobile operator, telephone number, IMEI, user account balance, contacts and current location). The harvested data are then forwarded to remote Command and Control (C&C) server via a communication channel, such as: HTTP, SMTP or SMS.  
An example of such a malware is the *Trojan-Spy://Android/SMSAgent.B*, which collects all the SMS received by the user and forwards them via email to a specific address. (Rovelli, 2013b)

### 2.2.3 Malware Propagation Methodologies

Malware can also be classified according to the way in which they spread.

Also in this kind of classification, in Android, there are either propagation methods that are common also in the desktop space (e.g. drive-by download) and others that are mostly unique to Android space (e.g. repackaging).

Type	Family	Propagation	Privilege Escalation	Remote C&C	Direct Financial Charges	Privacy Leak
Backdoor	AnserverBot	Update	No	HTTP	SMS	No
	Obad	Standalone	Yes	HTTP and SMS	SMS	Yes
	Geinimi	Repackaging	No	HTTP	Calls and SMS	Yes
	SpamSoldier	Repackaging	No	HTTP	SMS	No
Rogue	FakeFlash	Standalone	No	-	-	No
	JobOffer	Standalone	No	-	-	No
Rogue-AV	AndroidDefeder	Standalone	No	-	-	No
	FakeDefeder	Standalone	No	HTTP	-	No
Rootkit	Oldboot	Pre-Installed	-	HTTP	-	Yes
Trojan	BaseBridge	Repackaging or Update	Yes	HTTP	Calls and SMS	No
	DroidKungFu	Repackaging	Yes	HTTP	-	Yes
	FakeMarket	Standalone	No	-	-	No
	GoldDream	Repackaging	No	HTTP	Calls and SMS	Yes
	Plankton	Update	No	HTTP	-	No
	Skullkey	“Master Key”	Yes	HTTP	SMS	Yes
Trojan-Banker	ZitMo	Drive-by Download	No	HTTP or SMS	SMS	Yes
	SpitMo	Drive-by Download	No	HTTP	SMS	Yes
Trojan-Downloader	DroidDream	Repackaging	Yes	HTTP	-	No
	RootSmart	Repackaging	Yes	HTTP	-	Yes
Trojan-Password	FakeNetflix	Standalone	No	HTTP	-	Yes
Trojan-Spy	ADRD	Repackaging	No	HTTP	-	Yes
	Loozfon	Standalone	No	HTTP	-	Yes
	Nickyspy	Standalone	No	HTTP	SMS	Yes
	SMSZombie	Repackaging	No	SMS	SMS	Yes
Trojan-SMS	Hippo	Repackaging	No	-	SMS	Yes
	Lemon	Standalone	No	HTTP	SMS	Yes
	OpFake	Repackaging	No	-	SMS	No
	Raden	Repackaging	No	-	SMS	No

**Table 2.1:** An overview of some of the most known Android malware. For each malware, it is shown: its type, its family, the propagation method, if it tries a privilege escalation (root the device), if it connects to a Command and Control (C&C) server and in that case with which channel (e.g. SMS, HTTP or SMTP), if it leads to direct financial costs (e.g. by sending SMS messages or performing phone calls) and if it steal personal information (e.g. mobile operator, telephone number, IMEI, contacts, web history, geographical location, SMS messages and phone calls logs).

In particular, the most common types are:

- **Drive-by Download:** it is a Social Engineering attack in which an application is downloaded and installed on the device without the user knowledge or understanding. Drive-by download usually happens due to the user clicking on a deceptive pop-up (e.g. proposing a killer-feature application) or due to a malware already installed on the device.

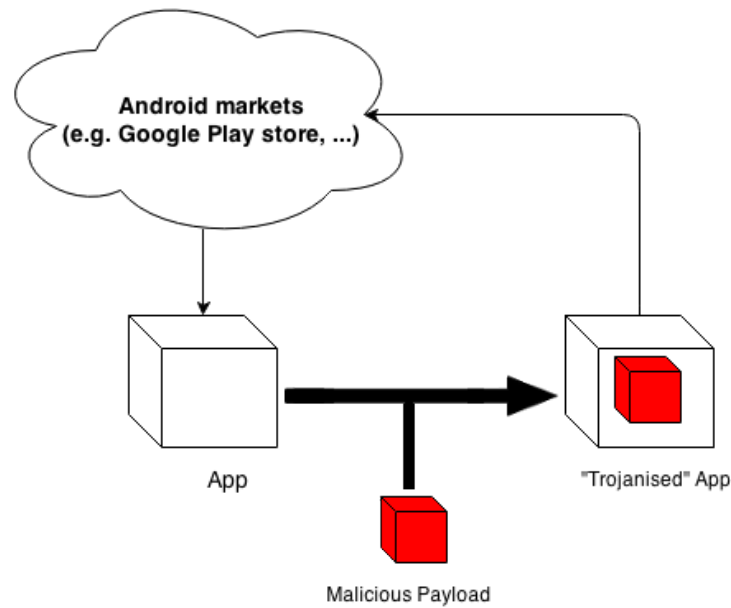
A classic example of a drive-by download attack is the one performed by *Trojan-Banker://Win32/Zeus*. Indeed, when one is doing online banking on a PC infected with the *Trojan-Banker://Win32/Zeus*, he/she will be redirected to download a particular mobile application that is claimed to enhance the safety of online banking. However, the downloaded application is the *Trojan-Banker://Android/ZitMo*, which intercepts all the incoming SMS and forward them to a C&C server in order to defeat the SMS-based two-factor authentication (see Trojan-Banker in Section 2.2.2). (Rovelli, 2013e; Maslennikov, 2011)

- **Malicious Ad Libraries:** it is one of the most recent attack discovered in the Android space. In this attack, malware are spreading through the ad libraries used by benign (non-malicious) applications. Indeed, in Android, many applications are distributed or have a free version which includes in-app advertisements. These ads displayed, however, are not controlled by the application developers, but by the chose ad libraries. Malicious ad libraries display normal ads until a big enough number of users is reached, then they start displaying misleading ads that claim there are some updates but instead install malware on the device.

In April, Panda Security reported that they have found 32 applications available in Google Play that were using malicious ad libraries. The total number of downloads of these applications through Google Play reached 9 million. (Panda Security, 2013b)

- **Repackaging:** it is the most common attack used in the Android space. Thanks to Repackaging, malicious payloads can be injected into popular, benign applications (see Figure 2.8).

Basically, the APK package of a benign application is retrieved from an Android market (usually Google Play Store) and disassembled (usually the *Dalvik* bytecode is converted into *JAR* or *smali* code). Then, the malicious payload is injected and the APK package is re-assembled. Finally, the new “Trojanised” version of the application is submitted to one or more alternative, third-party Android markets (see



**Figure 2.8:** An example of repackaging. A benign application is downloaded from an Android market (usually Google Play Store) and disassembled. A malicious payload is then injected into the application, the package is re-assembled and, finally, the new “Trojanised” version of the application is uploaded back into an alternative, third-party Android market.

### Seciton 2.1.3).

Since it is strictly based on the APK package structure (see Seciton 2.1.2) and on the possibility of re-publishing an application in third-party Android markets, this kind of attacks is unique to the Android environment.

A classic example of repackaging is given by *Trojan-Downloader://Android/RootSmart.A*, which is a repackaged version of the legitimate application named *QuickSettings* (see Trojan-Downloader in Section 2.2.2). (Mullaney, 2012; Jiang, 2012b)

A total of 1083 out of 1260 (the 86%) of the samples collected by Zhou et al. in their *Android Malware Genome Project* are repackaged (Y. Zhou & Jiang, 2012). In the literature, many frameworks that are designed specifically to detect repackaged applications have been proposed, such as DroidMOSS (W. Zhou, Zhou, Jiang, & Ning, 2012) and DNADroid (Crussell, Gibler, & Chen, 2012)

It is important to note that, in classic Repackaging, due to the Android platform design, the “Trojanised” versions of the original applications have to be digitally re-signed with their own certificate. Indeed, in Android all installed applications have to be digitally signed with a certificate whose private key is held by the application’s developer (The Android Open Source Project, 2013r). If there’s a mismatch between the certificate and the APK package, the application is rejected. This is clearly a main limitation of the classic repackaging attack, since an examination of the application’s certificate file (i.e. *CERT.RSA*) can instantly show whether it was created by the legitimate publisher or not.

- **Android “Master Key” vulnerability:** it is one of the most recent repackaging attack discovered in the Android space that allows attackers to inject malicious code into an APK package without having to digitally re-sign the application. Indeed, if two files with the same name would be putted in an APK package (see Figure 2.9), the Android system verifies the digital signature of the first but installs the latter. Thus, it can be the case in which the system verifies the original, legitimate application but installs its “Trojanised” versions. (Forristal, 2013; Sophos, 2013a)

The first discovered “in the wild” malware that used this technique was the *Trojan://Android/Skullkey.A*, which allows to remotely control the infected devices, steals sensitive data and sends SMS messages to premium-rate numbers. (Symantec Security Response, 2013)

assets	File folder	
lib	File folder	
META-INF	File folder	
res	File folder	
AndroidManifest.xml	XML Docum...	22 KB
AndroidManifest.xml	XML Docum...	27 KB
classes.dex	DEX File	1,191 KB
classes.dex	DEX File	1,260 KB
resources.arsc	ARSC File	70 KB

**Figure 2.9:** The Android “Master Key” vulnerability allows attackers to inject malicious code into an APK package without having to digitally re-sign the application. Here, two *classes.dex* and two *AndroidManifest.xml* files are putted in the same APK package. The Android system verifies the digital signature of the first (the legitimate one) but installs the latter (the malicious one).

- **Update:** it is an attack in which, instead of injecting the entire malicious payloads into an APK package, only an update component is injected. This update component will be responsible to retrieve the malicious payload at run-time. A classic example of an update attack is given by some variants of the *Trojan://Android.BaseBridge*, where at the first run a dialogue that claims that a new version of the application is available is displayed. The new version is actually stored inside the application’s APK package as an resource file. If the user accepts to install the “update”, the application containing the malicious payload will be installed. (Y. Zhou & Jiang, 2012; NQ Mobile, 2012)
- **Usurping ads:** it is a specific repackaging attack in which the only changes regard the publisher IDs in the *AndroidManifest.xml* file.  
As already said (see *Malicious ad libraries* in Section 2.2.3), in Android, many applications take advantage of in-app advertisements. In order to do that, the developers have to register to one or more ad networks, which in turn assign one or more publisher IDs to each developer. These publisher IDs are used

to properly identify and revenue developers for the user clicks and ad traffics. Ad libraries typically require that the developer includes its publisher IDs into *meta-tags* in the *AndroidManifest.xml* file (see Figure 2.10). Thus, by replacing the original publisher IDs with the cyber-criminal ones, the repackaged applications will behave exactly as the original one but the ad revenues will be collected by the cyber-criminals. (W. Zhou et al., 2012)

```
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="2" android:versionName="1.1" android:installLocation="auto"
package="com.brnk.apps.wifihacker" xmlns:android="http://schemas.android.com/apk/res/android">
.....
  <application android:theme="@*android:style/Theme.NoTitleBar" android:label="@string/app_name" android:icon="@drawable/
ic_launcher" android:allowBackup="true">
.....
    <meta-data android:name="MINIMOB_APPID" android:value="XXXXXXXXXX" />
    <meta-data android:name="MINIMOB_APPKEY" android:value="XXXXXXXXXX" />
    <meta-data android:name="MINIMOB_APPHOST" android:value="http://mpm.minimob.com/mobile/serve.asp" />
    <meta-data android:name="MINIMOB_OUTOFAPP" android:value="true" />
.....
  </application>
</manifest>
```

Figure 2.10: An example of *AndroidManifest.xml* file in which it is included the developer's publisher IDs given by an ad network (*MiniMob* in this example). Cyber-criminals can replace the original publisher IDs with their own in order to steal the ad revenues.

- **Standalone:** it is a Social Engineering attack in which an application misleads users for downloads. It is often the case of malware that masquerade as legitimate applications (e.g. Rogue or Trojan) or that actually provide the claimed functionalities but, unbeknownst to the user, they also perform other malicious actions.
  - **Malicious ToS:** it is the most recent attack discovered in the Android space. In this attack, malicious applications lure the user into accepting misleading Term Of Service (ToS) - usually too small to be read - through which, for example, the user is unwittingly accepting to be subscribed to premium-rate SMS service. (Corrons, 2013b)

## 2.3 Related Work

A complete, standard (desktop) classification of malware according to their propagation methods and goals can be found in (Bontchev, 1998).

In (Felt, Finifter, Chin, Hanna, & Wagner, 2011), Felt et al. survey the state-of-the-art of mobile malware “in the wild” and discusses possible defences. The authors study malware samples on three different mobile platforms: Android (16 samples), iOS (4 samples) and Symbian (24 samples). The main difference with our work is that, having a broader dataset, we focus only on Android malware.

In (Y. Zhou & Jiang, 2012), Zhou et al. systematically characterise existing Android malware from various aspects, including their installation methods and activation mechanisms. Zhou et al. work is more detailed than ours. We have preferred just to give an overview of the topic in order to focus more on Android malware detection (see Chapters 3, 5 and 6). However, since our work is more recent and malware are continually evolving, we discuss about malware families and propagation methodologies that did not exist, or had not yet been discovered, at the time (see for example *Malicious ad libraries*, *Android “Master Key” vulnerability* and *Usurping ads* in Section 2.2.3).

In (Spreitzenbarth & Freiling, 2012), Spreitzenbarth et al. give an overview of the existing Android malware families and their main functionality.

In (Suarez-Tangil, Tapiador, Peris-Lopez, & Blasco, 2014), Suarez et al. propose Dendroid, a text mining approach to analyzing and classifying code structures in Android malware families. The authors use these code structures in order to investigate hierarchical relationships (phylogenetic tree) among malware families.

## 2.4 Conclusion

In this chapter we have introduced the Android Operating System and presented a classification and characterisation of the current “in the wild” Android malware.

In our opinion, the exponential rise of Android malware can be attribute to four main factors:

- The continuous growth in Android adoption, by both device manufacturers and end users;
- Android malware can be successful even without exploiting system vulnerabilities (especially Rogue-AV, Trojan-Banker and Trojan-SMS);
- The trust of Android users in Google Play Store and in third-party Android market as well (which is in turn leading to an increasing number of services for buying downloads and ratings in Google Play, in order to lure the user to download an application);
- The policies (or non-policies) of Google in the field of in-app advertising.

New propagation methodologies will surely appear in the coming years. In particular, we will probably face an increasing number of attempts to infect mobile devices directly from PCs and Social Networks. (Corrons, 2013a; Liu, 2014)

Furthermore, we are already seeing an increasing number of digital currency-mining Trojan (JS & AB, 2014; Rogers, 2014) and of Ransomware (Zorabedian, 2014). However, given their success, we suppose Trojan-SMS and premium SMS frauds will still remain the biggest threat in Android during the next year. However, starting from Android 4.4 (KitKat), Google has introduced some changes to the API which should significantly slow down the rise of such threat. (Main, Scott and Braun, David, 2013)

We would like to share a particular, final, though on the Google policies in the field of in-app advertising and on the recent rise of Android Adware (Uscilowski, 2013). Indeed, in Android, on the very opposite of other mobile platforms, in-app advertising has become really popular. Many Android developers use ad libraries to legally monetize their applications by displaying advertisements on them. In the beginning of 2013, in order to make in-app advertising more effective, Google has removed all the ad blocking applications from Google Play Store. However, with its extensive policies, Google had let the spreading of many applications which use “aggressive” ad libraries, which are able to leak sensitive information or mislead the user to download other malware. In September 2013, Google have finally updated the Term Of Service (ToS) and consequentially removed around 36.000 Adware from Google Play Store (Gamble, 2012). However, there is a lot still to do in order to reduce the number of “in the wild” Adware drastically.



## Chapter 3

# Implementing a Signature-based Malware Detection System in Android

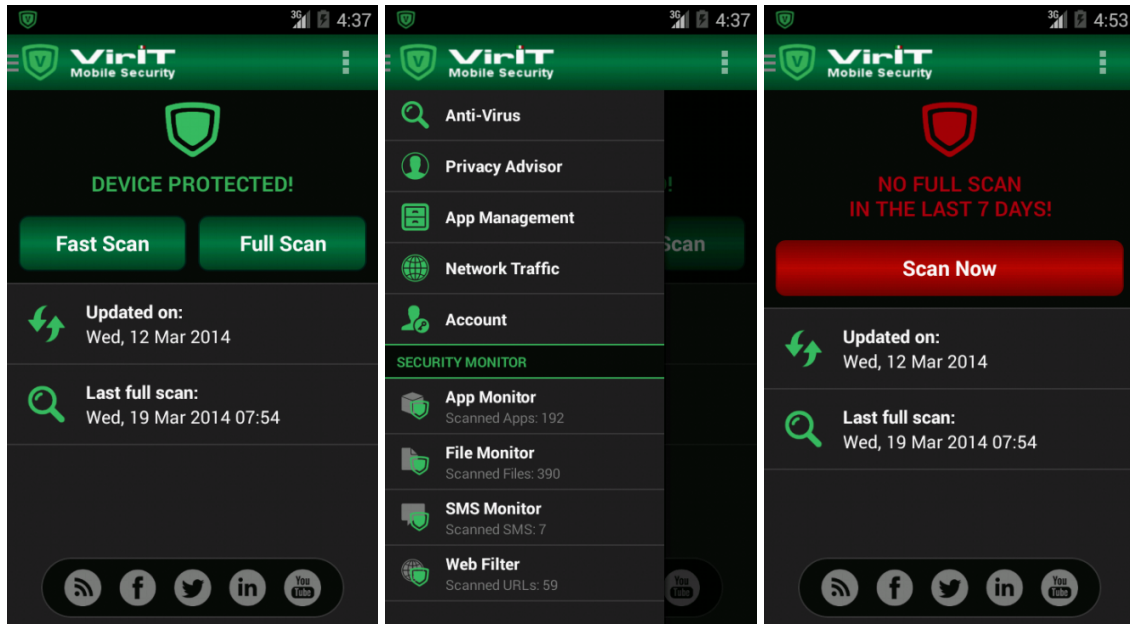
In this chapter we introduce the malware detection problem (see Section 3.1) and present a signature-based detection solution specifically designed for Android (see Sections 3.3 and 3.4).

Experimental results show that our signature-based detection system was able to properly detect and remove 95.95% of the malware in our test.

### 3.1 Introduction

Modern antivirus software typically rely on a variety of methods to detect and identify malware, such as: signature-based (Ask, 2006; ClamAV, 2007; Griffin, Schneider, Hu, & Chiueh, 2009), heuristic (Arnold & Tesauro, 2000; Szor, 2008), and behavioral detection (Jacob, Debar, & Filiol, 2008).

As first malware detection approach of VirIT Mobile Security, we decided to opt for signature-based detection. Although reactive in nature - i.e. a malware has to infect a device before it is identified -, and thus it will practically never lead to reach 100% of detection rate, signature-based detection is still the most prevalent approach used to detect and identify malware. This is due to several factors, first of which that it is a consolidate, easily customizable and effective method that, if properly implemented, generates a very low (close to zero) number of false positives - benign files wrongly detected as malicious.



**Figure 3.1:** The home screen of VirIT Mobile Security. The Activity changes dynamically, displaying the relative security-related messages (picture on the left).

Furthermore, signature-based detection systems rely on the consideration that, generally speaking, the more infective a malware is the faster arrives in the hands of security researchers. Thus, even if it does not guarantee perfection, it guarantees the protection from the most widespread threats.

In this sense, it is important to realise that VirIT Mobile Security does not aim to perfectly detect all possible Android malware.

In (F. Cohen, 1987), Frederick B. Cohen proved that there is no algorithm that can perfectly detect all possible/future computer viruses in finite time. Precise detection is undecidable.

According to Cohen, absolute protection can actually be attained by absolute isolationism, which is usually an unacceptable solution. However, as Cohen himself stated, there are methods that can be used to limit undetected spreading.

Limit the undetected spreading of malware is particularly important in order to avoid an epidemic. Indeed, an enormous number of new pieces of malware are produced every day<sup>1</sup> and, as security researchers, we cannot just leave the users without a proper protection. We need to try to fight malware back, even if our detection systems will never be perfect and lead to a 100% detection rate.

Catching the malware as they emerge is even more important in the mobile space, where most of the malware are able to lead to direct loss of money (e.g. by sending SMS mes-

<sup>1</sup> In (Williams, 2012), Mikko Hyppönen (F-Secure) stated that about 200.000 malware arrive in F-Secure office every day. In (Lyne, 2013), James Lyne (Sophos) stated that approximately 250.000 new pieces of malware are created every day.

sages or performing phone calls) and to steal personal and sensitive information (see Section 2.2.2).

Luckily, our detection systems do not necessarily need to be perfect as a single. We can count on multiple systems - or layers of defence - in order to protect a device from being infected and/or to recover from attacks which do manage to get their way into it.

Finally, it is also important to realise that VirIT Mobile Security is a standard Android application and, therefore, it is subject to all the limitations of the current Android application model (see Section 2.1). An example of such limitations is that VirIT Mobile Security is not able to protect the device against exploits targeting specific kernel vulnerabilities.

## 3.2 Related Work

During the years, many works on cyber-security have been carried out. In this section we review some of the most known works in Android malware detection.

So far two main approaches for malware analysis and detection have been proposed: the static analysis (Christodorescu & Jha, 2006; Shabtai, Moskovitch, Elovici, & Glezer, 2009; Schmidt, Camtepe, & Albayrak, 2010) and dynamic analysis (Lee & Mody, 2006; Christodorescu, Jha, & Kruegel, 2008; Bose, Hu, Shin, & Park, 2008; Schmidt et al., 2009). Basically, static analysis is based on code inspection, while dynamic analysis is based on runtime behaviour control. Both the two types of analysis have their own strengths and limitations.

In (Arp, Spreitzenbarth, Hübner, Gascon, & Rieck, 2013), Arp et al. propose DREBIN, a lightweight Android malware detection system which can be run directly on an Android device. DREBIN performs a broad static analysis on both the *AndroidManifest.xml* file and the dex one. From the first one, it gathers the hardware components, the permissions required, the application components (i.e. *Activity*, *BroadcastReceiver*, *ContentProvider* and *Service*) and the filtered *Intents*, while from the dex file it gathers the restricted and the suspicious API calls, the used permissions and the network addresses. All the extracted features are then mapped to a joint vector space, such that typical malware patterns can be automatically detected.

DREBIN, as well as VirIT Mobile Security, is a standard Android application. However, while DREBIN relies only on static analysis and, specifically, only on a single machine learning technique, VirIT Mobile Security takes also advantage of other detection systems, such as signature-based detection and the dynamic analysis of applications' network

usage.

In a similar fashion, in (Shabtai, Kanonov, Elovici, Glezer, & Weiss, 2012), Shabatai et al. present Andromaly, a host-based framework for anomaly detection on Android devices. Andromaly continuously monitors various features and events obtained from the mobile device and then applies data mining anomaly detectors to classify the collected data as either benign or malicious.

Other works have proposed a cloud-based security model, such as Android Application Sandbox (AASandbox) (Blasing, Batyuk, Schmidt, Camtepe, & Albayrak, 2010) and DroidRanger (Y. Zhou, Wang, Zhou, & Jiang, 2012). The first one is a system able to perform both static and dynamical analysis to automatically detect suspicious applications. AASandbox firstly perform a static analysis on the APK package in order to detect malicious patterns. Afterwards, the dynamical analysis is performed in a fully isolated environment. During the dynamical analysis, all the events occurring in the device are monitored.

On the other hand, DroidRanger, by Zhou et al., is a permission-based behavioral footprinting scheme to detect new samples of known Android malware families. In DroidRanger, applications are firstly filtered based on the Android permissions required and, then, an heuristics-based filter is applied.

Finally, in (ForeSafe, 2013) the authors present ForeSafe Mobile Security, which combined static and dynamic analysis. ForeSafe Mobile Security performs static analysis directly on the device, by decompiling the Dalvik bytecode back into the Java bytecode and finally into the Java source code. On the other hand, ForeSafe Mobile Security can perform dynamic analysis on-demand via a cloud sandbox.

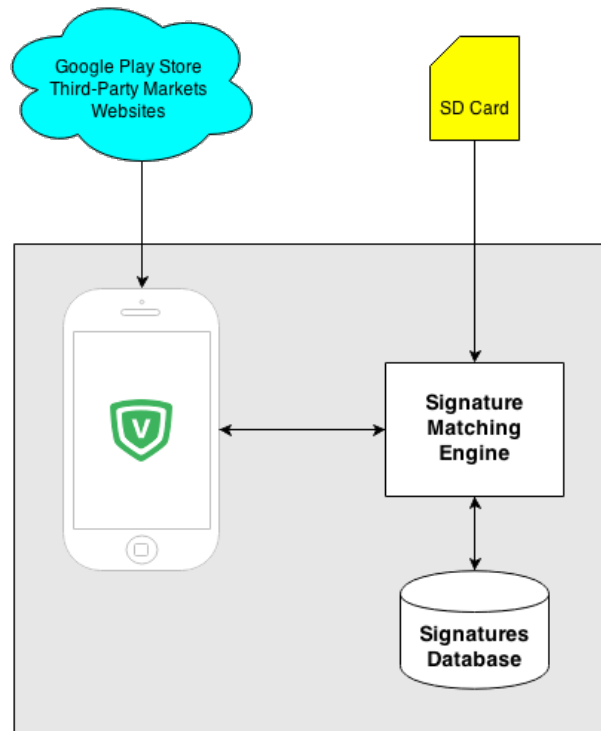
Even if, like VirIT Mobile Security, also ForeSafe Mobile Security uses a signature-based detection, the two works are pretty different. Indeed, at present, our signature-based detection rely only on the binary code. Thus, our engine does not reconstruct the Java source code. Furthermore, VirIT Mobile Security also offers other (on-device) detection mechanisms and, at present, it does not offer a cloud scanner.

### 3.3 Design

In this section we present the design of the signature-based detection system in VirIT Mobile Security.

First we present the malware signatures and, in particular, the string malware signatures. Afterwards, we explain the design choices related to our signatures database and signature

matching engine.



**Figure 3.2:** The architecture of the signature-based detection system in VirIT Mobile Security. Thanks to the on-install scanner, the applications installed from Google Play or other third-party markets (see Seciton 2.1.3) are automatically scanned (after the installation) by the signatures matching engine (see Seciton 3.4.3). The application installed from the SD Card, instead, can be also scanned before the installation. Thanks to the on-demand scanner, the user can always decide to launch a scan on the installed applications and on the External Storage (see Seciton 3.4.2). Finally, thanks to the on-access scanner, each file on the External Storage is scanned every time it is written (see Seciton 3.4.4). For the overall VirIT Mobile Security architecture see Figure 6.1

### 3.3.1 Malware Signatures

As already said, between others, antivirus software relies on signatures in order to properly detect and identify malware.

A signature is a sequence of bytes that represent and identify, with the best accuracy possible (theoretically unequivocally), a particular malware or variant of malware.

Substantially, a malware is analysed (by malware researchers or by dynamic analysis systems), then a signature is extracted and added to the signatures database of the antivirus software. When a particular file has to be scanned, the antivirus compares its content with all the malware signatures in its signatures database. If the file matches one signature, then it is almost surely malicious. Instead, if the file does not match any signature, it cannot be said anything for sure. It might be benign or it might be a malware that has not been yet encountered, and thus analysed, therefore its signature is not yet in the signatures database. These kind of malware are known as zero-day or next-generation malware.

It is clear that the accuracy of a signature-based detection system is heavily based on the correctness of the malware signatures and on the completeness of the signatures database. Indeed, an inaccurate malware signature may lead to the non-detection of a malware or to a false positive - benign files wrongly detected as malicious. On the other hand, the performance of a signature-based detection system depends also on the number of malware signatures in the signatures database.

There are many methods of extracting malware signatures, such as: file hash, byte signature, hex signature, wildcards and regular expressions.

As first signature-based detection method we decided to rely on string signatures. A string signature is made of a variable number of (not consecutive) byte signatures, which are short sequences of contiguous bytes extracted from the malware binary code (e.g. a dex file).

String signatures give many advantages, first of which that string signatures are easy to extract and change (e.g. in case of false positives or to improve an existing signature), and that each string signature can cover several malware files. Furthermore, using string signatures, we can take advantage of current, existing system to automatically generate malware signatures (Ask, 2006; Griffin et al., 2009).

### 3.3.2 Signatures Database

Since our scope is the one of developing a mobile security solution specifically designed for Android platform, we have decided to focus only on malware specific for the Android platform. Thus, no signature of malware for Windows or other platforms has been included. However, it is important to understand that a mobile device may be used as vehicle to carry a malware from one computer to another. For example, the *Backdoor://Android/Ssuecl.A* tries to execute a Windows malware when the infected device is plugged to a PC (F-Secure, 2013; Chebyshev, 2013; AB, 2013). However, in our opinion, adding all the signatures of Windows malware to the signatures database would not be worthy. Indeed, the signatures database would become unnecessarily heavy and slow down the scan. This is an important concern above all when we think about old devices, which does not have a lot of storage capability. A solution to this might be to scan locally for Android malware and to use a cloud scanner to scan for Windows malware. However, since this is out of the scope of this work of thesis, we did not implement such functionality.

### 3.3.3 Signatures Extraction and Signature Matching Engine

In order to properly understand how we extract the malware signatures, we firstly have to understand what we actually can scan in an Android device.

As explained in Section 2.1.1, by design, in Android application's code runs in isolation from other applications. Thus, it is not possible to read another application's memory. All we can do to decide if an application is malicious or not, without modifying the Android Operating System itself, is to scan its APK package saved in the */data/app* directory (see Chapter 2.1).

Since an APK package is basically a ZIP file, a natural question can be raised: what is a good signature of an APK package? Or, in other words, what should we actually scan of an APK package?

One can read, and in turn scan, the APK package as a (stand-alone) binary file or, since it is a package, it is possible to extract and scan its entries - the files contained into the APK package (see Section 2.1.2). However, generally speaking, taking signatures from the binary code of an APK package is not a good strategy. Indeed, such signatures are not robust to (little) changes into the APK packages and, as a result, many signatures will have to be generated for the same malware variants - APK packages where the actual malicious payload is the same or almost the same - leading to a huge signatures database and a slow scan.

Thus, one should extract the signatures from the files within an APK package. But, this raises another question: from which file/s we should extract the signature?

Since in an APK package all the Java classes are compiled into the dex file (see Section 2.1.2), a malware should be detectable from the code in its *classes.dex* file<sup>2</sup>. That means we can theoretically restrict the scan of an APK package to the one of its *classes.dex* file, rather than scanning all the enclosed files (which are usually many). This would significantly reduce the number of comparison the signature matching engine must perform. A similar technique, named filtering technique, is used in many antivirus engine in order to limit the comparison of a particular kind of file only to the subset of signatures that represents malware that typically infect that kind of file (Szor, 2005).

However, by limiting the scan (signatures matching) to only the *classes.dex* file might lead to less accuracy. Indeed, there might be the case in which the antivirus does not have the signature of a particular malware in its signatures database, but it has anyway the one

<sup>2</sup> In future versions of Android the dex file format will probably be replaced with the *oat* one, since ART will probably replace *Dalvik* Virtual Machine (The Android Open Source Project, 2013m). However, since at present this is only an experimental feature and there is still a lack of documentation about *oat* file format, we have preferred not to take this into account. Anyway, our design can be easily modified at any time in order to threat also *oat* files as executables (actually, with *DeepScan* enabled, our scanners already scan also those files).



of an “object” included inside the APK package of that malware. For example, in some malicious applications, the exploit binaries used to root the device are included in the APK package itself (usually in the *assets* directory or in the *res/raw/* one). There might also be the case in which an application “carries” a malicious lib file or another malicious APK inside itself (see *Update attack* in Section 2.2.3). Moreover, in some cases, it might be simpler (and lead to catch more variants of the same malware) to take a signature also on a particular resource file, rather than only on the dex file. This is also important because, in Android, we can stop the scan at the first signature matched, since the only way of cleaning an infection is to remove the entire application, no matter which infection is. Thus, in most of the cases, it is not really important to tell apart two different malware or two different variants of the same malware.

Finally, even if limiting only to the scan of the dex file, it would anyway be a good practice to scan for every dex file inside the APK package. Theoretically, there should be only one of them in each APK package, and only one should matter. However, as recently showed by the Android “Master Key” vulnerability (see Section 2.2.3), this might not be the case. Taking all what we said into account, we have decided to develop two different scan mechanisms: a “fast scan”, in which only the *dex* files will be scanned, and a “full scan” (also known as *DeepScan*). In the latter scan all the entries within an APK package are scanned. Moreover, if a second APK package or a ZIP file is encountered inside the scanned APK, then it is recursively scanned. This theoretically exposes to the risk of *archive bombs* - files that are repeatedly compressed (F-Prot, 2013) - and slows down the scan, but ensure a better accuracy. Finally, in order to accelerate the scan with *DeepScan* enabled, the dex files are anyway scanned firstly. So, if one of them matches a signature, the rest of the APK package entries can be skipped.

## 3.4 Implementation

In this section we present the implementation of the signature-based detection system in VirIT Mobile Security.

First we discuss the implementation of the signatures matching engine and, after, we present three different scanners which make use of the engine: *on-demand*, *on-install* and *on-access* scanners.



### 3.4.1 Signature Matching Engine

Since APK packages are basically ZIP files, in order to scan their entries - comparing them with all the malware signatures in the signatures database -, we need or to unpack and save all the entries in a temporary folder or to directly access to the entries in-memory. It is also possible to use a mix of the two techniques. Indeed, for example, an engine can scan all the APK packages in-memory (see an example in Listing 3.1) but the enclosed multi-level APK package, which can be unpacked and recursively scanned.

```

/*
 * Given:
 * - appDir: the application directory.
 */

final ZipFile zipFile = new ZipFile( new File( appDir ) );

// Retrieve all the entries (both files and folders) in the APK package:
Enumeration<? extends ZipEntry> zipEntries = zipFile.entries();

while ( zipEntries.hasMoreElements() ) {
    final ZipEntry zipEntry = zipEntries.nextElement();

    final String fileName = zipEntry.getName();
    final long fileSize = zipEntry.getSize();
    final InputStream is = zipFile.getInputStream(zipEntry);

    ...
}

zipFile.close();

```

**Listing 3.1:** Example of unzipping an APK package in-memory, using the *ZipFile* class ([The Android Open Source Project, 2013w](#)), in order to access to its entries.

In Listing 3.1 is showed how it is possible to retrieve the *InputStream* ([The Android Open Source Project, 2013l](#)) of the entries in an APK package. Once we access to the *InputStream*, we can read it in order to compare its content with the signatures database (see Listing 3.2).

The *InputStream* class offers the *mark()* method to set a mark location in this stream and the *reset()* method to reset the stream to the last marked location. Furthermore, it offers the *skip()* method in order to skip a given number of bytes.

Since an *InputStream* is a sequential construct (a stream indeed), the *skip()* method is not like jump or seek in a random access. *skip()* actually reads the stream content without returning it ([The Android Open Source Project, 2013n](#)). Moreover, there is no guarantee on how many bytes will actually be skipped. Indeed, the method may choose to skip fewer bytes than requested (but never more).

```

/*
 * Given:
 * - is: the file InputStream.
 * - fileSize: the file size.
 * - offset: byte array containing the signature offsets (ordered).

```

```

    * - buffer: byte array of the same size of the signature.
    */

    final BufferedInputStream bis = new BufferedInputStream( is );

    int fp = 0; // file pointer
    long numBytesSkipped = 0;
    long numBytesToSkip = 0;
    int numBytesRead = 0;

    //Check whether the biggest offset is greater than the file size or not:
    if ( fileSize < offsets[offsets.length-1] ) {
        return; // the file is too small to match the signature!
    }

    for (int i=0; i < offsets.length; i++) {
        try {
            //Jump to the offset [i]:
            while( fp < offsets[i] ) {
                numBytesToSkip = offsets[i] - fp;
                numBytesSkipped = bis.skip(numBytesToSkip);

                fp += numBytesSkipped;
            }

            //Check whether it has jump properly or not:
            if ( fp != offsets[i] ) {
                return; // it did NOT jump properly!
            }

            //Read [buffer.length] number of bytes the in the file starting at offset [i]:
            numBytesRead = bis.read(buffer);

            [...] Compare buffer with the all the malware signatures [...]

            fp += numBytesRead;
        }
        catch ( IOException e ) { ... }
        catch ( IndexOutOfBoundsException e ) { ... }
    }

    //Close the BufferInputStream:
    bis.close()

```

**Listing 3.2:** Example of scanning a file from its *InputStream* ([The Android Open Source Project, 2013l](#)). The *InputStream* class offers the *mark()* method to set a mark location in this stream and the *reset()* method to reset the stream to the last marked location. Furthermore, it offers the *skip()* method in order to skip a given number of bytes. However, since an *InputStream* is a sequential construct (a stream indeed), the *skip()* method is not like jump or seek in a random access. *skip()* actually reads the stream content without returning it. Moreover, there is no guarantee on how many bytes will actually be skipped. Indeed, the method may choose to skip fewer bytes than requested (but never more).

Clearly scanning a file from its *InputStream* is not a robust enough solution. Indeed, especially with big files, the *skip()* method may need to be called many times in order to reach the desired location in the file. Of course, from a program-flow prospective, we know we will reach that desired location and, since before starting the offsets comparison we check that the greatest offset is actually inside (smaller than) the file size, we do not need to worry about reaching the End of File (EoF). However, from a performance prospective, since we are going to compare every file with a huge number of malware signatures, we may want to prevent to be stuck in a cycle of too many *skip()* calls for each comparison. Definitely, limiting the maximum number of *skip()* calls is not a solution, since it may lead not to arrive to the desired location, resulting in the impossibility to properly scan the file.

Thus, what we need to achieve is to have a random access to the file. This is possible, for example, using the *RandomAccessFile* class ([The Android Open Source Project, 2013q](#)). As shown in Listing 3.3, scanning a *RandomAccessFile* is a robust and fast solution.

```

/*
 * Given:
 * - fp: the RandomAccessFile instance.
 * - fileSize: the file size.
 * - offset: byte array containing the signature offsets (ordered).
 * - buffer: byte array of the same size of the signature.
 */

int numBytesRead = 0;

//Check whether the biggest offset is greater than the file size or not:
if ( fileSize < offsets[offsets.length-1] ) {
    return; // the file is too small to match the signature!
}

for (int i=0; i < offsets.length; i++) {
    try {
        //Jump to the offset [i]:
        fp.seek(offsets[i]);

        //Read [buffer.length] number of bytes the in the file starting at offset [i]:
        numBytesRead = fp.read(buffer);

        [...] Compare buffer with all the malware signatures [...]
    }
    catch ( IOException e ) {...}
    catch ( IndexOutOfBoundsException e ) {...}
}

//Close the RandomAccessFile:
fp.close();

```

**Listing 3.3:** Example of scanning a *RandomAccessFile* ([The Android Open Source Project, 2013q](#)). This solution is much more robust than the one of scanning the file from its *InputStream* (see Listing 3.2).

However, if we are starting from an *InputStream* (for example because we are unzipping the APK package in-memory using the *ZipFile* class), we need to turn it into a *RandomAccessFile* before. This might not be as smooth as before. Indeed, in order to do that, we need to create a temporary (cached in our case) file and open it in random access (see Listing 3.4).

Even if creating temporary files will slow down the scan, in our opinion, scanning a *RandomAccessFile* is much more robust solution than scanning a file *InputStream*. Furthermore, the more signatures there will be in the signatures database the better this solution will be in comparison with the other one. Indeed, we will initially lose time creating the temporary copy of the file, but then we will have a random access to the file - which is a lot faster than having to go through the stream for every signature comparison (as state previously, *skip()* actually reads the stream content without returning it ([The Android Open Source Project, 2013n](#))).

```

/*
 * Given:
 * - is: the file InputStream.

```

```

    * - fileName: the file name.
    * - fileSize: the file size.
    */

    final File tempFile = File.createTempFile(fileName, null, getCacheDir());
    final RandomAccessFile fp = new RandomAccessFile(tempFile, "rw");

    //An array has an integer size (we do not want integer overflow here...):
    if ( fileSize > Integer.MAX_VALUE ) {
        fileSize = Integer.MAX_VALUE;
    }

    byte[] buffer = new byte[fileSize];
    int numBytesRead = 0;

    while ( (numBytesRead = is.read(buffer)) != -1 ) {
        fp.write(buffer, 0, numBytesRead);
    }

    fp.seek(0);

    return randomAccessFile;

```

**Listing 3.4:** Example of converting *InputStream* to *RandomAccessFile*. This is needed, for example, if we unzipped the APK package in-memory using the *ZipFile* class (see Listing 3.1). This solution creates temporary (cached) files, and this will slow down the scan. However, in our opinion, scanning a *RandomAccessFile* is much more robust solution than scanning a file *InputStream*. Furthermore, the more signatures there will be in the signatures database the better this solution will be in comparison with the other one. Indeed, we will initially lose time creating the temporary copy of the file, but then we will have a random access to the file - which is a lot faster than having to go through the stream for every signature comparison (as state previously, *skip()* actually reads the stream content without returning it).

One particular warning for mobile space is that a mobile application does not usually have as much available (heap) memory as a desktop program. Thus, the code shown in Listing 3.4 to convert an *InputStream* into a *RandomAccessFile* object may fail at the byte array (buffer) instantiation throwing an *OutOfMemoryError* exception. This kind of exception is thrown when a request for memory is made that cannot be satisfied using the available platform resources ([The Android Open Source Project, 2013o](#)). This will happen especially when we need to deal with files that have a “large” size (sometimes it is enough a size greater than 2 MB). A solution to this problem can be to programmatically assign the size of the buffer based on the space we can actually allocate (see Listing 3.5). To speed up the check, we also previously cut down all the sizes we already know will fail.

```

/*
 * Given:
 * - fileSize: the file size.
 */

byte[] buffer;

while ( true ) {
    try {
        buffer = new byte[fileSize]; // you need to cast to int here!
        return buffer;
    } catch ( OutOfMemoryError e ) {
        //Reduce the memory allocation (of a factor of 10):
        if ( fileSize > 10 ) {
            fileSize /= 10;
        }
        else {
            if ( fileSize > 1 ) {
                fileSize = 1;
            }
        }
    }
}

```

```

        }
        else {
            throw new IOException(); // what the hell?!
        }
    }
}

```

**Listing 3.5:** Dynamically allocating a byte array. This is particularly important in mobile space, because a mobile application does not usually have as much available (heap) memory as a desktop program. Thus, the code shown in Listing 3.4 to convert an *InputStream* into a *RandomAccessFile* object may fail at the byte array (buffer) instantiation throwing an *OutOfMemoryError* exception. This kind of exception is thrown when a request for memory is made that cannot be satisfied using the available platform resources.

### 3.4.2 On-Demand Scanner

In order to let the user to manually scan Android applications, we decided to start implementing an *on-demand scanner* (see Figures 3.3 and 3.4), which makes use of the signatures matching engine presented in Section 3.4.1, firstly focusing on scanning the installed applications.

#### Scanning the installed applications

Retrieving the list of applications currently installed on an Android device - whether they are user or system applications (see Section 2.1.1) - is possible using the *PackageManager* class, through which is also possible to retrieve various kinds of information related to them. ([The Android Open Source Project, 2013p](#))

As shown in the Listing 3.6, in order to retrieve the list of installed applications, one needs to call the *getInstalledApplications()* method of the *PackageManager* class. The method returns a list of *ApplicationInfo* objects. Each of these objects represents an installed application, from which we can retrieve many useful information, such as the application name, its package and its directory. ([The Android Open Source Project, 2013e](#))

```

final PackageManager pm = getPackageManager();
final List<ApplicationInfo> listOfInstalledApps = pm.getInstalledApplications(PackageManager.GET_META_DATA);

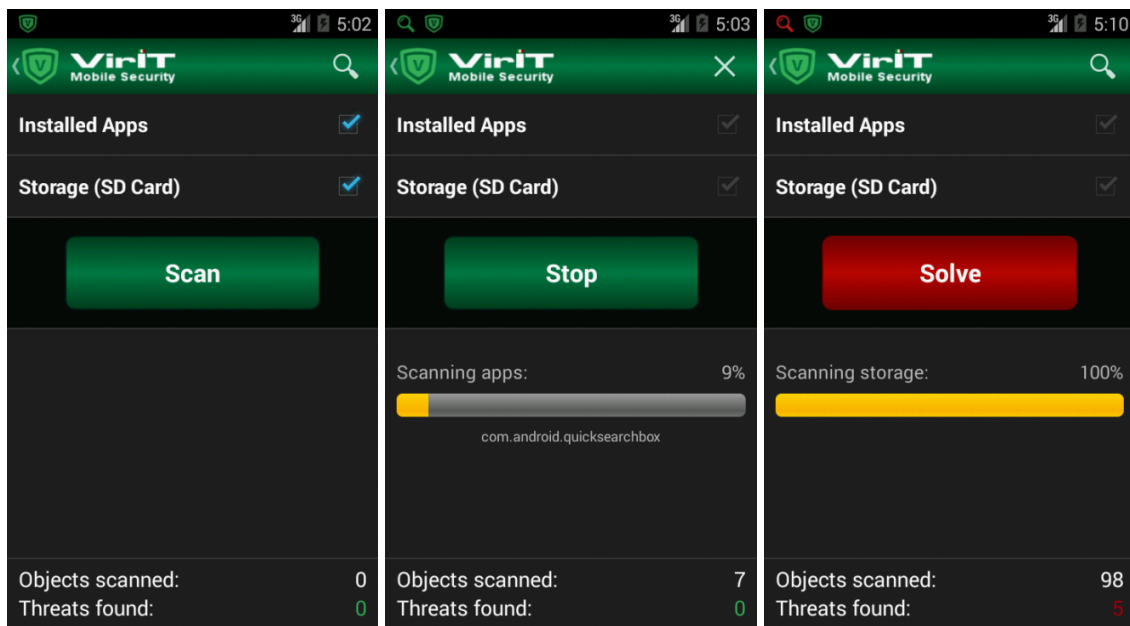
for ( final ApplicationInfo ai : listOfInstalledApps ) {
    final String appName = pm.getApplicationLabel(ai).toString();
    final String appPackage = ai.packageName;
    final String appDir = ai.sourceDir;

    ...
}

```

**Listing 3.6:** Example of how to retrieve the list of installed applications in Android. The *getInstalledApplications()* method of the *PackageManager* class returns a list of *ApplicationInfo* objects. Each of these objects represents an installed application, from which we can retrieve many useful information.

In particular, once we have an application APK package directory, we call the proper method of our signatures matching engine (see Section 3.3.3) in order to scan all its entries by comparing them with all the malware signatures in the signatures database (see



**Figure 3.3:** The on-demand scanner of ViriT Mobile Security. A scan is performed before on the installed applications and after on the External Storage. In case some malicious applications are found, the “Solve” button will appear in order to show to the user the list of threats found (see Figure 3.4)

### Chapter 3.3.1). Scanning the storage

Once we were able to scan the installed applications, we extended our on-demand scanner to the External Storage ([The Android Open Source Project, 2013s](#)). In Android, under this umbrella term are grouped external SD cards and possible internal non-removable storages.

Although, in Android, the External Storage should be always mounted as non-executable ([The Android Open Source Project, 2013j](#)), it might be used from applications to save/download temporary files, including APK packages and/or exploit binaries. Furthermore, a user (or a malware) can upload an APK package on the SD card directly from a computer when the device is connected via USB.

In Android, it is possible to retrieve the primary External Storage directory of a device using the `getExternalStorageDirectory()` method of the *Environment* class ([The Android Open Source Project, 2013i](#)). However, the External Storage (if exists) may not be always accessible ([The Android Open Source Project, 2013s](#)). Indeed, it may happen that it has been removed from the device or mounted on a computer, or some other problem has happened. Therefore, before accessing it, we need to check its current state. This can be done with the `getExternalStorageState()` method of the *Environment* class (see Listing 3.7).

Furthermore, we might have more than one External Storage. For example, we can be in the case of having one external SD Card and one internal non-removable storage. In the Android API there is no reference of a standard method to retrieve all the External Storages. However, like in every Unix/Linux system, all the mounting storage devices are

stored/linked in a common directory. For Android this is usually the */mnt* directory (the standard Unix/Linux directory for mounting storage devices) or, in the newest versions, the */storage* one. In any case, as shown in Listing 3.7, we can easily retrieve this “root” directory from the primary External Storage one (it will be its parent). If not empty, every of its subdirectory - but the USB drive ones - will be a different External Storage.

```
final List<File> storageDirectories = new ArrayList<File>();
String storageRootDirectory = null;

//Retrieve the primary External Storage:
final File primaryExternalStorage;
final String primaryExternalStorageDirectory = System.getenv("EXTERNAL_STORAGE");
if ( (primaryExternalStorageDirectory != null) && !primaryExternalStorageDirectory.equals("") ) {
    primaryExternalStorage = new File(primaryExternalStorageDirectory);
}
else {
    primaryExternalStorage = Environment.getExternalStorageDirectory();
}

//Check whether the primary External Storage is a real/readable directory or not:
if ( (primaryExternalStorage != null) && primaryExternalStorage.isDirectory() &&
    primaryExternalStorage.canRead() && (primaryExternalStorage.length() > 0) ) {
    storageRootDirectory = primaryExternalStorage.getParent();

    //Add the primary External Storage directory:
    storageDirectories.add( primaryExternalStorage );
}

//Retrieve all the secondary External Storage:
final String secondaryExternalStorageDirectories = System.getenv("SECONDARY_STORAGE");
if ( (secondaryExternalStorageDirectories != null) && !secondaryExternalStorageDirectories.equals("") ) {
    if ( secondaryExternalStorageDirectories.contains(":") ) { // more than one secondary External Storage...
        for ( final String storage : secondaryExternalStorageDirectories.split(":") ) {
            if ( (storage != null) && !storage.equals("") ) {
                final File file = new File(storage);

                //Check whether it is a real/readable directory or not:
                if ( file.isDirectory() && file.canRead() && (file.listFiles().length > 0) ) {
                    //Add all the subdirectory in the External Storage root directory:
                    if ( !storageDirectories.contains(file) ) {
                        storageDirectories.add(file);
                    }
                }
            }
        }
    }
    else { // only one secondary External Storage...
        final File file = new File(secondaryExternalStorageDirectories);

        //Check whether it is a real/readable directory or not:
        if ( file.isDirectory() && file.canRead() && (file.listFiles().length > 0) ) {
            //Add all the subdirectory in the External Storage root directory:
            if ( !storageDirectories.contains(file) ) {
                storageDirectories.add( file );
            }
        }
    }
}
else {
    if ( (storageRootDirectory != null) && !storageRootDirectory.equals("") ) { // there's a parent directory...
        final File storageRoot = new File( storageRootDirectory );

        //Check whether the root storage folder is a real/readable directory or not:
        if ( storageRoot.isDirectory() && storageRoot.canRead() && (storageRoot.length() > 0) ) {
            final File[] files = storageRoot.listFiles();

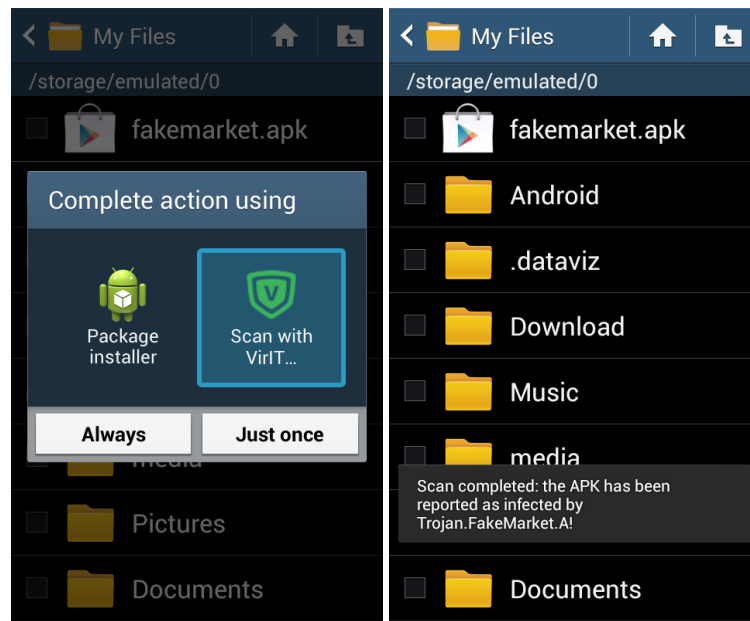
            if ( (files != null) && files.length > 0 ) { // there actually is at least one file...
                for ( final File file : files ) { // for each file into the External Storage root directory...
                    //Check whether it is a real/readable directory or not:
                    if ( file != null && file.isDirectory() && file.canRead() && file.listFiles().length > 0 ) {
```





Due to platform design, in Android, it is not possible to interact with the installation process.

As shown in Listings 3.8 and 3.9, when one tries to install an APK package from the



**Figure 3.5:** The on-install scanner of VirIT Mobile Security, which scans the applications before they are installed from the External Storage. In the pictures, the on-install scanner detects the installation of the *Trojan://Android/FakeMarket.A*, which pretends to be the Google Play store application but runs silently in the background to perform click fraud. (Rovelli, 2014; AndroTotal, 2014)

External Storage, it is actually possible to scan an application before it is installed.

```
<activity android:name="com.example.onInstallPackageActivity" android:label="..." android:excludeFromRecents="true">
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:scheme="content" />
    <data android:scheme="file" />
    <data android:mimeType="application/vnd.android.package-archive" />
  </intent-filter>
</activity>
```

**Listing 3.8:** (XML) Example of *Activity* to scan an application before it is installed, when it is installed from the SD Card.

```
public class onInstallPackageActivity extends Activity {
    /**
     * Start up the Activity.
     *
     * @param savedInstanceState if the activity is being re-initialized after previously being shut down
     * then this Bundle contains the data it most recently supplied in onSaveInstanceState(Bundle).
     * Otherwise, it is null.
     */
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        try {
            //Retrieve the local Intent:
            final Intent localIntent = getIntent();

            if ( localIntent != null ) {
                final Uri localUri = localIntent.getData();

                if ( localUri != null ) {
                    //Retrieve the package directory:
```

```

        final String appDir = localUri.getEncodedSchemeSpecificPart();

        [...] Scan the APK package [...]
    }
}
}
catch ( RuntimeException e ) { ... }
}
}

```

**Listing 3.9:** (Java) Example of *Activity* to scan an application before it is installed, when it is installed from the SD Card.

However, this is not possible when an application is installed from Google Play Store or another third-party markets. All we can do is to register and handle the event that a new application has been installed. Therefore, our on-install scanner will detect a malware only after it has been installed. However, it should be able to detect it before it can do harm.

In order to “hook” our scanner to the installation process, we create and register a *BroadcastReceiver* for the *PACKAGE\_ADDED* and *ACTION\_PACKAGE\_REPLACED* broadcast actions (see Listing 3.10). The first action is broadcasted every time a new application package has been installed on the device, while the latter every time a new version of an application package has been installed, replacing an existing version that was previously installed.

Once the *BroadcastReceiver* is triggered, the information on the installed package are extracted and a scan on the APK package is performed (see Listing 3.11).

```

<receiver android:name="com.example.onInstallBroadcastReceiver" android:exported="false">
  <intent-filter android:priority="1000">
    <action android:name="android.intent.action.PACKAGE_ADDED" />
    <action android:name="android.intent.action.ACTION_PACKAGE_REPLACED" />
    <data android:scheme="package" />
  </intent-filter>
</receiver>

```

**Listing 3.10:** (XML) Example of *BroadcastReceiver* that handles the on-install event, thanks to the *PACKAGE\_ADDED* and *ACTION\_PACKAGE\_REPLACED* broadcast actions. The first action is broadcasted every time a new application package has been installed on the device, while the latter every time a new version of an application package has been installed, replacing an existing version that was previously installed.

```

public class onInstallBroadcastReceiver extends BroadcastReceiver {
    /**
     * Receiving an Intent broadcast.
     *
     * @param context the Context in which the receiver is running.
     * @param intent the Intent being received.
     */
    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();

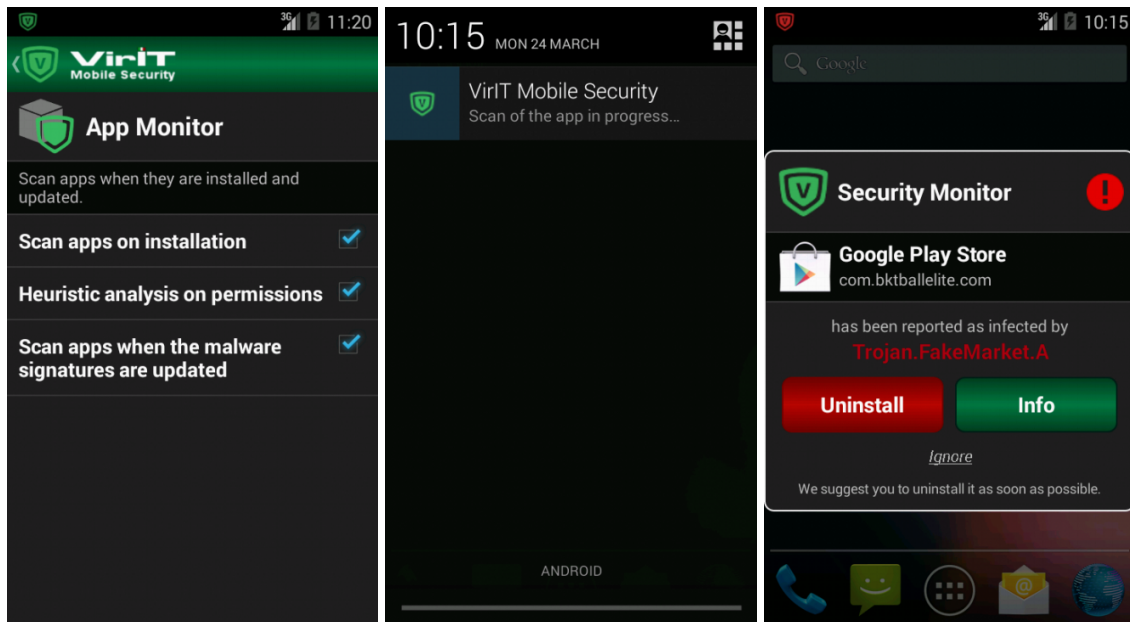
        if ( action.equals( Intent.ACTION_PACKAGE_ADDED ) || action.equals( Intent.ACTION_PACKAGE_REPLACED ) ) {
            // Retrieve the installed/updated package:
            final String appPackage = intent.getData().getEncodedSchemeSpecificPart();

            [...] Scan the APK package [...]
        }
    }
}

```

---

**Listing 3.11:** (Java) Example of *BroadcastReceiver* that handles the on-install event, thanks to the *PACKAGE\_ADDED* and *ACTION\_PACKAGE\_REPLACED* broadcast actions. The first action is broadcasted every time a new application package has been installed on the device, while the latter every time a new version of an application package has been installed, replacing an existing version that was previously installed.



**Figure 3.6:** The on-install scanner of ViriT Mobile Security, which scans the applications at installation. In the picture on the right, the on-install scanner detects the installation of the *Trojan://Android/FakeMarket.A*, which pretends to be the Google Play store application but runs silently in the background to perform click fraud. (Rovelli, 2014; AndroTotal, 2014)

### 3.4.4 On-Access Scanner

In order to complete the layer of security provided by our mobile antivirus, we decided to introduce a last scanner: an *on-access scanner* (see Figure 3.7). This automatically scans the files in the External Storage at creation and at any subsequent modification. The benefits/limits of such a scanner are the same discussed in Section 3.4.2 for the on-demand scanner on the External Storage.

In Android it is possible to use the *FileObserver* class to monitor the files into a specific directory. This is based on the Linux *inotify* files change notification system. Events will be triggered when a file is accessed or changed by any process on the device. (The Android Open Source Project, 2013k)

In particular, between all the events that will be triggered by the *FileObserver*, we are interested by the *CREATE* event, triggered when a new file or subdirectory is created under the monitored directory, and by the *MODIFY* and/or *CLOSE\_WRITE* events, which are respectively triggered when data are written to a file under the monitored directory and when someone had a file or directory open for writing and closed it (See Listing 3.12).

A *FileObserver* instance monitors only the (first-level) files and subdirectories inside a given directory. However, the sub-files and sub-subdirectories recursively contained into the subdirectories are not monitored. Therefore, we need to extend the *FileObserver* class in order to (recursively) monitor also the file in the subdirectories. Alternatively, it is possible to instantiate a *FileObserver* for each directory and recursively for their subdirectory in the External Storage. In any case, as shown in Listing 3.12, when a new directory is created into a monitored directory, we need to add it to the monitored directories.

```
//Instantiate and register the FileObserver:
MyFileObserver fileObserver;
fileObserver.startWatching();

private class MyFileObserver extends FileObserver {
    private static final int mask = FileObserver.ALL_EVENTS;
    private String path;

    /**
     * Class Constructor.
     *
     * @param directory the directory to be monitored/observed.
     */
    public ExternalStorageObserver(String directory) {
        super(directory, mask);

        this.path = directory;
    }

    /**
     * A content change occurs.
     *
     * @param event the event code (e.g. FileObserver.CREATE, FileObserver.CLOSE_WRITE or FileObserver.MODIFY).
     * @param fileDir the directory of the modified file.
     */
    @Override
    public void onEvent(int event, String fileDir) {
        if (fileDir == null) {
            return;
        }

        final String fileAbsoluteDir = this.path + "/" + fileDir;

        //Check whether a new file was created under the monitored directory:
        if ( (FileObserver.CREATE & event) != 0 ) {
            final File file = new File(fileAbsoluteDir);

            //Check whether the file is a directory:
            if ( file.isDirectory() ) {
                [...] Add the new directory to the monitored directories [...]
            }
            else {
                if ( file.length() > 0 ) {
                    [...] Scan the new file [...]
                }
            }
        }

        //Check whether data was written to a file:
        if ( (FileObserver.CLOSE_WRITE & event) != 0 ) {
            final File file = new File(fileAbsoluteDir);

            if ( !file.isDirectory() && file.length() > 0 ) {
                [...] Scan the modified file [...]
            }
        }

        [...]
    }
}
```

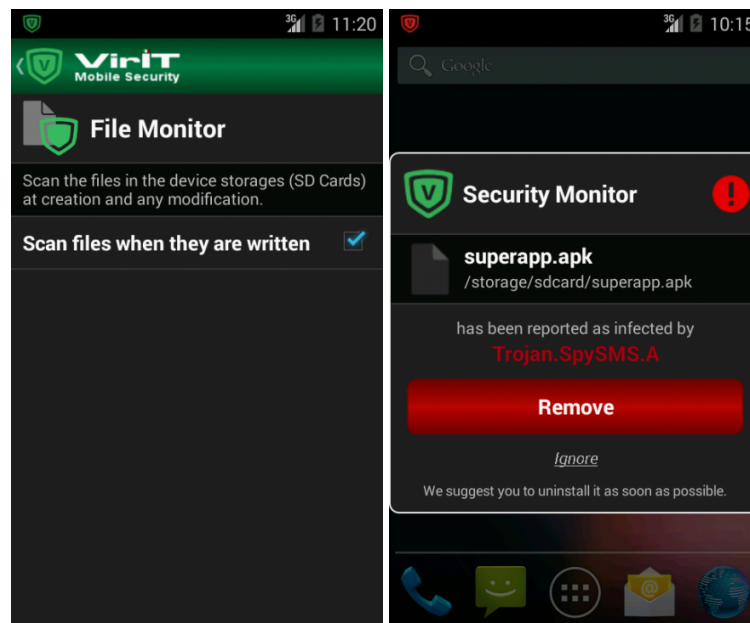
```

    }
}

```

**Listing 3.12:** Example of *FileObserver* used to monitor the files into a specific directory. Events will be triggered when a file is accessed or changed by any process on the device. In particular, between all the events that will be triggered by the *FileObserver*, we are interested by the *CREATE* event, triggered when a new file or subdirectory is created under the monitored directory, and by the *MODIFY* and/or *CLOSE\_WRITE* events, which are respectively triggered when data are written to a file under the monitored directory and when someone had a file or directory open for writing and closed it.

Since many Android applications use to save log files (and edit them frequently), the on-access scanner will continuously scan all these log files at any modification, resulting in an intensive use of the CPU and a consequent quick discharge of the device battery. A possible solution to this problem may be the one of keeping a queue of the recently scanned files in order to prevent that a file is scanned twice or more in a reduce amount of time. However, this may lead not to scan a malicious file properly because it is changed twice in the same “time-frame”.



**Figure 3.7:** The on-access scanner of VirIT Mobile Security, which scans the files in the External Storage at creation and any modification. In the picture on the right, the on-access scanner detects that the *Trojan://Android/SpySMS.A* has been written in the External Storage (precisely in the SD Card root directory: */storage/sdcard*).

## 3.5 Evaluation

In order to see the real effectiveness of an antivirus software, we want to test its robustness as well as its resilience (Hawes, 2013). With the term robustness we mean the ability of our solution to prevent Android malware from penetrating a device at all. On the other

Mobile Security solution	TP	Detection Rate
AV-1	132	89.19%
AV-2	122	82.43%
AV-3	146	98.65%
AV-4	138	93.24%
VirIT	142	95.95%

**Table 3.1:** Results of the robustness experiments on VirIT Mobile Security. With the term robustness we mean the ability of our solution to prevent Android malware from penetrating a device at all. In the table, for all the antivirus solutions tested, we point out the True Positives (TP) (i.e. the number of malware detected out of a total of 148 malware on the device) and the corresponding detection rate.

hand, with the term resilience we mean the ability of our solution to recover from zero-day or next-generation malware which do manage to get installed into a device.

In this section we present the results of the robustness test, while in Section 6.5 we present the results of the resilience test.

It is important to understand that such an experiment is definitely not to be considered complete in order to state the robustness of our solution. Indeed, the dataset we used was rather too small to asses the overall robustness of VirIT Mobile Security solution. However, it can give a rough idea.

### 3.5.1 Environment

In order not to be bias and to make an effective real-world protection test, we decided to separately collect a dataset from various web sources and markets. We have collected a total of 1200 samples.

In order to estimate the approximate number of benign and malicious applications in the dataset, we scan the dataset with four different antivirus solutions. We find out that our dataset was composed by 148 malicious applications, of which 125 were detected by at least two antiviruses and 23 were detected by only one antivirus (no matter which one), and 1052 applications considered benign - i.e. not detected by anyone of the antivirus.

Of course, in order to perform a real-world protection test, we did not study further these malicious applications nor we add their signatures to our signatures database.

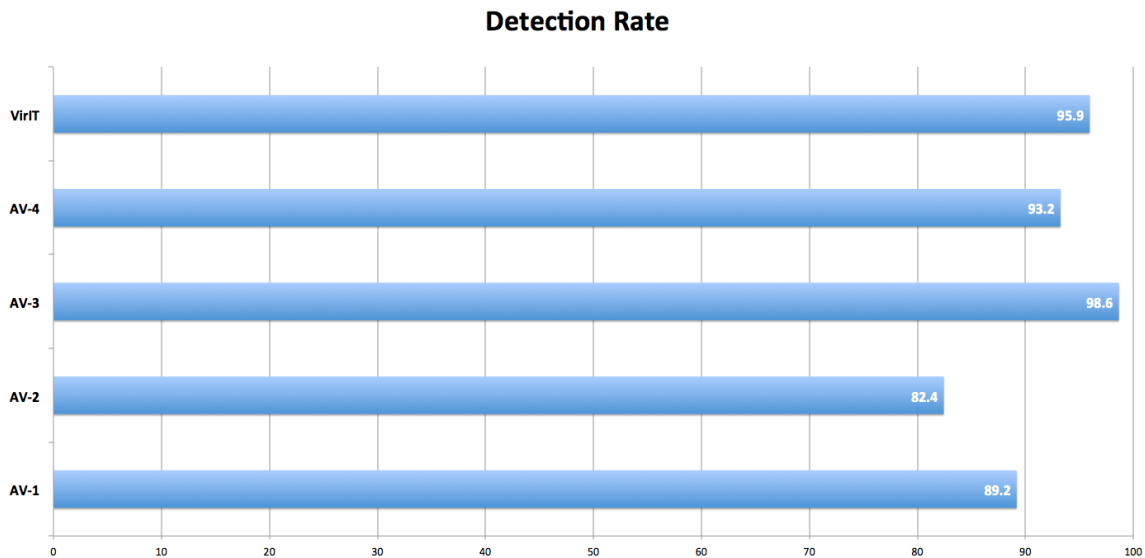
### 3.5.2 Results

After collecting the samples, we start installing them on an Android emulator in which VirIT Mobile Security was already installed.

As showed in Table 3.1 Figure 3.8, the *on-install scanner* successfully detects and re-

moves 142 malicious applications, 123 of those labelled as malicious by at least two antiviruses and 19 of those labelled as malicious by only one antivirus. That means a detection rate of 95.95%.

Even if the results were good, it is important to note that a robustness test such as this



**Figure 3.8:** Testing the robustness of the signature-based detection system in VirIT Mobile Security. With the term robustness we mean the ability of our solution to prevent Android malware from penetrating a device at all. We tested VirIT Mobile Security with a dataset of 1200 samples, 148 labelled as malicious applications and 1052 labelled as benign ones.

one it is not enough to state the real protection provided by a mobile security solution. Indeed, such a test shows only the malware detection rate of the different antivirus in a particular moment in time. But, what about the next days/months/years? And, above all, what about the malware that were not detected? We address these and other questions in Section 6.5, where we actually evaluate the complete VirIT Mobile Security solution rather than only its signature-based detection system.

## 3.6 Conclusion

In this chapter we have introduced the malware detection problem and presented a signature-based detection solution specifically designed for Android.

Our signature matching engine has proven effective, being able to detect 95.95% of the malware in our test. However, signature-based detection systems, like most of the existing security systems, are reactive approaches. This means that a malware has to infect a device before it is identified and, in turn, properly blocked/removed.

Clearly, in order to have a strong detection, we cannot exclusively rely on signatures. We

need to have a second detection mechanism that works alongside the signature-based detection in order to spot zero-day or next-generation malware as they emerge. This kind of approach is known as proactive approach.

We introduce this second kind scanners in Chapter 5 (heuristic) and in Section 6.4 (behavioural detection).



## Chapter 4

# Implementing a Real-Time Monitor in Android

In this chapter we introduce the Android real-time malware detection problem (see Section 4.1) and we discuss different possible solutions, with their benefits and drawbacks (see Section 4.2).

### 4.1 Introduction

In Android, as shown in Chapter 3, it is possible to develop an *on-demand scanner* (see Section 3.4.2) and an *on-install scanner* (see Section 3.4.3) for Android applications. However, the layer of security provided by these two scanners might not be enough, especially if a user does not perform complete scans often enough.

For example, we can think about the unlucky but recurrent situation in which a zero-day or next-generation malware - i.e. a malicious application which is not detected by the antivirus - does manage to get installed into a device. Now, even if the malware eventually arrives to the security firm which develops the antivirus (see Section 6.4), a proper signature of the malware is extracted and the antivirus is actually up-to-date, the malware will not be detected and, thus, removed until an on-demand scan will be performed.

Besides unacceptable, since it exposes the user to the risk of an extended infection, this is clearly caused by the fact that we did not implement a real-time detection/protection mechanism. Furthermore, an extended infection is even more dangerous in the mobile space, where malware may have the ability to send SMS messages and/or to perform phone calls (see Section 2.2.2), resulting in high financial losses for the user.

Unfortunately, at present, there is no framework-provided way to register a callback when an application is launched/run and, thus, to develop a real-time scanner for Android applications. However, some possible workarounds can be used in order to provide a real-time protection for the user.

## 4.2 Design and Implementation

In this section we discuss several different possible real-time detection mechanism.

First we present two solutions which are currently used by some commercial mobile security products, i.e. the *on-execution scanner* (Section 4.2.1) and the *on-update scanner* (Section 4.2.2), of which we actually implemented only the second one. Afterwards, we propose a novel concept known as “*live scanner*” and we discuss a possible design for a *cloud scanner*.

### 4.2.1 On-Execution Scanner

One possible solution to provide a real-time protection is the one of scanning the running applications recursively, every once in a while.

In Android, indeed, it is possible to schedule an application to be run at some point in the future thanks to *AlarmManager*. Basically, we can set up an *AlarmManager* and, when the alarm goes off, a registered *Intent* is broadcast by the system, automatically starting the target application if it is not already running. ([AlarmManager, 2013](#))

Thus, using *AlarmManager*, we can set up a *Service*, to be run every 5 or 10 minutes, in order to retrieve and scan the running applications.

Retrieving the list of applications currently running on an Android device - whether they are user or system applications - is possible using the *ActivityManager* class, through which is also possible to interact with the overall activities running in the system. ([The Android Open Source Project, 2013a](#))

As shown in the Listing 4.1, in order to retrieve the list of running applications, one needs to call the *getRunningAppProcesses()* method of the *ActivityManager* class. The method returns a list of *RunningAppProcessInfo* objects. Each of these objects represents a running application process, from which we can retrieve many useful information, such as the process name, its pid and all the packages that have been loaded into the process. ([The](#)

## Android Open Source Project, 2013b)

```
final ActivityManager activityManager = (ActivityManager) context.getSystemService( Activity .ACTIVITY_SERVICE);
final List<ActivityManager.RunningAppProcessInfo> runningApps = activityManager.getRunningAppProcesses();

if ( runningApps != null ) {
    for ( final ActivityManager.RunningAppProcessInfo runningApp : runningApps ) {
        final String processName = runningApp.processName;
        final int processPID = runningApp.pid;
        final String[] appPackages = runningApp.pkgList;

        ...
    }
}
```

**Listing 4.1:** Example of how to retrieve the list of running applications in Android. The *getRunningAppProcesses()* method of the *ActivityManager* class returns a list of *RunningAppProcessInfo* objects. Each of these objects represents a running application process.

Of course, the main advantage of this solution is that (with a low enough time frame between one scan and the next one) it really provides a real-time protection. Indeed, after the mobile security is updated with a proper signature of a previously unknown malware, the latter will be detected, and thus removed, as soon as it will be executed (that is immediately if it is already running).

However, this solution is not designed taking into account the particular Android architecture and, therefore, it has many drawbacks. First of all, running a *Service* constantly (or anyway that often) is both CPU and battery expensive. Furthermore, most of the scans performed will actually be useless. Indeed, assuming that an on-install scanner is correctly working on the device (see Section 3.4.3), the installed application packages will not be changed from one “on-execution” scan to the next one (see Chapter 3.3.3). Or, in other words, if an application is not detected in a first scan it will not be detected in a second one either, and so on so forth until the signatures database will be updated.

The number of applications scanned at every on-execution scan can be significantly reduced. For example, one can keep a queue of the recently scanned applications in order to prevent that an application is scanned twice or more in a reduce amount of time. Of course, in order to properly re-scan applications after they have been updated, a signature of the application (such as the *MD5* or *SHA-256* hash) should be stored. Finally, in order to provide real-time protection, the queue of the recently scanned applications should be cleaned after every update of the signatures database. However, also with these improvements, an on-execution scanner will still perform more scans than the one actually needed.

### 4.2.2 On-Update Scanner

As we stated, an on-execution scanner introduces a lot of redundancy, or in other words it performs more scans than the one needed. Indeed, a scan should be performed only when the malware signatures actually change. Therefore, a more efficient solution is the one of scanning all the installed applications automatically after every update of the signatures database.

After every update of the malware signatures of our mobile security, we can launch a scan on the installed applications. The list of applications currently installed on an Android device - whether they are user or system applications - can be easily retrieved as shown in Section 3.4.2.

As well as the on-execution scanner, also this solution provides a real-time protection. Indeed, a previously unknown malware will be removed as soon as the Mobile Security will be updated with a proper signature of the malware.

However, the performance of this solution clearly depends on the number of updates released per day/week. Generally speaking, in a situation of no more than one update a day, an *on-update scanner* would be less CPU and battery expensive than an *on-execution scanner*. Indeed, even if an on-update scan is performed on all the installed applications, this is performed only when new malware signatures are actually available and, thus, previously unknown malware can now be detected.

The main drawback of this solution is that the more updates will be released, the more CPU and battery expensive the scanner will be. In a situation in which several small updates a day are released, which should be the best strategy in a security prospective, the performance of an on-update scanner will be much worst than the one of an on-execution scanner.

In order to guarantee the maximum effectiveness being the less resource-intensive as possible, in VirIT Mobile Security, we opt for an on-update scanner which performs a “fast scan” - only the *dex* files are scanned (see Section 3.3.3) - on all the installed applications (see Figure 3.6).

### 4.2.3 Live Scanner

As we stated, the performance of an on-update scanner will be good as long as we do not release more than one update every second day or, at most, every day. In order to tackle the real-time protection problem and to improve the performance of the *on-update scanner* in situations of multiple updates a day, we have designed a novel concept known as “*live scanner*”.

Basically, given an amount of  $N$  installed applications, after every update of the signatures database, we randomly select a small percentage of the  $N$  installed applications (such as the 5 or 10%) and scan it. Of course, there is no certainty that, even if a malicious application is installed on the device, it will actually be selected to be scanned. Given  $N$  the total number of installed applications and knowing that one and only one of them is malicious, the probability of selecting the malicious application in one extraction is:

$$p = \frac{1}{N}$$

Then, the probability of selecting one in  $k$  random extractions (rounds) is:

$$\begin{aligned} P[\textit{selecting\_the\_malware\_in\_k\_rounds}] &= \\ &= P[\textit{not\_selecting\_all\_benign\_apps\_in\_k\_rounds}] = 1 - (1 - p)^k \simeq 1 - e^{-pk} \end{aligned}$$

That, if  $N$  is equal to 100 (installed applications) and  $k$  is equal to 10 (we randomly select the 10% of them), is:

$$p = \frac{1}{100} = 0.01$$

$$P[\textit{selecting\_the\_malware\_in\_k\_rounds}] = 1 - (1 - p)^{10} = 1 - \left(\frac{99}{100}\right)^{10} \approx 0.0956 \approx 9.56\%$$

Thus, having 100 installed applications, if we consider to extract and scan the 10% of them after every update, we will approximately have the 9.56% of probability to detect the malware at each update.

In order to have at least  $1 - \varepsilon$  probability of selecting the malware, we must set the variable  $k$  such that:

$$1 - e^{-pk} \geq 1 - \varepsilon$$

Which implies that:

$$-pk \leq \log(\varepsilon)$$

Or, in other words, that:

$$k \geq \frac{1}{p} \log\left(\frac{1}{\varepsilon}\right)$$

Thus, if we want to have a probability of at least 50% of selecting the malware:

$$k \geq \frac{1}{p} \log(2)$$

That, if  $N$  is equal to 100, is:

$$k \geq 100 \log(2) \Leftrightarrow k \geq 69.3147$$

Thus, in order to have a probability of about 50% of selecting the malware, we need to select approximately the 68-69% of the total amount of installed applications. This is a quite high amount given our initial motivation.

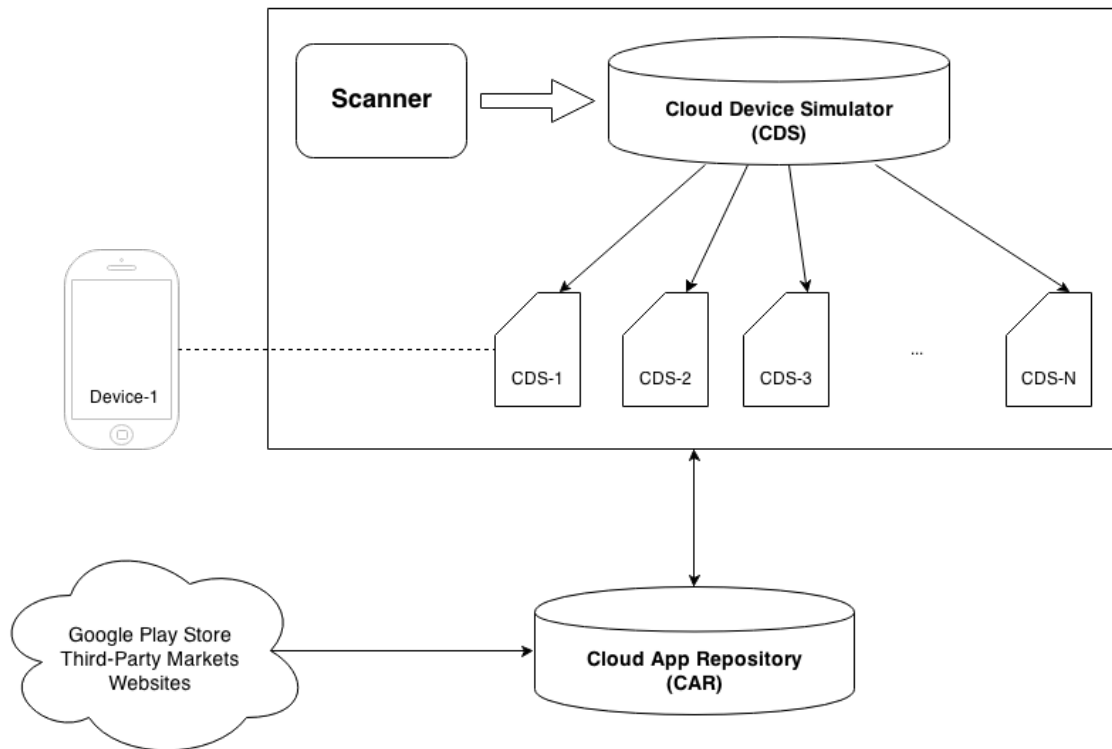
However, in all these calculation, we did not take into account several factors. First of all, we designed the live scanner in order to handle multiple updates per day without compromising the resource as it would happen with an on-update scanner. Thus, we assume we are in a situation in which the live scan will be repeated many times a day and, accordingly, the likelihood of selecting the malware should increase at each scan. Hence, theoretically, having around 25% of likelihood for a single scan should be good enough. Indeed, with such a probability we should already be able to “catch” the malware after few updates/scans. Furthermore, the likelihood of selecting at least a malware is dramatically higher in the situation in which there is more than one in the device.

#### 4.2.4 Cloud Scanner

An even better workarounds, completely different from all the previous one, is the one of storing all the applications installed in a device remotely on a web server and scan them through a *cloud scanner*. (Oberheide, Cooke, & Jahanian, 2007, 2008; Martignoni, Paleari, & Bruschi, 2009)

A cloud scanner has several advantages, first of which the fact that it is not subjected to the Android architecture design/limitations. Thus, with a cloud scanner we can perform a more accurate analysis, eventually resulting in a higher detection rate. A second main advantage is the fact that most of the resources are not spent on the device itself, and this in turn improve the battery lifetime.

However, developing a cloud scanner for a mobile device poses two main issues: we cannot trust the mobile devices to be connected round-the-clock and we should take care of using as less network traffic as possible.



**Figure 4.1:** A possible cloud scanner implementation is the one of dividing the so called Cloud Device Simulator (CDS), which will contain the virtual representations of the subscribed devices, and the Cloud App Repository (CAR), which will contain all the APK packages of known Android applications. Basically, a CDS instance contains the list of references of the applications installed on the real device at any moment in time. The scanner will scan all the applications installed by each CDS instance, retrieving the actual APK packages from the CAR. When a malware is detected on a CDS instance, this will be immediately reported to the corresponding real device.

For example, a possible design for mobile space, that should work alongside a local (on-device) mobile security application, is the one of developing a Cloud Device Simulator (CDS) and a Cloud App Repository (CAR). The CDS will contain one instance for each subscribed device. A CDS instance represents a virtual model of the device. Basically, a CDS instance contains the list of all the applications installed in the device at any moment in time. However, this is just a list of references (e.g. MD5 or SHA-256 hashes). On the contrary, the actual APK packages and their behaviours (e.g. benign, malicious or unknown) are all stored into the CAR, which is basically a database of all the known Android application. In this way we can eliminate duplicate APK package transfer, storing and analysis.

When a user install or uninstall an application, the local mobile security application signals this event to the CDS. In particular, in the case of the installation of an unknown application - an application that has not been encountered before and, thus, it is not present

into the Cloud App Repository -, the CDS will ask to the local mobile security application to upload also the actual APK package to the CAR.

On the server, of course, all the applications installed by each CDS instance will be scanned at least after every malware signatures update. The scanner will retrieve the actual APK packages from the CAR.

Since, as already said, a cloud scanner has the advantage of not being subjected to the Android architecture design/limitations, ideally, we can also take advantage of dynamic analysis and sandboxes.

If the cloud scanner, at some point, detects a malware installed on a CDS instance, this will be immediately reported to both the CAR and the corresponding real devices.

As already said, one issue of a cloud scanner for mobile devices is the overall network traffic generated. In the presented cloud scanner, it should be clear that the more subscribed devices the CDS has and the more likely will be that an application has already been uploaded into the Cloud App Repository. However, it is possible to lighten further the network traffic required by each device, for example, by using web services that run on the CAR server and continuously download new Android applications from various sources, such as: Google Play Store, third-party markets (see Section 2.1.3) and other websites.

## 4.3 Evaluation

In this section we discuss the effectiveness (i.e. detection rate on the known malware) and the maximum infection time frame (i.e. the maximum number of days a malware will last after a proper malware signature update has been released) of the discussed solutions. This is because, these two values are the ones that shows if our solution is actually able to detect malware in real-time.

As shown in Table 4.1, weekly and daily scans cannot be considered real-time detection mechanisms. Indeed, even if their detection rate on the known malware is 100%, they expose the user to remain infected for a period of time up to 7 and 1 day/s, respectively.

Also the *on-execution scanner* does not guarantee that a malware, which managed to get installed into the device, will be removed immediately after a proper signature is released. However, it provides a reasonably enough real-time protection, since the malware will be detected, and thus removed, the first time it will be executed (assuming it will not be in execution for less than the time frame between one on-execution scan and the next one).

Both the *on-update scanner* and the *cloud scanner* guarantees a real-time protection. In-



	Resource-Intensive	Redundancy	Effectiveness	Maximum Infection Time Frame [days]
Weekly Scan	No	No	100%	7
Daily Scan	Yes	No ( $\leq 1$ update/day) Yes ( $> 1$ update/day)	100%	1
On-Execution Scanner	Yes	Yes	100%	*The first time the malware will be executed
On-Update Scanner	No ( $< 1$ update/day) Yes ( $\geq 1$ update/day)	No	100%	0
Live Scanner	No	No	$< 100\%*$ *Depending on the configuration	$\geq 0*$ *Depending on the configuration
Cloud Scanner	No	No	100%	0

**Table 4.1:** An overview of the presented real-time protection mechanisms. For each model, it is shown: whether it requires significant system resources or time (i.e. resource-intensive), whether it performs more scans than the ones needed (i.e. redundancy), the detection rate on the known malware (i.e. effectiveness) and the maximum number of days a malware will last after a proper malware signature update has been released (i.e. maximum infection time frame).

deed, a malware which managed to get installed into the device will be removed as soon as a proper signature will be released.

Finally, since both the effectiveness and the maximum infection time frame of the *live scanner* depends on how it is configured, we decided to run some evaluation tests in order to asses them (see Secitons 4.3.1 and 4.3.2).

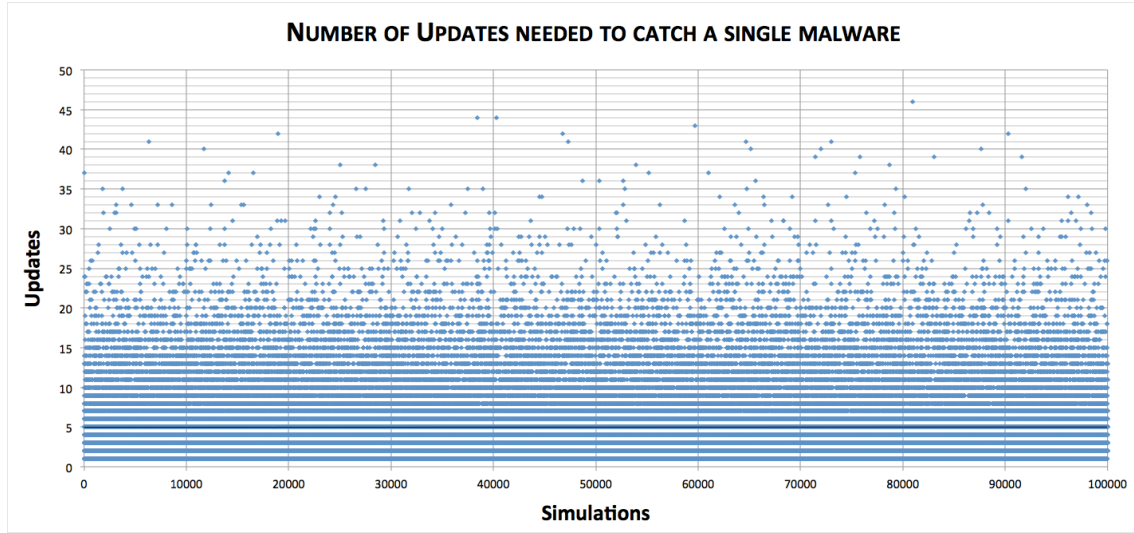
### 4.3.1 Environment

In order to evaluate the efficiency of the live scanner, we simulate a real device with a total number of 265 applications, of which 215 system applications and 50 user applications respectively. Then, we run different simulations on it in order to see how long zero-day or next-generation malware will last on the device after a proper update has been released. Therefore, in our experiments, we assume that:

- One or more next-generation malware have been previously installed on the device;
- We have received the samples of those malware and updated the signatures database of our mobile security;
- The mobile security is up-to-date.

Moreover, we decided to test 5 different configurations of the live scanner, 10 to 50% of the installed applications, increasing with steps of 10%. Therefore, respectively: 27, 53, 80, 106 and 133 distinct applications will be selected after each update of the signatures database.

Our simulation continuously scan the device with the live scanner, keeping trace of how many scans are needed in order to detect the presence of the malicious applica-



**Figure 4.2:** Live Scanner: Number of updates needed to catch one malware in the 100.000 simulations with the configuration of the live scanner at 20%. In a situation in which we release one update a day, this is equal to the number of days needed to catch one malware. The complete results are shown in Table 4.2.

tions. These, in turn, will be the number of updates needed before the zero-day or next-generation malware is actually detected.

We run the simulation 100.000 times and, then, we calculate the average of scans needed and the highest number as well. Furthermore, we run the simulation for different weights - i.e. different number of malicious applications in the device.

### 4.3.2 Results

In Table 4.2, and in Figures 4.3 and 4.4, are shown the results for the average and worst case.

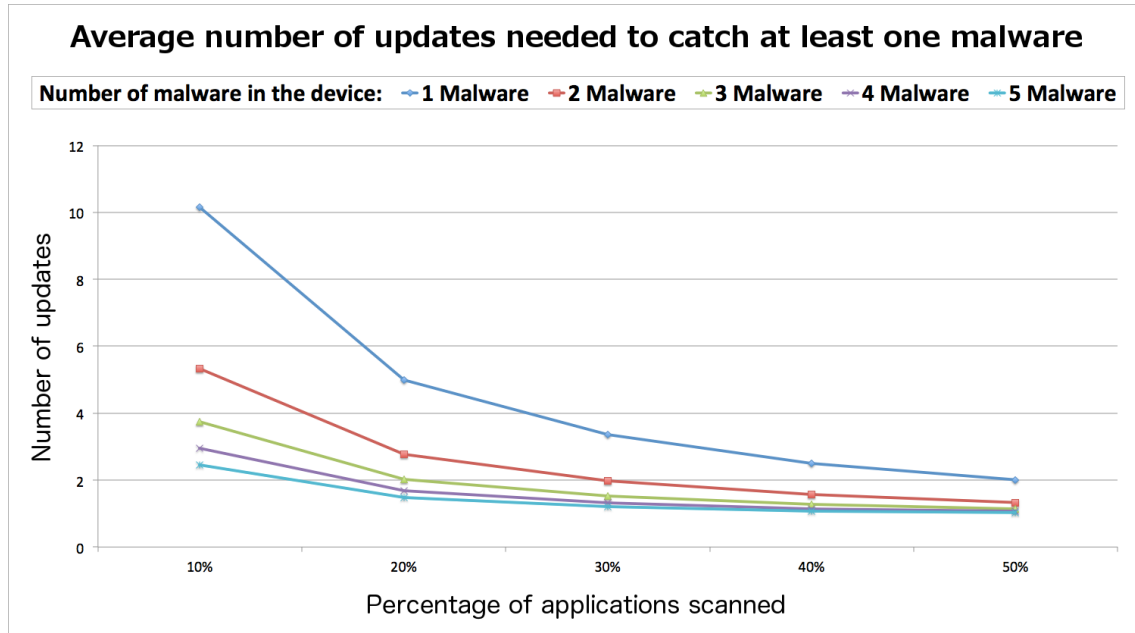
In average, having only one next-generation malware installed on the device and setting the live scanner in order to scan the 10% of the installed application, the live scanner needs about 10 updates before catching the malware. This is absolutely too much. By setting the live scanner at 20% (see Figure 4.2), we reduce this number at about 5 updates. This means that, even releasing only couple of updates per day, we will need less than three days to properly detect the malware presence. By setting the live scanner at 30%, we are able to catch the malware after about 3 updates, which seems a perfect compromise.

As we expected, the results are even better if more than one malware is installed on the device. Furthermore, in case one malware is found by the live scanner, a user might decide to launch a full scan (we might also design the live scanner in order to suggest this or to do it automatically).

However, if we look at the worst case (see Table 4.2), the results are not so good. Indeed,

Number of malware in the device	Number of scanned applications	Number of updates (average / worst case) needed to catch				
		1 malware	2 malware	3 malware	4 malware	5 malware
1 malware	10%	10.16 / 113	-	-	-	-
	20%	4.99 / 46	-	-	-	-
	30%	3.35 / 32	-	-	-	-
	40%	2.49 / 22	-	-	-	-
	50%	2.01 / 17	-	-	-	-
2 malware	10%	5.33 / 57	10.70 / 65	-	-	-
	20%	2.76 / 30	5.56 / 40	-	-	-
	30%	1.97 / 20	3.94 / 20	-	-	-
	40%	1.56 / 11	3.13 / 17	-	-	-
	50%	1.33 / 9	2.67 / 14	-	-	-
3 malware	10%	3.75 / 42	7.47 / 57	11.21 / 53	-	-
	20%	2.03 / 19	4.9 / 22	6.13 / 27	-	-
	30%	1.52 / 11	3.05 / 16	4.57 / 17	-	-
	40%	1.27 / 8	2.54 / 9	3.82 / 13	-	-
	50%	1.14 / 7	2.28 / 8	3.43 / 9	-	-
4 malware	10%	2.95 / 29	5.89 / 34	8.82 / 47	11.77 / 52	-
	20%	1.69 / 13	3.37 / 18	5.06 / 19	6.75 / 21	-
	30%	1.32 / 9	2.64 / 13	3.94 / 16	5.27 / 18	-
	40%	1.15 / 7	2.29 / 8	3.44 / 10	4.58 / 12	-
	50%	1.07 / 4	2.14 / 7	3.20 / 8	4.26 / 10	-
5 malware	10%	2.46 / 23	4.94 / 28	7.40 / 32	9.85 / 35	12.32 / 45
	20%	1.48 / 10	2.96 / 13	4.45 / 16	5.92 / 18	7.41 / 21
	30%	1.20 / 6	2.40 / 9	3.61 / 12	4.81 / 13	6.01 / 15
	40%	1.08 / 6	2.16 / 7	3.25 / 9	4.33 / 9	5.41 / 11
	50%	1.03 / 4	2.06 / 5	3.09 / 6	4.13 / 8	5.16 / 9

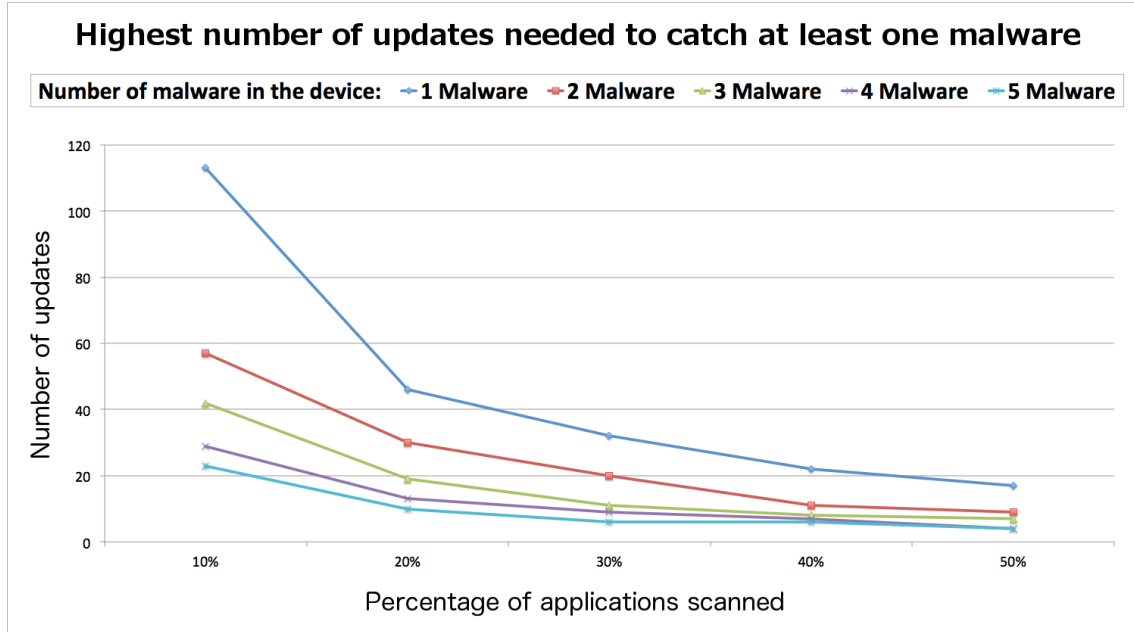
**Table 4.2:** Results of the live scanner experiments. In order to evaluate the efficiency of the live scanner, we simulate a real device with a total number of 265 installed applications. We set the live scanner in order to scan 10, 20, 30, 40 and 50% of the application after every update respectively. Then, we run the simulation 100.000 times and, finally, we calculate the average (approximated) and the highest number of scans/updates needed to catch the malware. In a situation in which we release one update a day, this is equal to the number of days needed to catch one malware.



**Figure 4.3:** Live Scanner: Average number of updates/scans needed to catch at least one malware in relation to the percentage of application scanned for each update. The various lines show the number of malware in the device, from 1 (blue line) to 5 (light blue line). In a situation in which we release one update a day, the number of updates is equal to the number of days needed to catch one malware. The complete results are shown in Table 4.2.

with only one next-generation malware installed, even setting the live scanner at 50% there has been a simulation in which the live scanner needed 17 updates in order to detect

the malware presence. By setting the live scanner at 30%, the registered worst case was 32 updates. This means that, even releasing five updates per day, we will need roughly one week to detect the malware presence. That is clearly not an option.



**Figure 4.4:** Live Scanner: Highest number of updates/scans needed to catch at least one malware in relation to the percentage of application scanned for each update. The various lines show the number of malware in the device, from 1 (blue line) to 5 (light blue line). In a situation in which we release one update a day, the number of updates is equal to the number of days needed to catch one malware. The complete results are shown in Table 4.2.

## 4.4 Related Work

As already stated, there is surprisingly little research on the Android real-time protection problem.

As far as we know, nowadays, only one commercial mobile security solution offers an *on-execution scanner* (Section 4.2.1), while there are couple of commercial mobile security solutions which actually offer an *on-update scanner* (Section 4.2.2). Most of the others mobile security solutions on the market seem to rely only on weekly scans.

On the other hand, most of the academic literature has been focused on cloud scanners. Examples of these scanners are *ThinAV* (Jarabek, Barrera, & Aycock, 2012) and *Paranoid Android* (Portokalidis, Homburg, Anagnostakis, & Bos, 2010).

ThinAV, proposed by Jarabek et al., is a cloud-based antivirus solution for Android that uses pre-existing web-based scanning services for malware detection. The main advantages of ThinAV is that it is a cheap and lightweight solution, and it does not generate an

excessive network traffic. However, it suffers of a main drawback: it needs to modify the Android architecture and, thus, it cannot be installed directly on an Android device as an application.

Paranoid Android, proposed by Portolakidis et al., is a cloud security architecture where analysis is performed on remote servers that host replicas of the mobile devices in virtual environments. Also Paranoid Android does not generate an excessive network traffic. The authors have shown that, even during periods of high activity (e.g. browsing), the transmission overhead can be kept well below 2.5KiBps.

Both ThinAV and Paranoid Android are related to the possible cloud scanner implementation proposed in Section 4.2.4. Indeed, even if our implementation was primarily designed in order to handle the real-time detection problem, the architectures of both ThinAV and Paranoid Android are somehow similar to the ours. Furthermore, in Paranoid Android, the authors use synchronised replica of real devices which, even if different in design and implementation, is a similar concept of our Cloud Device Simulator (CDS) instances.

## 4.5 Conclusion

In this chapter we have introduced the Android real-time malware detection problem and we have discussed four different possible solutions. All the discussed solutions have their own benefits and drawbacks.

The *on-execution scanner* provides a reasonable enough real-time protection. However, this solution introduces a lot of redundancy (i.e. it performs more scans than the ones needed) and, as a consequence, it is resource-intensive (it requires significant system resources or time).

The *on-update scanner* guarantees that previously unknown malware will be removed as soon as the mobile security will be updated with proper signatures. However, the performance of this solution depends on the number of updates released per day/week. The more updates will be released, the more resource-intensive the scanner will be.

The *live scanner* has shown to be a feasible alternative to the on-update scanner in situations of multiple updates a day, but it needs to be improved. At present, in a worst case perspective, the only feasible configuration with no more than 5 updates a day is to scan the 50% of the installed applications. As already stated, a good and easy first improvement should be the one of scanning the remaining installed applications in case one malware is found.

Finally, the *cloud scanner* has shown to be the overall best solution, even if the more

expensive one to develop. As already stated, the proposed solution should be designed to work alongside a local (on-device) mobile security application, in order to provide real-time protection as well as to increase the detection capabilities and to reduce the resource requirements.

## Chapter 5

# PMDS: Permission-based Malware Detection System for Android

In this chapter we propose a novel Android malware detection technique, called Permission-based Malware Detection System (PMDS), which has been later integrated into VirIT Mobile Security (see Figures 5.3, 5.1 and 5.4) in order to provide heuristic detection on zero-day or next-generation malware which are not detected by the signature-based detection system (see Chapter 3).

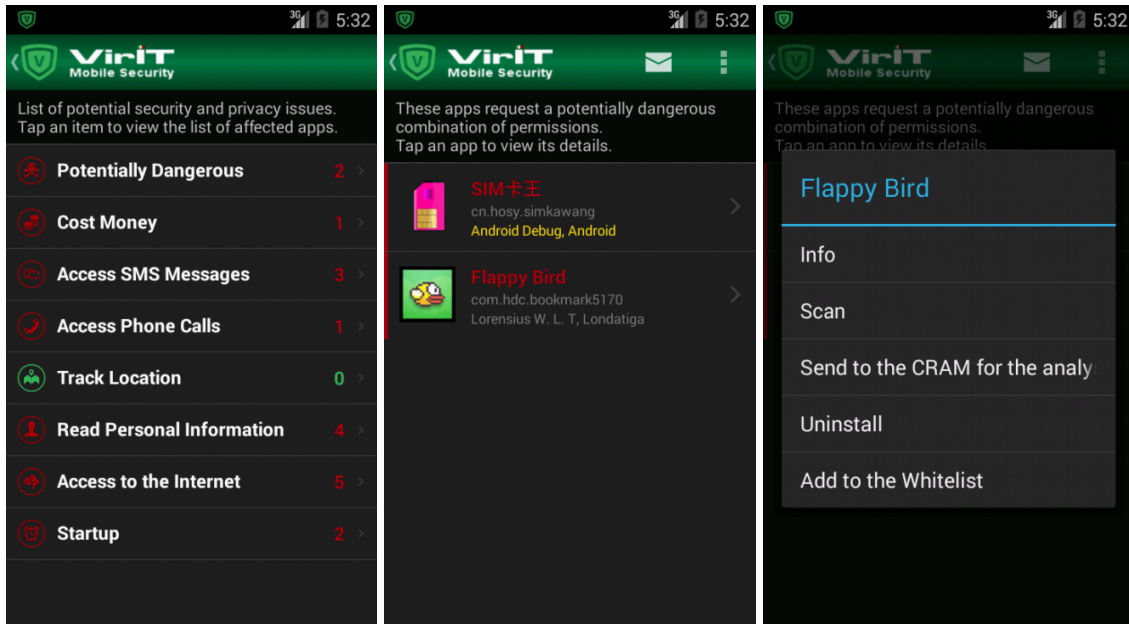
Experimental results show that our PMDS was able to detect more than 94% of previously unseen malware.

### 5.1 Introduction

In order to provide the best protection possible for the end user, we have to take advantage of everything. However, if we want to improve the state-of-the-art of mobile security, we cannot restrict our work in just catching/spotting “bugs” in malware implementation.

For example, we can (and should) take advantage of the typical patterns of SMS messages used by botnets, or of the typical code mistakes that malware authors make in Android applications (e.g. declaring a permission twice in the *AndroidManifest.xml* file or set the priority of an *intent-filter* at more than the maximum permitted value - i.e. 1000). However, bypassing these kinds of rule-based detection is pretty simple for cyber-criminals, since they just need to change the patterns their botnets use or have a better look at the Android API.

Therefore, we also need to seek the “flows” in malware implementation. Because these



**Figure 5.1:** The Privacy Advisor of VirIT Mobile Security, which makes use of the PMDS rule-based (RIPPER) classifier in order to retrieve and display the potentially dangerous applications installed on the device. As shown in the picture on the right, an application can be manually whitelisted (so that it will not display as potentially dangerous in future scans), scanned with the on-demand scanner (see Section 3.4.2) and/or sent to the Anti-Malware Research Center of TG Soft (CRAM) for the analysis. In the picture at the centre, the Privacy Advisor reports both the *Trojan-Spy://Android/Wapsx.A* (the first of the list) and the *Trojan-SMS://Android/FakeFlappyBird.A* (the second of the list) as possibly dangerous.

are the main limitations a cyber-criminal might not be able to work around. For example, a repackaged application will always have some differences with the original one or, similarly, an application that wants to have access to particular system components will always have to ask for specific permissions.

This is the main reason that led us to the design and implementation of the Permission-based Malware Detection System (PMDS) (see Chapter 5.3), which takes advantage of the group of permissions declared by an application - which in Android represents the group of actions an application can perform (see Section 5.3.1) - in order to classify its behaviour as either benign or malicious.

## 5.2 Related Work

Besides traditional malware detection methodologies (e.g. signature-based detection), there is an increasing amount of attempts to apply machine learning and data mining techniques to detect new or unknown malicious code. These, however, have been concentrated mostly on Microsoft Windows malware. (Siddiqui, Wang, & Lee, 2008; Ye, Wang, Li, & Ye, 2007; Schultz, Eskin, Zadok, & Stolfo, 2001; Kolter & Maloof, 2006; Tabish, Shafiq, & Farooq, 2009; Kiem, Thuy, & Quang, 2004; Firdausi, Lim, Erwin, & Nugroho,



2010; Dua & Du, 2011)

On the other hand, there is also an increasing amount of attempts to detect Android malware using permissions, even if most of them does not make use of machine learning.

In (Huang, Tsai, & Hsu, 2013), Huang et al. explore the possibility of detecting malicious applications using permissions. In order to retrieve the permissions, the authors disassemble the APK packages, identify the invoked Android system functions and, then, reconstruct the permissions used. To evaluate their detection model, the authors use a dataset of 124.769 benign applications and 480 malicious ones, and 4 machine learning algorithms, respectively: *AdaBoost*, *Näive Bayes*, *Decision Tree* and *Support Vector Machine*. However, in order to help the detection mechanism, the authors use several other features (e.g. the number of particular file formats and both the number of under-privileged and over-privileged permissions) in additions to the permissions. The authors claim that their experiments show that a single classifier is able to detect about 81% of malicious applications.

Although there are some similarities, the latter approach is mainly different from ours. Indeed, the main aim of our work is the one of exploring the possibility of using the Android permissions, founded in the *AndroidManifest.xml* file, in order to enhance our existing signature-based detection system, rather than the one of creating a stand-alone malware detection system based on the permissions used.

Other related works that take advantage of permissions are DroidRanger (Y. Zhou et al., 2012), a permission-based behavioural footprinting scheme to detect new samples of known Android malware families, and (Sarma et al., 2012), which use probabilistic generative models to compute a risk score depending on the permissions required by an application. Both the two works, however, are different from ours since they do not use machine learning algorithms.

## 5.3 Design

In this section we present the design of our Permission-based Malware Detection System (PMDS).

### 5.3.1 Permissions as (possible) behavioural markers

In the Android architecture (see Section 2.1.1), each application has access only to the components that it requires to do its work and no more. In other words, an application

that wants to have access to particular system components will always have to ask for specific permissions. This creates a very secure environment in which an application cannot access parts of the system for which it is not given permission ([The Android Open Source Project, 2013d, 2013t](#)).

All the permissions an application requires have to be explicitly declared in the *AndroidManifest.xml* file ([The Android Open Source Project, 2013t](#)), which is an entry file in APK packages that provides semantic-rich information about the application itself and its components (see Section 2.1.2).

It is important to note that, since each permission is related to an action, the permissions required by an application can be seen as a marker of its (possible<sup>1</sup>) behaviour, or at least of part of it.

However, it is also important to realise that not all the malware actually ask for a danger-

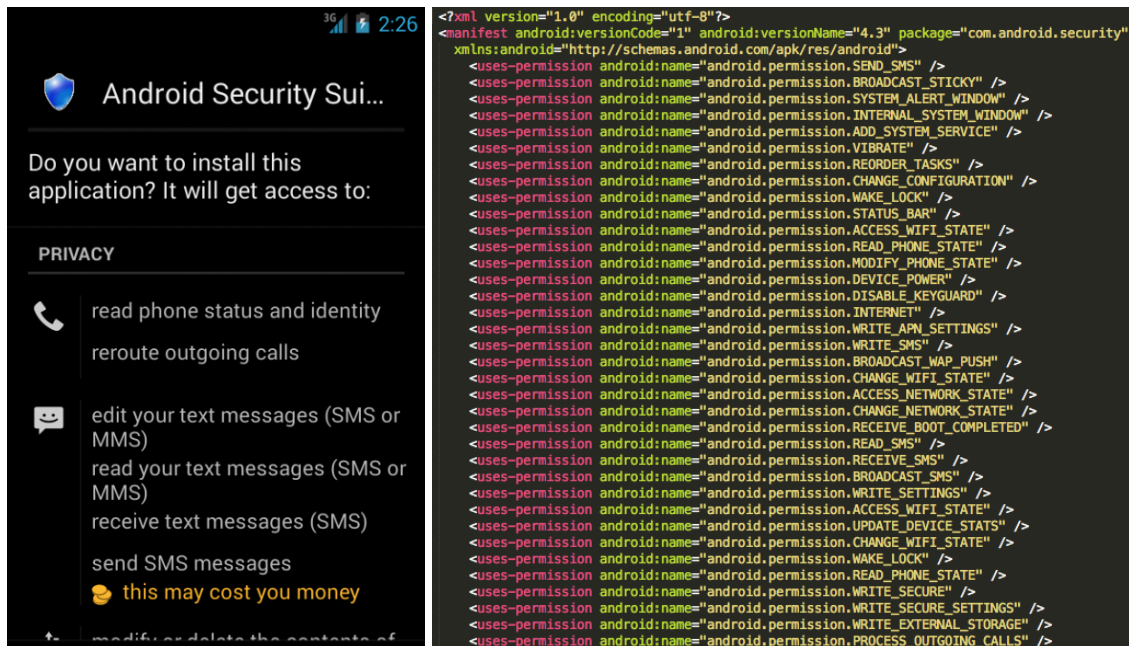


Figure 5.2: Example of a malicious application (*Trojan-Banker://Android/ZitMo.B*) which requires for a specific group of permissions. On the right the permissions are required during the installation process, while on the left the *AndroidManifest.xml* file in which the required permissions are declared.

ous combination of permissions. Some might also not ask for any permission at all (see *Update* attack in Section 2.2.3).

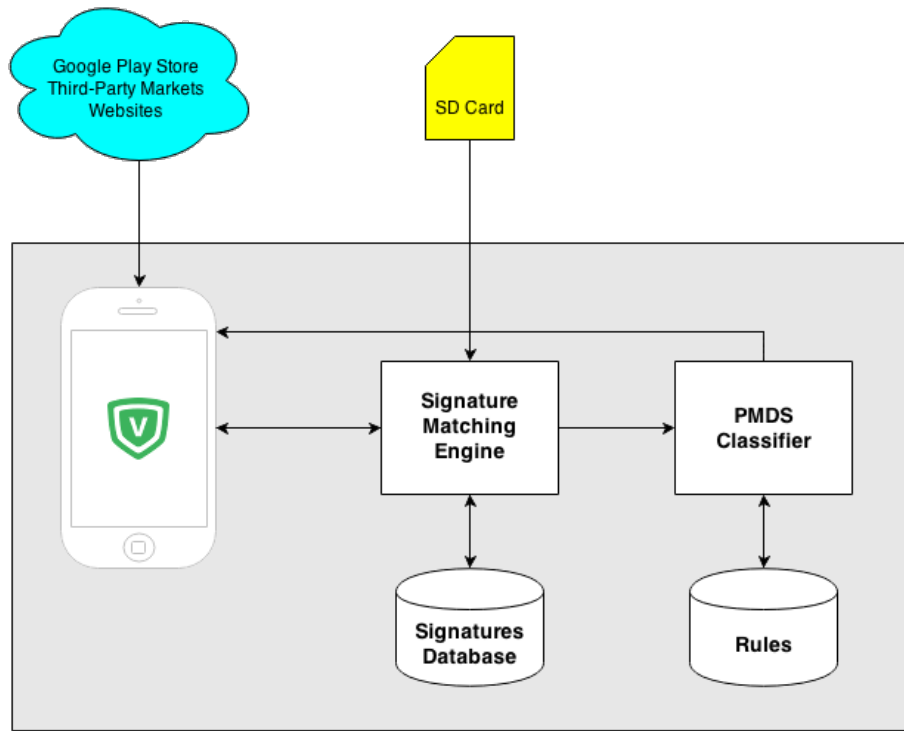
The aim of our work, indeed, is the one of exploring a possible novel technique that may work alongside traditional malware detection systems (e.g. signature-based detection), rather than creating a stand-alone detection algorithm.

<sup>1</sup> It is important to note that the declaration of certain permissions in the *AndroidManifest.xml* file does not necessarily imply their use at runtime.

### 5.3.2 Custom Permissions

Android applications (and libraries) can enforce their own, custom permissions ([The Android Open Source Project, 2013f, 2013t](#)). These custom permissions are also declared in the *AndroidManifest.xml* file together with the system ones.

Since the aim of our study is the one of understanding whether there is a correlation between the group of (system) permissions required by an application with its behaviour (i.e. benign or malicious), the custom permissions will result as noise for our classifier. Therefore, we decided to analyse only the system permissions available in the API documentation ([The Android Open Source Project, 2013c](#)), for a total of 130 permissions.



**Figure 5.3:** The architecture of the antivirus in VirIT Mobile Security. The applications are firstly scanned by the signature-based detection system and afterwards, if no signature matches them, they are scanned by the Permission-based Malware Detection System (PMDS) rule-based classifier in order to detect possible zero-day or next-generation malware. For the overall VirIT Mobile Security architecture see Figure 6.1

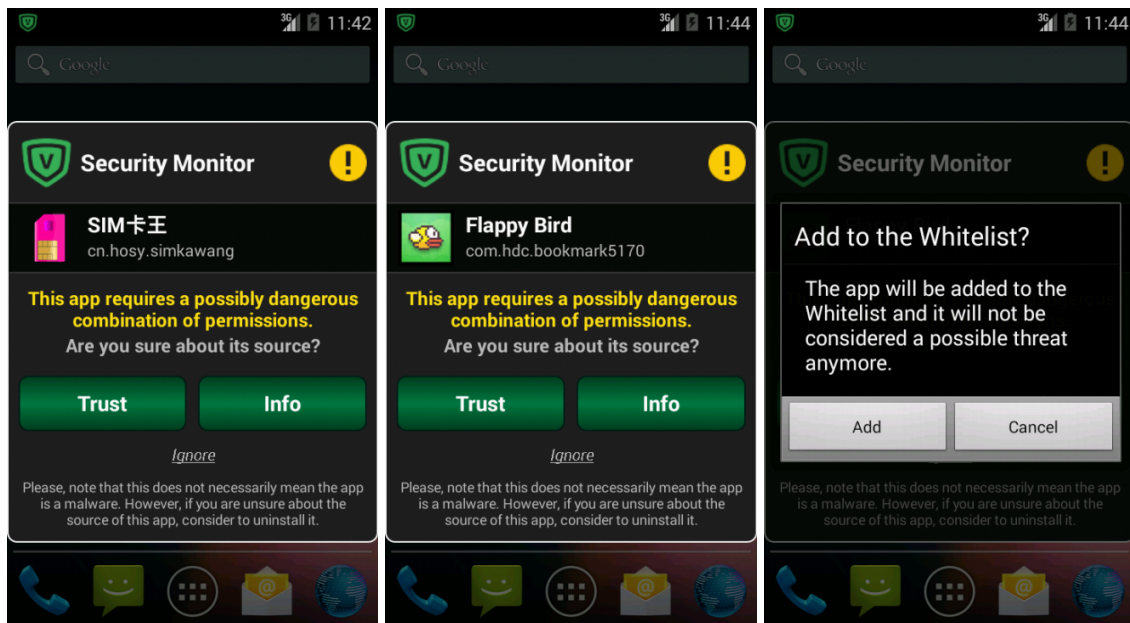
### 5.3.3 PMDS Classifier

The aim of our study is the one of understanding whether there is a correlation between a group of permissions required by an application with its behaviour (i.e. benign or malicious) and, if that is the case, how we can use this correlation to automatically identify (potentially) dangerous behaviours of previously unseen applications.

Such a system can be designed in many ways. Clearly, since our main goal is the one of protecting Android users, the first place in which our system should be is directly on Android devices. However, the same classifier might also be used outside an Android device. For example, a third-party market can apply it directly on its servers, in order to scan the applications when the developers upload them.

Albeit it is often the case, we cannot trust the mobile devices to be connected round-the-clock. Thus, at present, we prefer designing a local (on-device) system which works alongside our signature-based detection system. Future works may move our detection mechanism in the cloud.

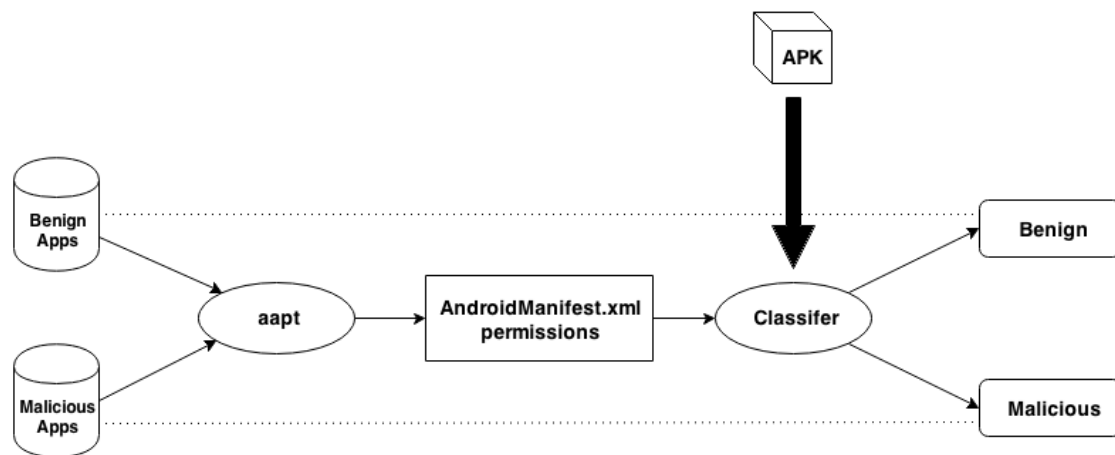
As shown in Figure 5.3, the applications are firstly scanned by the signature-based detection system and afterwards, if no signature matches them, they are scanned by the Permission-based Malware Detection System (PMDS) rule-based classifier in order to detect possible zero-day or next-generation malware.



**Figure 5.4:** The “heuristic analysis on permissions” of the on-install scanner of VirIT Mobile Security (see Figure 3.6), which makes use of the PMDS rule-based (RIPPER) classifier in order to alert if potentially dangerous applications are installed on the device. In the first two pictures (on the left), the on-install scanner detects the installation of the *Trojan-Spy://Android/Wapxx.A* and the *Trojan-SMS://Android/FakeFlappyBird.A* respectively. In the last picture (on the right) it is shown that applications can be manually whitelisted, so that they will not display as potentially dangerous in future scans.

## 5.4 Implementation

In this section we present the implementation of our Permission-based Malware Detection System (PMDS).



**Figure 5.5:** The Permission-based Malware Detection System (PMDS) architecture. The permissions declared in the *AndroidManifest.xml* file of an application are automatically extracted using the Android Asset Packaging Tool (aapt). Then, the classifier automatically labels the application behaviour, as either benign or (potentially) malicious, according to the combination of permissions the application requires.

To create the classifier’s dataset of the known benign and malicious applications, we developed a program in Python which automatically extracts the permissions declared in the *AndroidManifest.xml* file of an APK package (see Figure 5.5).

In order to extract the permissions, we decided to use the Android Asset Packaging Tool (aapt), a tool that is part of the Android SDK. Using aapt it is possible to retrieve various information about the APK package and its *AndroidManifest.xml* file (elinux.org, n.d.). In particular, thanks to the “*aapt dump permissions*” command it is possible to list the permissions declared by an application (see Listing 5.1).

```
$ aapt dump permissions MyApkPackage.apk | sed 1d | awk '{ print $NF }'
```

**Listing 5.1:** The Android Asset Packaging Tool (aapt) shell command used to retrieve the permissions declared in the *AndroidManifest.xml* file of an APK package (elinux.org, n.d.). In order to avoid custom permissions we collect only the 130 system permissions available in the Android API documentation (The Android Open Source Project, 2013c).

After extracting the permissions, the program automatically save the information in the Weka’s Attribute-Relation File Format (ARFF) (The University of Waikato, 2002, 2008). We finally use Weka (Holmes, Donkin, & Witten, 1994; Machine Learning Group at the University of Waikato, n.d.) to train multiple classifiers in order to detect new and unseen malware.

Since, as already said in Section 5.3.2, Android applications can enforce their own custom permissions, and these may result as noise to our classifier, we decided to avoid them and to collect only the system permissions available in the Android API documentation, for a total of 130 permissions. (The Android Open Source Project, 2013c)



a particular feature is unrelated to the presence (or absence) of any other feature of a class. (John & Langley, 1995)

## 5.5 Evaluation

In this section we describe the results of the experiments performed on our Permission-based Malware Detection System (PMDS).

### 5.5.1 Environment

To test the efficiency of our PMDS, we use a dataset of 2950 samples, divided into 1500 unique benign samples (i.e. no updated versions of the same applications are included) and 1450 malicious ones respectively. All the benign samples were taken from Google Play (Google, 2013), while all the malicious samples were taken from both the Android Malware Genome Project (Y. Zhou & Jiang, 2012) and Contagio Mobile (Mila, 2013). To make the analysis more effective, we collect only those applications that actually requires permissions.

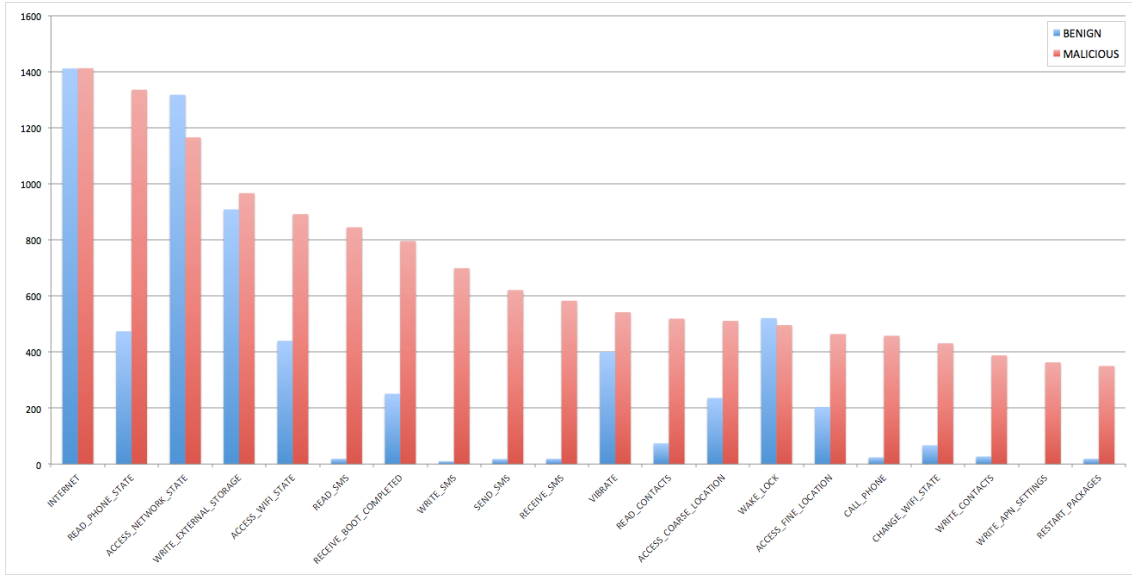
In Figure 5.6 are shown the most frequently requested permissions by the samples (both benign and malicious) in our dataset. The top required permission - i.e. *INTERNET* - is the same for both benign and malicious applications. On the other hand, as one can see, there is a vast difference in the total number of applications using certain permissions, such as: *READ\_PHONE\_STATE*, *ACCESS\_WIFI\_STATE*, *READ\_SMS*, *WRITE\_SMS*, *SEND\_SMS*, *RECEIVE\_SMS*, *READ\_CONTACTS* and *CALL\_PHONE*.

In order to evaluate the accuracy of each classifier, we use the standard tenfold cross-validation. Cross-validation is a model validation method that divides data into two segments, one used to train the machine learning algorithm and one used to test it. In particular, the tenfold cross-validation takes 90% of the dataset for training and 10% for testing, repeating the test 10 times (with different parts used for training and testing) and averaging the accuracy over the runs.

In order to evaluate the results of the performed experiments, the following standard evaluation measures are used:

- **True Positives** ( $TP$ ): the number of applications correctly classified as malicious;
- **True Negatives** ( $TN$ ): the number of applications correctly classified as benign;
- **False Positives** ( $FP$ ): the number of applications mistakenly classified as malicious;





**Figure 5.6:** Count of the most frequently requested permissions by the samples (both benign and malicious) in our dataset of 2950 samples - 1500 benign and 1450 malicious. The blue lines show the number of times the specific permissions have been requested by benign applications, while the red lines show the number of times they have been requested by malicious applications.

- **False Negatives ( $FN$ ):** the number of applications mistakenly classified as benign;
- **True Positives Rate ( $TPR = \frac{TP}{TP+FN}$ ):** the fraction of  $TP$  out of the sum of  $TP$  and  $FN$ ;
- **False Positives Rate ( $FPR = \frac{FP}{FP+TN}$ ):** the fraction of  $FP$  out of the sum of  $FP$  and  $TN$ ;
- **Accuracy ( $ACC = \frac{TP+TN}{TP+TN+FP+FN}$ ):** the fraction of applications correctly classified (that is  $TP + TN$ ) out of the total amount of applications;
- **Error Rate ( $ER = \frac{FP+FN}{TP+TN+FP+FN}$ ):** the fraction of applications mistakenly classified (that is  $FP + FN$ ) out of the total amount of applications;
- **Receiver Operating Characteristic (ROC) curve:** a graphical plot of the  $TPR$  versus the fraction of false positives out of the total actual negatives (i.e.  $\frac{FP}{TN+FP}$ ), at various threshold settings.

### 5.5.2 Classification with standard machine learning algorithms

In Table 5.1, and in Figures 5.7 and 5.8, are shown the results obtained in our first campaign of experiments, where we used the four machine learning algorithms presented before (i.e. J48, K\*, RIPPER and Näive Bayes).

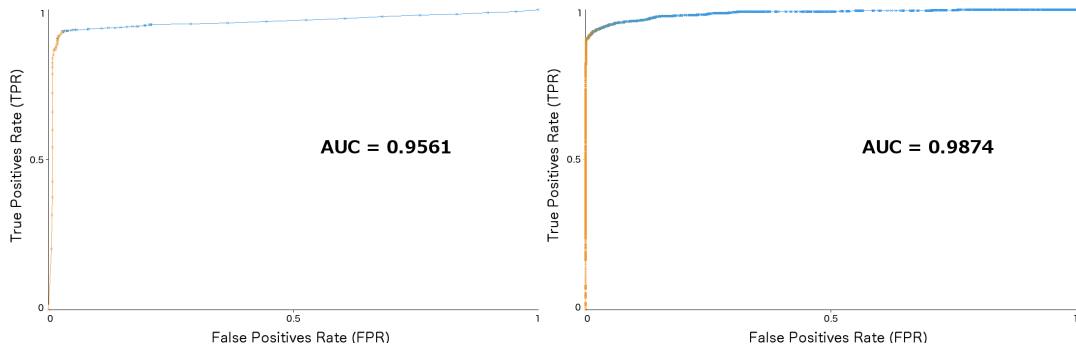
As one can see, in this first campaign, the overall best results were obtained using K\*, with which we achieved a detection rate of 92.28% and a false positives rate of 1.52% (the



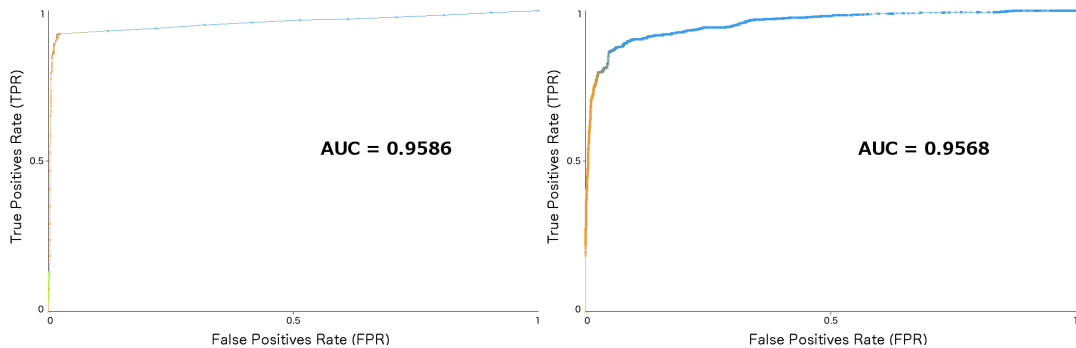
	TP	TN	FP	FN	TPR	FPR	ACC	ER
<b>J48</b>	1340	1456	44	110	92.41 %	3.03 %	94.78 %	5.22 %
<b>K*</b>	1338	1478	22	112	92.28 %	1.52 %	95.46 %	4.54 %
<b>RIPPER</b>	1338	1465	35	112	92.28 %	2.4 %	95.02 %	4.98 %
<b>Näive Bayes</b>	1155	1450	50	295	79.66 %	3.45 %	88.31 %	11.69 %

**Table 5.1:** Permission-based Malware Detection System (PMDS): Experimental results using four different classifiers - i.e. a *Decision Tree-based learner* (J48), a *Lazy (Instance-based) learner* (K\*), a *Rule-based learner* (RIPPER) and a *Bayesian learner* (Näive Bayes) - in order to automatically label the behaviour of previously unseen applications (as either benign or malicious). The experiments are performed using the standard tenfold cross-validation, which takes 90% of the dataset for training and 10% for testing, repeating the test 10 times.

lowest achieved in all the experiments). We achieved the highest detection rate (92.41%) using J48 (see its pruned tree in Listing 5.3), while the highest accuracy (95.02%) using RIPPER (see its rules set in Listing 5.4). The worst result, in term of both detection rate and false positives rate, were obtained using Näive Bayes.



**Figure 5.7:** The Receiver Operating Characteristic (ROC) Curve of our J48 (left) and K\* (right) classifiers respectively. The Area Under the Curve (AUC) represents the probability that a classifier will rank a randomly chosen *positive* instance higher than a randomly chosen *negative* one. The colour of the curve depicts the value of the threshold (i.e. closer to blue corresponds to the lower threshold value). Indeed, each point in the curve illustrates a prediction tradeoff that can be obtained by varying the threshold value between classes. Or, in other words, every point corresponds to setting a threshold on the probability assigned to the *positive* class. The typical threshold value of 0.5 means the predicted probability of *positive* must be higher than 0.5 for the instance to be predicted as *positive*.



**Figure 5.8:** The Receiver Operating Characteristic (ROC) Curve of our RIPPER (left) and Näive Bayes (right) classifiers respectively. The Area Under the Curve (AUC) represents the probability that a classifier will rank a randomly chosen *positive* instance higher than a randomly chosen *negative* one. The colour of the curve depicts the value of the threshold (i.e. closer to blue corresponds to the lower threshold value). Indeed, each point in the curve illustrates a prediction tradeoff that can be obtained by varying the threshold value between classes. Or, in other words, every point corresponds to setting a threshold on the probability assigned to the *positive* class. The typical threshold value of 0.5 means the predicted probability of *positive* must be higher than 0.5 for the instance to be predicted as *positive*.

```

READ_SMS = TRUE
|  | MANAGE_ACCOUNTS = TRUE: benign (9.0)
|  | MANAGE_ACCOUNTS = FALSE: malicious (848.0/3.0)
READ_SMS = FALSE
|  | SEND_SMS = TRUE
|  | | GET_ACCOUNTS = TRUE: benign (7.0/1.0)
|  | | GET_ACCOUNTS = FALSE: malicious (120.0)
|  | SEND_SMS = FALSE
|  | | CHANGE_WIFI_STATE = TRUE
|  | | | READ_PHONE_STATE = TRUE
|  | | | | BROADCAST_STICKY = TRUE: benign (8.0)
|  | | | | BROADCAST_STICKY = FALSE
|  | | | | USE_CREDENTIALS = TRUE: benign (5.0)
|  | | | | USE_CREDENTIALS = FALSE
|  | | | | BATTERY_STATS = TRUE: benign (2.0)
|  | | | | BATTERY_STATS = FALSE
|  | | | | | GET_PACKAGE_SIZE = TRUE: benign (2.0)
|  | | | | | GET_PACKAGE_SIZE = FALSE
|  | | | | | GET_ACCOUNTS = TRUE
|  | | | | | CAMERA = TRUE: malicious (2.0)
|  | | | | | CAMERA = FALSE: benign (7.0/1.0)
|  | | | | GET_ACCOUNTS = FALSE
|  | | | | WAKE_LOCK = TRUE
|  | | | | | RECEIVE_BOOT_COMPLETED = TRUE: malicious (38.0)
|  | | | | | RECEIVE_BOOT_COMPLETED = FALSE
|  | | | | | ACCESS_COARSE_LOCATION = TRUE: malicious (4.0/1.0)
|  | | | | | ACCESS_COARSE_LOCATION = FALSE
|  | | | | | MODIFY_AUDIO_SETTINGS = TRUE: malicious (2.0)
|  | | | | | MODIFY_AUDIO_SETTINGS = FALSE: benign (8.0)
|  | | | | WAKE_LOCK = FALSE: malicious (199.0/1.0)
|  | | READ_PHONE_STATE = FALSE: benign (21.0)
|  | CHANGE_WIFI_STATE = FALSE
|  | | INSTALL_PACKAGES = TRUE: malicious (15.0)
|  | | INSTALL_PACKAGES = FALSE
|  | | | WRITE_APN_SETTINGS = TRUE: malicious (11.0)
|  | | | WRITE_APN_SETTINGS = FALSE
|  | | | | DELETE_CACHE_FILES = TRUE: malicious (4.0)
|  | | | | DELETE_CACHE_FILES = FALSE
|  | | | | RECEIVE_SMS = TRUE
|  | | | | | GET_ACCOUNTS = TRUE: benign (2.0)
|  | | | | | GET_ACCOUNTS = FALSE: malicious (17.0)
|  | | | | RECEIVE_SMS = FALSE
|  | | | | ACCESS_LOCATION_EXTRA_COMMANDS = TRUE
|  | | | | | READ_PHONE_STATE = TRUE: malicious (19.0/2.0)
|  | | | | | READ_PHONE_STATE = FALSE: benign (9.0/1.0)
|  | | | | ACCESS_LOCATION_EXTRA_COMMANDS = FALSE
|  | | | | | READ_CONTACTS = TRUE
|  | | | | | MANAGE_ACCOUNTS = TRUE: benign (13.0)
|  | | | | | MANAGE_ACCOUNTS = FALSE
|  | | | | | ACCESS_FINE_LOCATION = TRUE: benign (8.0/1.0)
|  | | | | | ACCESS_FINE_LOCATION = FALSE
|  | | | | | WAKE_LOCK = TRUE
|  | | | | | | DISABLE_KEYGUARD = TRUE: malicious (2.0)
|  | | | | | | DISABLE_KEYGUARD = FALSE: benign (6.0)
|  | | | | | WAKE_LOCK = FALSE: malicious (35.0/6.0)
|  | | | | READ_CONTACTS = FALSE
|  | | | | ACCESS_NETWORK_STATE = TRUE
|  | | | | | SYSTEM_ALERT_WINDOW = TRUE
|  | | | | | WRITE_EXTERNAL_STORAGE = TRUE: benign (15.0/2.0)
|  | | | | | WRITE_EXTERNAL_STORAGE = FALSE
|  | | | | | ACCESS_WIFI_STATE = TRUE: malicious (5.0)
|  | | | | | ACCESS_WIFI_STATE = FALSE: benign (2.0)
|  | | | | | SYSTEM_ALERT_WINDOW = FALSE: benign (1228.0/60.0)
|  | | | | ACCESS_NETWORK_STATE = FALSE
|  | | | | | READ_PHONE_STATE = TRUE
|  | | | | | WAKE_LOCK = TRUE
|  | | | | | | DISABLE_KEYGUARD = TRUE: malicious (2.0)
|  | | | | | | DISABLE_KEYGUARD = FALSE: benign (9.0/2.0)
|  | | | | | WAKE_LOCK = FALSE: malicious (47.0/4.0)
|  | | | | READ_PHONE_STATE = FALSE
|  | | | | | BLUETOOTH = TRUE: malicious (6.0/1.0)
|  | | | | | BLUETOOTH = FALSE
|  | | | | | ACCESS_FINE_LOCATION = TRUE
|  | | | | | WAKE_LOCK = TRUE: malicious (5.0)
|  | | | | | WAKE_LOCK = FALSE: benign (8.0/1.0)
|  | | | | ACCESS_FINE_LOCATION = FALSE: benign (200.0/18.0)

```

**Listing 5.3:** The pruned tree generated by our J48 classifier in order to decide whether an application has a (potentially) malicious behaviour or not. J48 is an open source Java implementation of the Ross Quinlan's C4.5 *Decision Tree-based learning algorithm*. The C4.5 algorithm is able to build decision trees from a set of training data using the concept of information entropy. The decision trees are then used as predictive models which map observations about an item - in our case the permission requested - to conclusions about the item's target value - in our case the application's behaviour (i.e. benign or malicious). (Witten & Frank, 2005; Quinlan, 1993; Holmes et al., 1994)

```
(READ_SMS = TRUE) and (RECORD_AUDIO = FALSE) => behaviour=malicious (831.0/2.0)

(READ_PHONE_STATE = TRUE) and (CHANGE_WIFI_STATE = TRUE) and (WAKE_LOCK = FALSE) => behaviour=malicious (208.0/2.0)

(READ_PHONE_STATE = TRUE) and (ACCESS_NETWORK_STATE = FALSE) and (WAKE_LOCK = FALSE) => behaviour=malicious (109.0/4.0)

(READ_PHONE_STATE = TRUE) and (ACCESS_LOCATION_EXTRA_COMMANDS = TRUE) => behaviour=malicious (51.0/4.0)

(RECEIVE_SMS = TRUE) and (GET_ACCOUNTS = FALSE) => behaviour=malicious (73.0/0.0)

(READ_PHONE_STATE = TRUE) and (GET_ACCOUNTS = FALSE) and (READ_CONTACTS = TRUE) => behaviour=malicious (31.0/5.0)

(READ_PHONE_STATE = TRUE) and (RECEIVE_BOOT_COMPLETED = TRUE) and (READ_EXTERNAL_STORAGE = FALSE) and
(GET_ACCOUNTS = FALSE) and (ACCESS_COARSE_LOCATION = FALSE) and (CHANGE_WIFI_STATE = TRUE) and
(BLUETOOTH = FALSE) => behaviour=malicious (15.0/0.0)

(READ_PHONE_STATE = TRUE) and (WAKE_LOCK = FALSE) and (RECEIVE_BOOT_COMPLETED = TRUE) and (WRITE_SETTINGS = TRUE)
=> behaviour=malicious (11.0/2.0)

(READ_PHONE_STATE = TRUE) and (WRITE_APN_SETTINGS = TRUE) => behaviour=malicious (9.0/0.0)

(ACCESS_NETWORK_STATE = FALSE) and (SEND_SMS = TRUE) => behaviour=malicious (7.0/0.0)

(ACCESS_NETWORK_STATE = FALSE) and (WAKE_LOCK = TRUE) and (CAMERA = FALSE) and (ACCESS_FINE_LOCATION = TRUE)
=> behaviour=malicious (6.0/0.0)

(INSTALL_PACKAGES = TRUE) and (WAKE_LOCK = FALSE) => behaviour=malicious (14.0/0.0)

(READ_PHONE_STATE = TRUE) and (WRITE_EXTERNAL_STORAGE = FALSE) and (CAMERA = FALSE) and (GET_TASKS = TRUE)
=> behaviour=malicious (5.0/0.0)

(ACCESS_NETWORK_STATE = FALSE) and (BLUETOOTH = TRUE) => behaviour=malicious (6.0/1.0)

=> behaviour=benign (1574.0/94.0)
```

**Listing 5.4:** The rules generated by our RIPPER classifier in order to decide whether an application has a (potentially) malicious behaviour or not. RIPPER is a *Rule-based learning algorithm* developed by William W. Cohen. RIPPER algorithm classifies an instance according to a sequence of boolean clauses linked by logical AND operators, which together imply the membership of the instance to a particular class - in our case the application's behaviour (i.e. benign or malicious). (W. W. Cohen, 1995)

### 5.5.3 Classification with boosted machine learning algorithms

We decide to try to improve our classifiers further by using Adaptive Boosting (AdaBoost), a *machine learning meta-algorithm* developed by Yoav Freund and Robert Schapire that can be used in conjunction with many other learning algorithms to improve their performance. Substantially, AdaBoost uses a boosting approach in which multiple different classifiers are trained and, then, their output is combined into a weighted sum in order to have an accurate prediction. (Freund & Schapire, 1995)

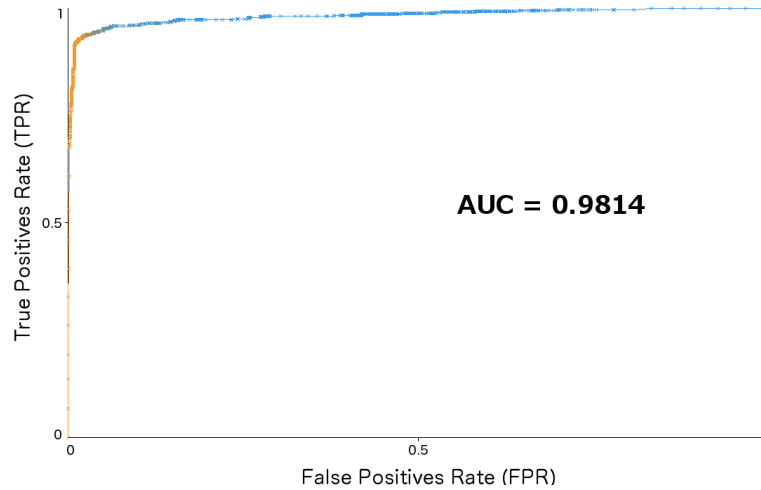
In Table 5.2, and in Figures 5.9 and 5.10, are shown the results obtained in our second campaign of experiments, where we use AdaBoost in conjunction with the previous ma-

	TP	TN	FP	FN	TPR	FPR	ACC	ER
<b>J48</b>	1366	1443	57	84	94.21 %	3.93 %	95.22 %	4.78 %
<b>RIPPER</b>	1353	1442	58	97	93.31 %	4 %	94.75 %	5.25 %
<b>Näive Bayes</b>	1345	1362	138	105	92.76 %	9.52 %	91.76 %	8.24 %

**Table 5.2:** Permission-based Malware Detection System (PMDS): Experimental results using AdaBoost in conjunction with the previous machine learning algorithms - i.e. J48, RIPPER and Näive Bayes - in order to automatically label the behaviour of previously unseen applications (as either benign or malicious). The experiments are performed using the standard tenfold cross-validation, which takes 90% of the dataset for training and 10% for testing, repeating the test 10 times.

chine learning algorithms.

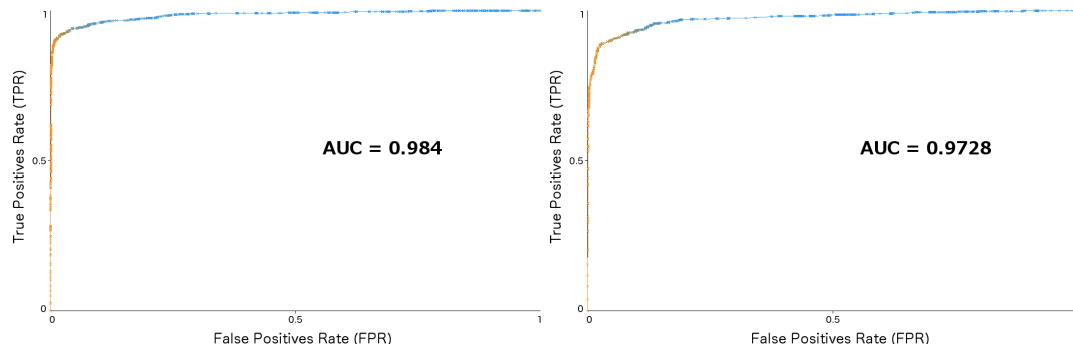
As one can see, in this second campaign, the overall best results were obtained using J48 as base classifier of AdaBoost, with which we achieved a detection rate of 94.21% and a false positives rate of 3.93%. Even if the results using Näive Bayes as base classifier for AdaBoost were the worst one, in term of both detection rate and false positives rate, it was the algorithm that showed the highest improvement thanks to the use of AdaBoost.



**Figure 5.9:** The Receiver Operating Characteristic (ROC) Curve of our AdaBoost classifier using J48 as base classifier. The Area Under the Curve (AUC) represents the probability that a classifier will rank a randomly chosen *positive* instance higher than a randomly chosen *negative* one. The colour of the curve depicts the value of the threshold (i.e. closer to blue corresponds to the lower threshold value). Indeed, each point in the curve illustrates a prediction tradeoff that can be obtained by varying the threshold value between classes. Or, in other words, every point corresponds to setting a threshold on the probability assigned to the *positive* class. The typical threshold value of 0.5 means the predicted probability of *positive* must be higher than 0.5 for the instance to be predicted as *positive*.

## 5.6 Conclusions

In this chapter we have proposed a novel Android malware detection technique, called Permission-based Malware Detection System (PMDS). Our work aimed to build a classifier that was able to automatically identify (potentially) dangerous behaviours/intents of previously unseen applications based on the combination of permissions they require.



**Figure 5.10:** The Receiver Operating Characteristic (ROC) Curve of our AdaBoost classifier using RIPPER (left) and Naïve Bayes (right) as base classifiers respectively. The Area Under the Curve (AUC) represents the probability that a classifier will rank a randomly chosen *positive* instance higher than a randomly chosen *negative* one. The colour of the curve depicts the value of the threshold (i.e. closer to blue corresponds to the lower threshold value). Indeed, each point in the curve illustrates a prediction tradeoff that can be obtained by varying the threshold value between classes. Or, in other words, every point corresponds to setting a threshold on the probability assigned to the *positive* class. The typical threshold value of 0.5 means the predicted probability of *positive* must be higher than 0.5 for the instance to be predicted as *positive*.

Even if with a little high False Positives Rate, the experimental results shows the feasibility of using these classifiers in order to provide heuristic detection on zero-day or next-generation malware which are not detected by the signature-based detection system (see Chapter 3).

However, even if the experimental results were good, at present, our approach has a main limitation. Indeed, we do not take in account implied permissions and the fact that the permissions declared by Android applications might change based on the API they are supporting/targeting. For example, the `READ_EXTERNAL_STORAGE` permission, which allows an application to read from the External Storage, is implicitly granted by the system if the targeted API level is equal or lower than 3 or if the application requires the `WRITE_EXTERNAL_STORAGE` permission, which allows an application to write to the External Storage. Furthermore, the `READ_EXTERNAL_STORAGE` permission is enforced only from API level 19, from which this permission is not required to read files in the application-specific directories anymore.

Another example is the `READ_CALL_LOG` permission, which allows an application to read the user's call log. If an application targets API level equals or lower than 15, this particular permission is implicitly granted by the system if the application requires also the `READ_CONTACTS` permission, which allows an application to read the user's contacts data.

Thus, it is possible that an application is able to use some permissions without declaring them in its *AndroidManifest.xml* file. Therefore, in future works, we should take into account these implied permissions in order to have a better correlation between the group of permissions required by applications (i.e. their actions) and their behaviour.

Furthermore, as already said, a main concern is that the declaration of certain permissions in the *AndroidManifest.xml* file does not necessarily mean that these are actually used at

runtime. Theoretically, since an application should request only the permissions it needs, this should not be a matter. However, according to several works, a large percentage (about one-third) of Android applications are over-privileged (Felt, Chin, Hanna, Song, & Wagner, 2011; Felt, Greenwood, & Wagner, 2011; Wei, Gomez, Neamtiu, & Faloutsos, 2012).

The final area we considered is the kinds of applications that are being considered. Indeed, at present, we are able to correlate a group of permissions required by an application with its behaviour, meant as either benign or (potentially) malicious. In future works, we might divide the dataset of benign applications based on their type (e.g. browser, mail client, etc...) and the malware one as well (e.g. Adware, Trojan, etc...). This should theoretically give us a better overview and, accordingly, a better detection.

## Chapter 6

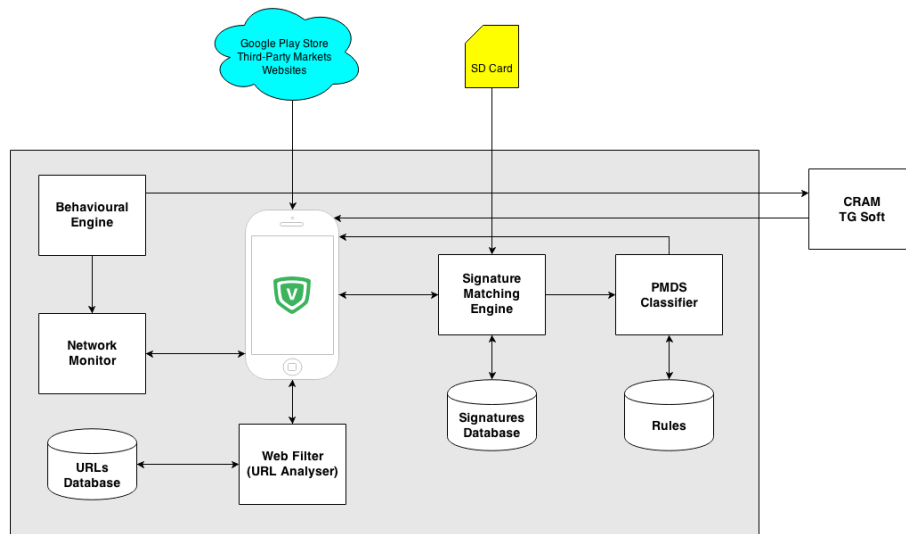
# Implementing Network Protection and a Behavioural Detection System in Android

In this chapter we present the Android network protection problem and propose a Web Filter, which alerts the user about malicious and phishing websites (see Section 6.2), and a Network Monitor, which monitors the network usage of each application (see Section 6.3). Finally, in Section 6.4, we present a simple but effective behavioural detection system, known as Community Network, that uses both the information provided by the PMDS rule-based (RIPPER) classifier (see Section 5) and the Network Monitor in order to collect and signal (potentially) dangerous applications to the Anti-Malware Research Center of TG Soft (CRAM). This allows TG Soft to spot zero-day or next-generation malware - i.e. malicious applications which are not detected by the antivirus - as they emerge.

Experimental results show that our Community Network was able to spot 75% of the zero-day or next-generation malware in our test.

### 6.1 Introduction

Most of the mobile devices are nowadays connected to the Internet round-the-clock. As already stated, in 2012 Lookout has reported that the global yearly likelihood of an Android user clicking on an unsafe link is 36% (Lookout, 2012).



**Figure 6.1:** The overall architecture of VirIT Mobile Security. Suspicious applications detected by the behavioural detection system are forwarded to the Anti-Malware Research Center of TG Soft (CRAM). If it is the case of a zero-day or next generation malware - i.e. a malicious application which is not detected by the antivirus - a proper signature is extracted and, in turn, an update of the signatures database is released.

Thus, in an effective mobile security solution it is crucial to monitor application's incoming and outgoing network traffic, preventing it if unauthorized, and to protect the user against the risk that one is exposed to while browsing the web (i.e. malicious, infected and phishing websites).

Unfortunately, at present, in Android there is no way to build a network filtering system without having the root privileges. Furthermore, by design, in Android it is not possible to interact with the network interface in order to read a web page content or its URL before it is actually loaded in the browser.

However, even with this and other main limitations, some possible workarounds can be used in order to at least warning an user in case a malicious or phishing website is opened (see Section 6.2) and to monitor the network usage of each application (see Section 6.3).

## 6.2 Web Filter

As already said, by design, in Android it is not possible to interact with the network interface in order to read a web page content or its URL before it is actually loaded in the browser. However, it is possible to use the *ContentObserver* class in order to retrieve the visited URLs from the default Android browser's history ([The Android Open Source Project, 2013g](#)) (See Listing 6.1).

It is important to realise that we can only check the URLs the user visits, not the actual



content of the web pages, and only the ones that are visited with the default Android browser or with Google Chrome. Another main limitation to this approach is that it is not sure that we will be able to scan an URL before its web page content will actually be loaded. Moreover, even if an URL is detected as malicious before the actual content has been finished to be loaded, with the design of the Android application model, we will not be able to block it or to close the web page. Thus, such a Web Filter is totally useless to protect the user from some web-based attacks (e.g. malicious JavaScript). However, even with this main limitation, our Web Filter can protect the user from other web threats, such as phishing websites.

```
private static class BrowserObserver extends ContentObserver {
    //Query values:
    final String[] projection = new String[] { Browser.BookmarkColumns.URL };
    final String selection = Browser.BookmarkColumns.BOOKMARK + "="_0";
    final String sortOrder = Browser.BookmarkColumns.DATE + "_DESC";

    /**
     * Class Constructor.
     *
     * @param handler the handler to run onChange(boolean) on, or null if none.
     */
    public BrowserObserver(Handler handler) {
        super(handler);
    }

    /**
     * A content change occurs.
     *
     * @param selfChange true if this is a self-change notification.
     */
    @Override
    public void onChange(boolean selfChange) {
        onChange(selfChange, null);
    }

    /**
     * A content change occurs.
     *
     * @param selfChange true if this is a self-change notification.
     * @param uri the Uri of the changed content, or null if unknown.
     */
    @Override
    public void onChange(boolean selfChange, Uri uri) {
        super.onChange(selfChange);

        if ( uri != null ) {
            //Retrieve all the visited URLs:
            final Cursor cursor =
                getContentResolver().query(Browser.BOOKMARKS_URI, projection, selection, null, sortOrder);

            //Retrieve the last-visited URL:
            cursor.moveToFirst();
            if ( !cursor.isAfterLast() ) {
                final String url = cursor.getString(cursor.getColumnIndex(projection[0]));

                //Close the cursor:
                cursor.close();

                [...] Scan the URL [...]
            }
            else {
                //Close the cursor:
                cursor.close();
            }
        }
    }
}
```

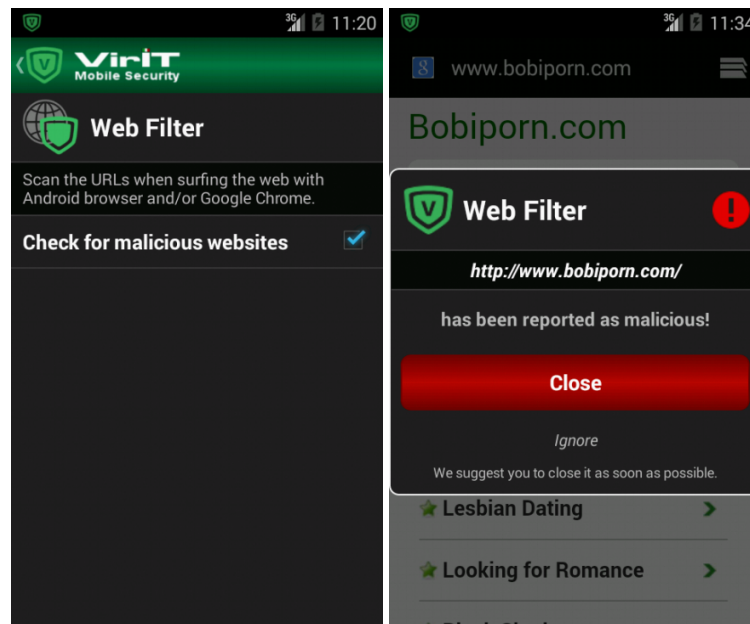
```

    }
    }
}

```

**Listing 6.1:** Example of *ContentObserver* used to retrieve the visited URLs from the default Android browser's history. Thanks to it we can develop a Web Filter which alerts the user in case a malicious or phishing website is opened.

At present, our Web Filter (see Figure 6.2) is designed to be integrated with both the default Android browser and Google Chrome.



**Figure 6.2:** The Web Filter of VirIT Mobile Security, which scans the URLs when surfing the web with the default Android browser and Google Chrome. Thanks to the Web Filter we can alert the user when a malicious or phishing website is opened.

## 6.3 Network Monitor

As already said, in Android it is not possible to monitor the applications' network traffic. However, it is still possible to retrieve the network usage of each application (see Figure 6.3).

By monitoring the network usage of the installed applications we can have an overview of the incoming and outgoing network traffic for each application and, possibly, decide if it is the case of a zero-day or next-generation malware - i.e. a malicious application which is not detected by the antivirus.

Starting from API 8 it is possible to use the Android *TrafficStats* class ([The Android Open Source Project, 2013u](#)) in order to retrieve the bytes received and/or transmitted (with

both TCP and UDP) by a given kernel User-ID (UID)<sup>1</sup> since device boot (see Listing 6.2).

```

/*
 * Given:
 * - uid: the kernel User-ID.
 */

float totalDownloadTraffic = (float) TrafficStats.getUidRxBytes(uid);
float totalUploadTraffic = (float) TrafficStats.getUidTxBytes(uid);

if ( (totalDownloadTraffic == TrafficStats.UNSUPPORTED) && (totalUploadTraffic == TrafficStats.UNSUPPORTED) ) {
    Log.e(TAG, "The_device_does_not_support_TrafficStats_monitoring!");
}

```

**Listing 6.2:** Example of monitoring the network usage with *TrafficStats*. Thanks to it we can develop a Network Monitor which, by monitoring the network usage of all the installed applications, have an overview of the incoming and outgoing network traffic for each application and, possibly, decide if it is the case of a zero-day or next-generation malware.

However, as already said, *TrafficStats* returns the network usage only from the device boot. Therefore, we need to take care of calculating and storing the total amount of daily, monthly and yearly network usage. Since, in Android, there is no way to know when the device is going to be turned off, we check the values returned by the *TrafficStats* many times a day - for example every 10 or 15 minutes - in order to update the total amount of daily (incoming and outgoing) network traffic.

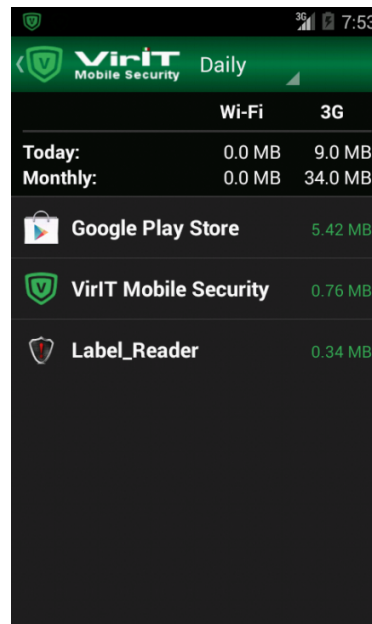
If an unknown, next-generation malware is installed on the device, and if it connects to the Internet, one might be able to spot it by looking at its network usage.

## 6.4 Community Network and Behavioural Detection

As already said in Section 3.3.1, the accuracy of a signature-based detection system is heavily based on the completeness of the signatures database. Thus, besides working at the implementation and improvement of external systems that automatically gather new malware samples, we have included into VirIT Mobile Security (see Figure 6.1) a behavioural detection system, called *Community Network*, which uses behavioural analysis based on the applications' life cycle.

At present, the aim of our behavioural detection system is not the one of being a reliable malware detection system, but rather the one of collecting and signalling (potentially) dangerous applications to the Anti-Malware Research Center of TG Soft (CRAM), in order to spot zero-day or next-generation malware. In these way, the CRAM researchers and/or dynamic analysis systems can study further the suspicious applications in order to decide whether they actually are malware or not. If it is the case of zero-day or next-generation

<sup>1</sup> It is important to note that, currently, the UID is not a unique identifier. Multiple applications can have the same UID (usually the one of the same developer). ([The Android Open Source Project, 2013v](#))



**Figure 6.3:** The Network Monitor of ViriT Mobile Security, which monitors the network usage of each application. Thanks to the Network Monitor we can have an overview of the incoming and outgoing network traffic for each application and, possibly, decide if it is the case of a zero-day or next-generation malware - i.e. a malicious application which is not detected by the antivirus. In the picture, we can see the daily network usage of both the *Trojan://Android/FakeMarket.A*, which pretends to be the Google Play store application but runs silently in the background to perform click fraud (Rovelli, 2014; AndroTotal, 2014), and the *Rogue-AV://Android/AndroidDefeder.A*, which pretends to be an antivirus solution and claims to detect some threats on the victims' devices in order to lure the victims to pay to remove the non-existent threats (Rovelli, 2013c; Ducklin, 2013).

malware, a proper signature is extracted and, in turn, an update of the signatures database is released. This allows TG Soft to dramatically shorten the life of those threats.

Since our goal is the one of detecting zero-day or next-generation malware, our behavioural detection system is applied only to those applications which have not already been detected and removed by the signature-based detection system (see Chapter 3).

However, at present, our behavioural detection system is pretty simple and relies only on the PMDS rule-based (RIPPER) classifier (see Section 5) and on few others hard-coded rules regarding the network usage (see Section 6.3).

## 6.5 Evaluation

As already said, in order to see the real effectiveness of an antivirus software, we want to test its robustness as well as its resilience (Hawes, 2013).

In this section we present the results of the resilience experiments, that is the ability of our solution to recover from zero-day or next-generation malware which do manage to get installed into a device.

Generally speaking, testing the resilience is much more complicated than testing the ro-

Mobile Security solution	TP	Detection Rate
AV-1	7	87.5%
AV-2	4	50%
AV-3	5	62.5%
AV-4	5	62.5%
VirIT	6	75%

**Table 6.1:** Results of the resilience experiments on VirIT Mobile Security. With the term resilience we mean the ability of our solution to recover from zero-day or next-generation malware which do manage to get installed into a device. In the table, for all the antivirus solutions tested, we point out the True Positives (TP) (i.e. the number of malware detected out of a total of 8 malware on the device) and the corresponding detection rate.

bustness. Ideally, a test can be seeing how long all those threats that have not been initially detected by the antivirus will last. Indeed, thanks to the behavioural detection system or to other external systems which gather new malware samples (see Section 6.4), we should be able to collect previously unknown malware samples and release proper updates of the signatures database. Once VirIT Mobile Security will be updated, the *on-update scanner* should be able to properly detect and remove the malware from the device (see Section 4.2.2).

Thus, in order to evaluate the resilience of VirIT Mobile Security, we decide to re-run the same experiments we run Section 3.5.1 in order to evaluate its robustness.

Since our aim is the one of stating the resilience of our mobile security, we re-run them with the same exactly configuration and signatures database.

### 6.5.1 Results

As shown in Section 3.5.2, the signature-based detection system of VirIT Mobile Security was able to properly detect and remove 142 malicious applications out of 148, that is a reactive detection rate of 95.95%.

In addition, this time, the behavioural detection system of VirIT Mobile Security was able to report 10 applications in less than a day. Analysing these applications, we discover that 4 of them were actually part of the group of malware we did not catch. Of the remaining 6 applications, we have discovered that 2 of them were actually zero-day or next-generation malware that were not detected by any of the other antivirus solutions used, while the other 4 were benign applications.

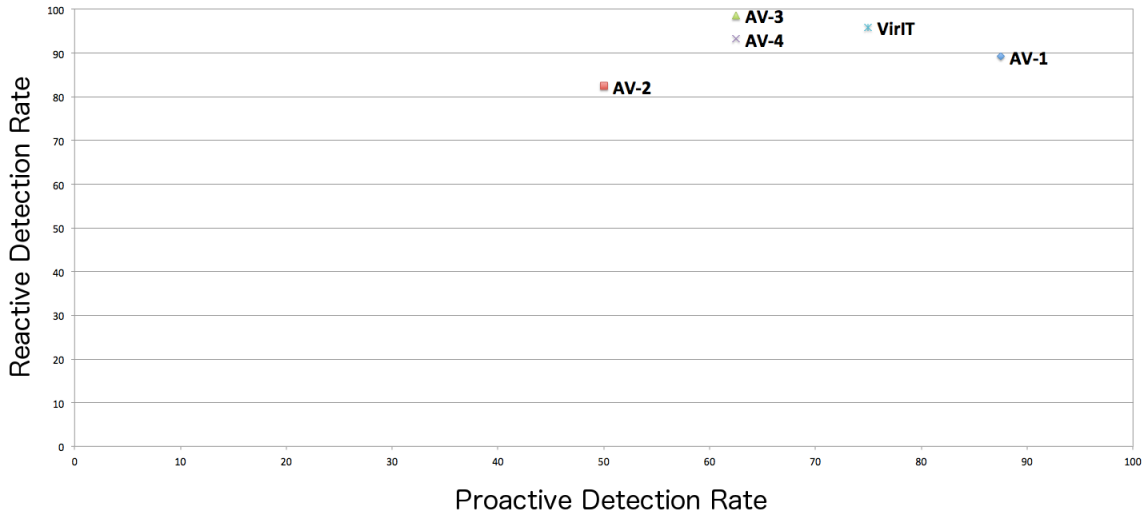
According to these numbers, VirIT Mobile Security has shown a proactive detection rate of 75%, being able to catch 6 malware out of the remaining 8.

In order to compare our result with the one of the others, we decide to re-scan our dataset with the previously used antivirus solutions. The results are shown in Table 6.1.

Finally, in Figure 6.4 it is possible to see the overall reactive and proactive (RAP) results.

It is important to note that this experiment was a little bit unfair. Indeed, we run our

## RAP (Reactive And Proactive)



**Figure 6.4:** Testing the resilience of VirIT Mobile Security. With the term reactive detection rate we mean the number of malware properly detected at the first scan (i.e. the robustness of the antivirus), while with the term proactive detection rate we mean the number of malware detected in a second scan (i.e. the resilience of the antivirus).

mobile security solution directly on an Android emulator and, by doing so, we were able to take advantages of our behavioural detection system. On the other hand, however, we gave to our detection systems a time frame of 1 day on a single device (emulator), while the other antivirus solutions had several weeks and runs simultaneously on thousands or millions of devices.

As already stated, since the dataset used was rather too small, this experiment is not to be considered complete in order to state the robustness of our mobile security solution. However, it can give a rough idea.

## 6.6 Conclusion

In this chapter we have present the network protection problem and proposed possible workarounds. Indeed, at present, implementing network protection mechanisms in Android is infeasible, at least without having the root privileges. However, we were able to design and implement a Web Filter, which is able to protect users from phishing websites. Furthermore, we have implemented and evaluated a behavioural detection system, known as Community Network, that helps us to spot zero-day or next-generation malware as they emerge.

Although with a too high number of false alerts (40%), our behavioural detection system

has proven effective, being able to discover 6 zero-day or next-generation malware.





## Chapter 7

# Conclusions and Future Work

In this work of thesis we have presented VirIT Mobile Security, a mobile security solution specifically designed and developed to counter Android malware.

To this end, we have firstly analysed the “in the wild” Android malware, characterising the current known threats and discussing possible feature ones.

Afterwards, we have implemented different malware detection models, in order to provide both reactive and proactive protection to Android end-users. In particular we have designed and implemented: a signature-based detection system, with four different scanners (i.e. *on-demand*, *on-install*, *on-access* and *on-update*), a novel heuristic Permission-based Malware Detection System (PMDS), and, finally, a behavioural detection system.

In our experiments, all those malware detection systems have shown encouraging results. Indeed, our signature-based detection system was able to properly detect and remove 95.95% of the malware in our test, while our PMDS was able to detect more than 94% of previously unseen malware. Finally, our behavioural detection system was able to spot and report back to the Anti-Malware Research Center of TG Soft (CRAM) 75% of the zero-day or next-generation malware.

In order to implement VirIT Mobile Security, we have dealt with several challenges. Indeed, at present, although the Android architecture and application model have shown to create a very secure environment, they also show some main limitations.

In our work of thesis we have often encountered such limitations, such as in the implementation of a real-time monitor (see Section 4.1) or in the implementation of network protection mechanisms (see Section 6.1).

It is clear that, albeit these limitations we faced are also limitations for cyber-criminals, with this security model it is impossible to properly protect the end-users from some threats (e.g. exploits targeting specific kernel vulnerabilities and web attacks). Indeed, apart for Web Filter (see Section 6.2) which is able to protect users from phishing attacks,

our work of thesis has been mostly limited to malware detection and prevention. However, there is much more to do in order to protect mobile users. Indeed, in the mobile space there are other threats, such as privacy leaks (Enck et al., 2010; Gibler, Crussell, Erickson, & Chen, 2012) and Wi-Fi vulnerabilities (Rovelli, 2013d).

In our opinion, among many others, there are at least two main functionalities that Google should add to the Android API in order to improve the effectiveness of mobile security solutions:

- **Extending the *inotify* files change notification system** in order to be able not only to monitor the files into a specific directory, but also to retrieve which application is accessing these files;
- **Monitoring the incoming and outgoing network traffic**, that is to be able to determine which application is opening which URL.

As far as we can tell, adding these permissions should not broke the Android security model, and it would surely provide useful features to improve the end-user protection.

Another possible workaround, similar to the “*default SMS app*” concept introduced starting from Android 4.4 (KitKat) (Main, Scott and Braun, David, 2013), might be the one of creating a “*default security app*” which will be the only application in the system able to perform certain security related actions, such as interacting with the installation process - i.e. the ability to scan applications before they are actually installed - and filtering web contents. This would end up being very useful also in order to cut down the threat of the Rogue-AVs.

Besides continually improving all the existing detection mechanisms, in future, we would like to focus more on our behavioural detection system in order to make it more reliable and effective. A possible detection approach might be the one taken by Shabtai et al. in Andromaly, a machine learning Android malware detection system which continuously monitors various features and events obtained from the system and then applies standard machine learning classifiers to classify collected observations as either benign or malicious. (Shabtai et al., 2012)

Furthermore, in VirIT Mobile Security, we opted for an on-update scanner as real-time protection mechanism. In our design the on-update scanner performs a “fast scan” - only the *dex* files are scanned (see Section 3.3.3) - on all the installed applications after every update. In future releases, if we will be able to improve it, we might think to replace it with the live scanner. However, in our opinion, the best option is given by the cloud scanner, so we might prefer implementing it rather than improving the live scanner.

Other possible future works regard static analysis of both *dex* and *AndroidManifest.xml* files, for example in order to extract URLs and ad networks, and telemetry information.

Indeed, having many customers spread all over the world, there are lots of telemetry information we can use to decide, or to help to decide, whether an application observed by a certain customer in a certain location is likely malicious or not.

Cyber-criminals will not stop trying to improve, so do we.



# Bibliography

- AB. (2013, February). *Android malware infects Windows PCs with spy bot!* G Data Software. Retrieved from <http://blog.gdatasoftware.com/blog/article/android-malware-infects-windows-pcs-with-spy-bot.html>
- AlarmManager. (2013). *ActivityManager.RunningAppProcessInfo*. Retrieved from <http://developer.android.com/reference/android/app/AlarmManager.html>
- AndroTotal. (2014, January). *Android FakeMarket Analysis*. Retrieved from <http://blog.andrototal.org/post/73944579198/android-fakemarket-analysis>
- Arnold, W., & Tesauro, G. (2000). Automatically generated win32 heuristic virus detection. In *Proceedings of the 2000 international virus bulletin conference*.
- Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., & Rieck, K. (2013). DREBIN: Efficient and Explainable Detection of Android Malware in Your Pocket.
- Ask, K. (2006). Automatic Malware Signature Generation.
- Asrar, I., & Imano, S. (2011, October). *Will Your Next TV Manual Ask You to Run a Scan Instead of Adjusting the Antenna?* Symantec. Retrieved from <http://www.symantec.com/connect/blogs/will-your-next-tv-manual-ask-you-run-scan-instead-adjusting-antenna>
- AV-TEST. (2013, February). *Protection Apps for Android*. Retrieved from [http://www.av-test.org/fileadmin/pdf/avtest\\_2013-01\\_android\\_testreport\\_english.pdf](http://www.av-test.org/fileadmin/pdf/avtest_2013-01_android_testreport_english.pdf)
- Ayer, E. (n.d.). Mobile Malware and Rootkits in Perspective.
- Barroso, D. (2010, September). *Zeus Mitmo: Man-in-the-mobile*. S21sec. Retrieved from <http://securityblog.s21sec.com/2010/09/zeus-mitmo-man-in-mobile-i.html>
- Blasing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S. A., & Albayrak, S. (2010). An android application sandbox system for suspicious software detection. In *Malicious*

- and unwanted software (malware)*, 2010 5th international conference on (pp. 55–62).
- Bontchev, V. V. (1998). *Methodology of computer anti-virus research*.
- Bose, A., Hu, X., Shin, K. G., & Park, T. (2008). Behavioral detection of malware on mobile handsets. In *Proceedings of the 6th international conference on mobile systems, applications, and services* (pp. 225–238).
- Brodbeck, R. C. (2012). *Covert Android Rootkit Detection: Evaluating Linux Kernel Level Rootkits on the Android Operating System* (Tech. Rep.). DTIC Document.
- Castillo, C. (2011, July). *Dissecting Zeus for Android (or Is It Just SMS Spyware?)*. McAfee. Retrieved from <https://blogs.mcafee.com/mcafee-labs/dissecting-zeus-for-android-or-is-it-just-an-sms-spyware>
- Chebyshev, V. (2013, February). *Mobile attacks!* Kaspersky. Retrieved from [https://www.securelist.com/en/blog/805/Mobile\\_attacks](https://www.securelist.com/en/blog/805/Mobile_attacks)
- Christodorescu, M., & Jha, S. (2006). *Static analysis of executables to detect malicious patterns* (Tech. Rep.). DTIC Document.
- Christodorescu, M., Jha, S., & Kruegel, C. (2008). Mining specifications of malicious behavior. In *Proceedings of the 1st india software engineering conference* (pp. 5–14).
- ClamAV. (2007). *Creating signatures for clamav*. Retrieved from <http://www.clamav.net/doc/latest/signatures.pdf>
- Cleary, J. G., & Trigg, L. E. (1995). K\*: An Instance-based Learner Using an Entropic Distance Measure. In *Icml* (pp. 108–114).
- CNN. (2011). *Smartphone Shipments Tripled Since '08. Dumb Phones Are Flat*. Retrieved from <http://tech.fortune.cnn.com/2011/11/01/smartphone-shipments-tripled-since-08-dumb-phones-are-flat>
- Cohen, F. (1987). Computer viruses: theory and experiments. *Computers & security*, 6(1), 22–35.
- Cohen, W. W. (1995). Fast effective rule induction. In *Icml* (Vol. 95, pp. 115–123).
- Corrons, L. (2013a, February). *Android users under attack through malicious ads in Facebook*. Panda Security. Retrieved from <http://pandalabs.pandasecurity.com/android-users-under-attack-through-malicious-ads-in-facebook/>
- Corrons, L. (2013b, February). *New malware attack through Google Play*. Panda Security. Retrieved from <http://pandalabs.pandasecurity.com/new-malware-attack-through-google-play/>

- Corrons, L., & Correll, S. (2010). The Business of Rogueware: Analysis of the New Style of Online Fraud. *Web Application Security*, 72, 7–7.
- CRAM. (2014, April). *Anche quest'anno il C.R.A.M. sarà presente a SMAU Padova con un workshop formativo*. TG Soft. Retrieved from [http://www.tgsoft.it/italy/news\\_archivio.asp?id=583](http://www.tgsoft.it/italy/news_archivio.asp?id=583)
- Crussell, J., Gibler, C., & Chen, H. (2012). Attack of the Clones: Detecting Cloned Applications on Android Markets. In *Computer security—esorics 2012* (pp. 37–54). Springer.
- Dagon, D., Martin, T., & Starner, T. (2004). Mobile Phones as Computing Devices: The Viruses are Coming! *Pervasive Computing, IEEE*, 3(4), 11–15.
- Dua, S., & Du, X. (2011). *Data mining and machine learning in cybersecurity*. Taylor & Francis.
- Ducklin, P. (2013, May). *Android malware in pictures - a blow-by-blow account of mobile scareware*. Sophos. Retrieved from <http://nakedsecurity.sophos.com/2013/05/31/android-malware-in-pictures-a-blow-by-blow-account-of-mobile-scware/>
- elinux.org. (n.d.). *Android Asset Packaging Tool*. Retrieved from [http://elinux.org/Android\\_aapt](http://elinux.org/Android_aapt)
- Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., & Sheth, A. (2010). TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Osdi* (Vol. 10, pp. 255–270).
- F-Prot. (2013). *What is an archive bomb?* Author. Retrieved from [http://www.f-prot.com/support/windows/fpwin\\_faq/90.html](http://www.f-prot.com/support/windows/fpwin_faq/90.html)
- F-Secure. (2013). *F-Secure Mobile Threat Report Q1 2013*. Author. Retrieved from [http://www.f-secure.com/static/doc/labs\\_global/Research/Mobile\\_Threat\\_Report\\_Q1\\_2013.pdf](http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q1_2013.pdf)
- Felt, A. P., Chin, E., Hanna, S., Song, D., & Wagner, D. (2011). Android permissions demystified. In *Proceedings of the 18th acm conference on computer and communications security* (pp. 627–638).
- Felt, A. P., Finifter, M., Chin, E., Hanna, S., & Wagner, D. (2011). A survey of mobile malware in the wild. In *Proceedings of the 1st acm workshop on security and privacy in smartphones and mobile devices* (pp. 3–14).
- Felt, A. P., Greenwood, K., & Wagner, D. (2011). The effectiveness of application permissions. In *Proceedings of the 2nd unix conference on web application development* (pp. 7–7).
- Firdausi, I., Lim, C., Erwin, A., & Nugroho, A. S. (2010). Analysis of machine learning techniques used in behavior-based malware detection. In *Advances in computing*,

- control and telecommunication technologies (act), 2010 second international conference on* (pp. 201–203).
- Foley, S. N., & Dumigan, R. (2001). Are handheld viruses a significant threat? *Communications of the ACM*, 44(1), 105–107.
- ForeSafe. (2013). *ForeSafe Mobile Security: Redesigned Security for Android*. Retrieved from [http://www.foresafe.com/ForeSafe\\_WhitePaper.pdf](http://www.foresafe.com/ForeSafe_WhitePaper.pdf)
- Forristal, J. (2013, July). *Uncovering Android Master Key That Makes 99% of Devices Vulnerable*. Bluebox Security. Retrieved from <http://bluebox.com/corporate-blog/bluebox-uncovers-android-master-key>
- Freund, Y., & Schapire, R. E. (1995). A decision-theoretic generalization of on-line learning and an application to boosting. In *Computational learning theory* (pp. 23–37).
- G Data Software. (2013, June). *G Data Mobile Malware Report: January-June 2013*. Author. Retrieved from [http://blog.gdatasoftware.com/uploads/media/GData\\_MobileMWR\\_H1\\_2013\\_EN\\_01.pdf](http://blog.gdatasoftware.com/uploads/media/GData_MobileMWR_H1_2013_EN_01.pdf)
- Gamble, J. (2012, February). *2013: Made-to-Measure Malware and the Battle Against Adware*. Lookout. Retrieved from <https://blog.lookout.com/blog/2014/02/20/malware-made-to-measure/>
- Gibler, C., Crussell, J., Erickson, J., & Chen, H. (2012). Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Trust and trustworthy computing* (pp. 291–307). Springer.
- Google. (2013). *Android Open Source Project*. Retrieved from <http://source.android.com>
- Google. (2013). *Google Play*. Retrieved from <https://play.google.com/store>
- Griffin, K., Schneider, S., Hu, X., & Chiueh, T.-c. (2009). Automatic generation of string signatures for malware detection. In *Recent advances in intrusion detection* (pp. 101–120).
- Hawes, J. (2013, November). *Are anti-virus testers measuring the right things?* Sophos. Retrieved from <http://nakedsecurity.sophos.com/2013/11/05/are-av-testers-measuring-the-right-things/>
- Holmes, G., Donkin, A., & Witten, I. H. (1994). Weka: A machine learning workbench. In *Intelligent information systems, 1994. proceedings of the 1994 second australian and new zealand conference on* (pp. 357–361).
- Huang, C.-Y., Tsai, Y.-T., & Hsu, C.-H. (2013). Performance Evaluation on Permission-Based Detection for Android Malware. In *Advances in intelligent systems and applications-volume 2* (pp. 111–120). Springer.



- Hypponen, M. (2007). State of cell phone malware in 2007. *USENIX*. Retrieved from [www.usenix.org/events/sec07/tech/hypponen.pdf](http://www.usenix.org/events/sec07/tech/hypponen.pdf)
- IDC. (2013, 4). *More Smartphones Were Shipped in Q1 2013 Than Feature Phones, An Industry First According to IDC*. Retrieved from <http://www.idc.com/getdoc.jsp?containerId=prUS24085413>
- Jacob, G., Debar, H., & Filiol, E. (2008). Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in computer Virology*, 4(3), 251–266.
- Jarabek, C., Barrera, D., & Aycok, J. (2012). ThinAV: truly lightweight mobile cloud-based anti-malware. In *Proceedings of the 28th annual computer security applications conference* (pp. 209–218).
- Jiang, X. (2012a, February). *Clickjacking Rootkits for Android: the Next Big Threat?* Retrieved from <http://web.ncsu.edu/abstract/technology/wms-jiang-clickjack/>
- Jiang, X. (2012b, March). *Security Alert: New RootSmart Android Malware Utilizes the GingerBreak Root Exploit*. NC State University. Retrieved from <http://www.csc.ncsu.edu/faculty/jjiang/RootSmart/>
- John, G. H., & Langley, P. (1995). Estimating continuous distributions in Bayesian classifiers. In *Proceedings of the eleventh conference on uncertainty in artificial intelligence* (pp. 338–345).
- JS, & AB. (2014, February). *Android Malware goes “To The Moon!”: Mobile devices are being misused as cash collectors*. G Data Software. Retrieved from <https://blog.gdatasoftware.com/blog/article/android-malware-goes-to-the-moon.html>
- Kiem, H., Thuy, N., & Quang, T. (2004). A Machine Learning Approach to Anti-virus System. In *Proceedings of joint workshop of vietnamese society of ai, sigkbs-jsai, ics-ipsj and ieice-sigai on active mining, hanoi-vietnam* (pp. 61–65).
- Kolter, J. Z., & Maloof, M. A. (2006). Learning to Detect and Classify Malicious Executables in the Wild. *The Journal of Machine Learning Research*, 7, 2721–2744.
- Lawton, G. (2008). Is it finally time to worry about mobile malware? *Computer*, 41(5), 12–14.
- Leavitt, N. (2000). Malicious code moves to mobile devices. *IEEE Computer*, 33(12), 16–19.
- Lee, T. J., & Mody, J. (2006, April). Behavioral classification. In *Proceedings of eicar 2006*.
- Liu, F. (2014, January). *Windows Malware Attempts to Infect Android Devices*. Symantec. Retrieved from <http://www.symantec.com/connect/blogs/windows-malware-attempts-infect-android-devices>

- Lookout. (2011, March). *Technical Analysis: DroidDream Malware*. Author. Retrieved from <https://blog.lookout.com/droiddream/>
- Lookout. (2012). *Lookout Malwarenomics: 2012 Mobile Threat Predictions*. Author. Retrieved from <https://blog.lookout.com/blog/2011/12/13/2012-mobile-threat-predictions>
- Lyne, J. (2013, February). *Everyday cybercrime – and what you can do about it*. TED. Retrieved from [http://www.ted.com/talks/james\\_lyne\\_everyday\\_cybercrime\\_and\\_what\\_you\\_can\\_do\\_about\\_it.html](http://www.ted.com/talks/james_lyne_everyday_cybercrime_and_what_you_can_do_about_it.html)
- Machine Learning Group at the University of Waikato. (n.d.). *Weka*. Retrieved from <http://www.cs.waikato.ac.nz/ml/weka>
- Main, Scott and Braun, David. (2013). *Getting Your SMS Apps Ready for KitKat*. Android Developers Blog. Retrieved from <http://android-developers.blogspot.it/2013/10/getting-your-sms-apps-ready-for-kitkat.html>
- Martignoni, L., Paleari, R., & Bruschi, D. (2009). A framework for behavior-based malware analysis in the cloud. In *Information systems security* (pp. 178–192). Springer.
- Maslennikov, D. (2011, October). *ZeuS-in-the-Mobile - Facts and Theories*. Kaspersky. Retrieved from [http://www.securelist.com/en/analysis/204792194/ZeuS\\_in\\_the\\_Mobile\\_Facts\\_and\\_Theories](http://www.securelist.com/en/analysis/204792194/ZeuS_in_the_Mobile_Facts_and_Theories)
- Mila. (2013). *Contagio Mobile*. Retrieved from <http://contagiominedump.blogspot.it>
- Mullaney, C. (2012, February). *Android.Bmaster: A Million-Dollar Mobile Botnet*. Symantec. Retrieved from <http://www.symantec.com/connect/blogs/androidbmaster-million-dollar-mobile-botnet>
- NQ Mobile. (2012, February). *2011 Mobile Security Report*. Retrieved from [http://docs.nq.com/2011\\_NQ\\_Mobile\\_Security\\_Report.pdf](http://docs.nq.com/2011_NQ_Mobile_Security_Report.pdf)
- Oberheide, J., Cooke, E., & Jahanian, F. (2007). Rethinking antivirus: Executable analysis in the network cloud. In *2nd unix workshop on hot topics in security (hotsec 2007)*.
- Oberheide, J., Cooke, E., & Jahanian, F. (2008). CloudAV: N-Version Antivirus in the Network Cloud. In *Unix security symposium* (pp. 91–106).
- Oi, T. (2011). Yet Another Android Rootkit. *Black Hat*. [https://media.blackhat.com/bhad-11/Oi/bh-ad-11-Oi-Android\\_Rootkit-WP.pdf](https://media.blackhat.com/bhad-11/Oi/bh-ad-11-Oi-Android_Rootkit-WP.pdf) (accessed December 13, 2012).
- Panda Security. (2013a, December). *Malware creation and Android security threats will hit record-high numbers in 2014*. Author. Retrieved from

<http://press.pandasecurity.com/news/malware-creation-and-android-security-threats-will-hit-record-high-numbers-in-2014/>

Panda Security. (2013b). *PandaLabs Report: April-June 2013*. Author. Retrieved from <http://press.pandasecurity.com/wp-content/uploads/2010/05/Quarterly-Report-PandaLabs-April-June-2013.pdf>

Portokalidis, G., Homburg, P., Anagnostakis, K., & Bos, H. (2010). Paranoid Android: versatile protection for smartphones. In *Proceedings of the 26th annual computer security applications conference* (pp. 347–356).

Quinlan, J. R. (1993). *C4.5: programs for machine learning* (Vol. 1). Morgan kaufmann.

Rogers, M. (2014, March). *CoinKrypt: How criminals use your phone to mine digital currency*. Lookout. Retrieved from <https://blog.lookout.com/blog/2014/03/26/coinkrypt/>

Rovelli, P. (2013a, October). *Discovered a new malware for Android which subscribes its victims to paid services via SMS!* TG Soft. Retrieved from [http://www.tgsoft.it/english/news\\_archivio\\_eng.asp?id=565](http://www.tgsoft.it/english/news_archivio_eng.asp?id=565)

Rovelli, P. (2013b, November). *Discovered the first Android malware that uses SMTP!* TG Soft. Retrieved from [http://www.tgsoft.it/english/news\\_archivio\\_eng.asp?id=568](http://www.tgsoft.it/english/news_archivio_eng.asp?id=568)

Rovelli, P. (2013c, September). *The FraudTools arrive also on Android... the CRAM team analyzes AndroidDefender.A!* TG Soft. Retrieved from [http://www.tgsoft.it/english/news\\_archivio\\_eng.asp?id=556](http://www.tgsoft.it/english/news_archivio_eng.asp?id=556)

Rovelli, P. (2013d, September). *Wi-Fi networks and the new bad habits*. TG Soft. Retrieved from [http://www.tgsoft.it/english/news\\_archivio\\_eng.asp?id=555](http://www.tgsoft.it/english/news_archivio_eng.asp?id=555)

Rovelli, P. (2013e, October). *ZitMo for Android: Analysis of a Man-in-the-Mobile attack!* TG Soft. Retrieved from [http://www.tgsoft.it/english/news\\_archivio\\_eng.asp?id=561](http://www.tgsoft.it/english/news_archivio_eng.asp?id=561)

Rovelli, P. (2014, January). *How safe is really Google Play Store?* TG Soft. Retrieved from [http://www.tgsoft.it/english/news\\_archivio\\_eng.asp?id=574](http://www.tgsoft.it/english/news_archivio_eng.asp?id=574)

Rovelli, P., & Tonello, G. (2013, September). *A new menace for Android... it is the TrojanSMS.Agent.A!* TG Soft. Retrieved from [http://www.tgsoft.it/english/news\\_archivio\\_eng.asp?id=554](http://www.tgsoft.it/english/news_archivio_eng.asp?id=554)

Sarma, B. P., Li, N., Gates, C., Potharaju, R., Nita-Rotaru, C., & Molloy, I. (2012). Android permissions: a perspective combining risks and benefits. In *Proceedings of the 17th acm symposium on access control models and technologies* (pp. 13–22).

- SB. (2014, February). *Worth looking again: fake Flash Player apps in Google Play store*. G Data Software. Retrieved from <http://blog.gdatasoftware.com/blog/article/worth-looking-again-fake-flash-player-apps-in-google-play-store.html>
- Schmidt, A.-D., Camtepe, S. A., & Albayrak, S. (2010). Static smartphone malware detection.
- Schmidt, A.-D., Peters, F., Lamour, F., Scheel, C., Çamtepe, S. A., & Albayrak, Ş. (2009). Monitoring smartphones for anomaly detection. *Mobile Networks and Applications*, 14(1), 92–106.
- Schultz, M. G., Eskin, E., Zadok, F., & Stolfo, S. J. (2001). Data Mining Methods for Detection of New Malicious Executables. In *Security and privacy, 2001. s&p 2001. proceedings. 2001 ieee symposium on* (pp. 38–49).
- Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., & Weiss, Y. (2012). “Andromaly”: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1), 161–190.
- Shabtai, A., Moskovitch, R., Elovici, Y., & Glezer, C. (2009). Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *Information Security Technical Report*, 14(1), 16–29.
- Siddiqui, M., Wang, M. C., & Lee, J. (2008). A Survey of Data Mining Techniques for Malware Detection using File Features. In *Proceedings of the 46th annual southeast regional conference on xx* (pp. 509–510).
- Skulason, A. F., & Bontchev, V. (1991). A new virus naming convention. In *Caro meeting*.
- Sophos. (2013a, July). *Anatomy of a security hole - Google’s “Android Master Key” debacle explained*. Author. Retrieved from <http://nakedsecurity.sophos.com/2013/07/10/anatomy-of-a-security-hole-googles-android-master-key-debacle-explained/>
- Sophos. (2013b). *Sophos Security Threat Report 2013*. Author. Retrieved from <http://www.sophos.com/en-us/medialibrary/PDFs/other/sophossecuritythreatreport2013.pdf>
- Spreitzenbarth, M. (2012, February). *Detailed Analysis of Android.Bmaster*. forensic blog. Retrieved from <http://forensics.spreitzenbarth.de/2012/02/12/detailed-analysis-of-android-bmaster/>
- Spreitzenbarth, M., & Freiling, F. (2012). Android Malware on the Rise. *University of Erlangen, Germany, Tech. Rep. CS-2012-04*.
- Stone-Gross, B., Abman, R., Kemmerer, R. A., Kruegel, C., Steigerwald, D. G., & Vigna, G. (2013). The underground economy of fake antivirus software. In *Economics of*

- information security and privacy iii* (pp. 55–78). Springer.
- Strategy Analytics. (2012, 10). *Global Smartphone Installed Base by Operating System for 88 Countries: 2007 to 2017*. Retrieved from <http://www.strategyanalytics.com/default.aspx?mod=reportabstractviewer&a0=7834>
- Suarez-Tangil, G., Tapiador, J. E., Peris-Lopez, P., & Blasco, J. (2014). Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications*, 41(4), 1104–1117.
- Svajcer, V. (2011, March). *Aftermath of the Droid Dream Android Market malware attack*. Sophos. Retrieved from <http://nakedsecurity.sophos.com/2011/03/03/droid-dream-android-market-malware-attack-aftermath/>
- Symantec Security Response. (2013, July). *First Malicious Use of 'Master Key' Android Vulnerability Discovered*. Symantec. Retrieved from <http://www.symantec.com/connect/blogs/first-malicious-use-master-key-android-vulnerability-discovered>
- Szor, P. (2005). *The art of computer virus research and defense*. Pearson Education.
- Szor, P. (2008, August). *Heuristic detection of malicious computer code by page tracking*. Google Patents. (US Patent 7,418,729)
- Tabish, S. M., Shafiq, M. Z., & Farooq, M. (2009). Malware Detection using Statical Analysis of Byte-Level File Content. In *Proceedings of the acm sigkdd workshop on cybersecurity and intelligence informatics* (pp. 23–31).
- The Android Open Source Project. (n.d.). *Building and Running*. Retrieved from <http://developer.android.com/tools/building/index.html>
- The Android Open Source Project. (2013a). *ActivityManager*. Retrieved from <http://developer.android.com/reference/android/app/ActivityManager.html>
- The Android Open Source Project. (2013b). *ActivityManager.RunningAppProcessInfo*. Retrieved from <http://developer.android.com/reference/android/app/ActivityManager.RunningAppProcessInfo.html>
- The Android Open Source Project. (2013c). *Android Permissions*. Retrieved from <http://developer.android.com/guide/topics/security/permissions.html>
- The Android Open Source Project. (2013d). *Application Fundamentals*. Retrieved from <http://developer.android.com/guide/components/fundamentals.html>
- The Android Open Source Project. (2013e). *ApplicationInfo*. Retrieved from

<http://developer.android.com/reference/android/content/pm/ApplicationInfo.html>

The Android Open Source Project. (2013f). *App Manifest*. Retrieved from <http://developer.android.com/guide/topics/manifest/manifest-intro.html>

The Android Open Source Project. (2013g). *ContentObserver*. Retrieved from <http://developer.android.com/reference/android/database/ContentObserver.html>

The Android Open Source Project. (2013h). *Dalvik EXecutable Format (DEX)*. Retrieved from <http://source.android.com/devices/tech/dalvik/dex-format.html>

The Android Open Source Project. (2013i). *Environment*. Retrieved from <http://developer.android.com/reference/android/os/Environment.html>

The Android Open Source Project. (2013j). *External Storage Technical Information*. Retrieved from <http://source.android.com/devices/tech/storage/>

The Android Open Source Project. (2013k). *FileObserver*. Retrieved from <http://developer.android.com/reference/android/os/FileObserver.html>

The Android Open Source Project. (2013l). *InputStream*. Retrieved from <http://developer.android.com/reference/java/io/InputStream.html>

The Android Open Source Project. (2013m). *Introducing ART*. Retrieved from <http://source.android.com/devices/tech/dalvik/art.html>

The Android Open Source Project. (2013n). *OpenJDK java.io.InputStream*. Retrieved from <http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/io/InputStream.java#InputStream.skip%28long%29>

The Android Open Source Project. (2013o). *OutOfMemoryError*. Retrieved from <http://developer.android.com/reference/java/lang/OutOfMemoryError.html>

The Android Open Source Project. (2013p). *PackageManager*. Retrieved from <http://developer.android.com/reference/android/content/pm/PackageManager.html>

The Android Open Source Project. (2013q). *RandomAccessFile*. Retrieved from <http://developer.android.com/reference/java/io/>



[RandomAccessFile.html](#)

The Android Open Source Project. (2013r). *Signing Your Applications*. Retrieved from <http://developer.android.com/tools/publishing/app-signing.html>

The Android Open Source Project. (2013s). *Storage Options: Using the External Storage*. Retrieved from <http://developer.android.com/guide/topics/data/data-storage.html#filesExternal>

The Android Open Source Project. (2013t). *System Permissions*. Retrieved from <http://developer.android.com/guide/topics/security/permissions.html>

The Android Open Source Project. (2013u). *TrafficStats*. Retrieved from <http://developer.android.com/reference/android/net/TrafficStats.html>

The Android Open Source Project. (2013v). *UID (kernel User-ID)*. Retrieved from <http://developer.android.com/reference/android/content/pm/ApplicationInfo.html#uid>

The Android Open Source Project. (2013w). *ZipFile*. Retrieved from <http://developer.android.com/reference/java/util/zip/ZipFile.html>

The University of Waikato. (2002). *Attribute-Relation File Format (ARFF)*. Retrieved from <http://www.cs.waikato.ac.nz/ml/weka/arff.html>

The University of Waikato. (2008). *ARFF*. Retrieved from <http://weka.wikispaces.com/ARFF>

Tonello, G. (2014, April). *Conto corrente sotto attacco: come l'evoluzione dei Trojan Banker minacciano i nostri soldi...* SMAU. Retrieved from <http://www.smau.it/padoval4/schedules/conto-corrente-sotto-attacco-come-levoluzione-dei-trojan-banker-minacciano-i-nostri-soldi/>

Unuchek, R. (2013, June). *The most sophisticated Android Trojan*. Kaspersky. Retrieved from [https://www.securelist.com/en/blog/8106/The\\_most\\_sophisticated\\_Android\\_Trojan](https://www.securelist.com/en/blog/8106/The_most_sophisticated_Android_Trojan)

Uscilowski, B. (2013, October). *Mobile Adware and Malware Analysis*. Symantec. Retrieved from [http://www.symantec.com/content/en/us/enterprise/media/security\\_response/whitepapers/madware\\_and\\_malware\\_analysis.pdf](http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/madware_and_malware_analysis.pdf)

Wei, X., Gomez, L., Neamtiu, I., & Faloutsos, M. (2012). Permission evolution in the Android ecosystem. In *Proceedings of the 28th annual computer security applications*

- conference* (pp. 31–40).
- Williams, G. (2012, March). *The digital detective: Mikko Hypponen's war on malware is escalating*. Wired. Retrieved from <http://www.wired.co.uk/magazine/archive/2012/04/features/the-digital-detective>
- Witten, I. H., & Frank, E. (2005). *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.
- Wyatt, T. (2011, August). *Inside the Android Security Patch Lifecycle*. Lookout. Retrieved from <https://blog.lookout.com/blog/2011/08/04/inside-the-android-security-patch-lifecycle/>
- Xiao, Z., Dong, Q., Zhang, H., & Jiang, X. (2014, January). *Oldboot: the first bootkit on Android*. Qihoo 360 Technology. Retrieved from <http://blogs.360.cn/360mobile/2014/01/17/oldboot-the-first-bootkit-on-android/>
- Ye, Y., Wang, D., Li, T., & Ye, D. (2007). IMDS: Intelligent Malware Detection System. In *Proceedings of the 13th acm sigkdd international conference on knowledge discovery and data mining* (pp. 1043–1047).
- Zhou, W., Zhou, Y., Jiang, X., & Ning, P. (2012). Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *Proceedings of the second acm conference on data and application security and privacy* (pp. 317–326).
- Zhou, Y., & Jiang, X. (2012). Dissecting Android Malware: Characterization and Evolution. In *Security and privacy (sp), 2012 ieee symposium on* (pp. 95–109). Retrieved from <http://www.malgenomeproject.org>
- Zhou, Y., Wang, Z., Zhou, W., & Jiang, X. (2012). Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the 19th annual network and distributed system security symposium*.
- Zorabedian, J. (2014, May). *Android "police warning" ransomware - how to avoid it, and what to do if you get caught*. Sophos. Retrieved from <http://nakedsecurity.sophos.com/2014/05/19/android-police-warning-ransomware-how-to-avoid-it-and-what-to-do-if-you-get-caught/>







School of Computer Science  
Reykjavík University  
Menntavegi 1  
101 Reykjavík, Iceland  
Tel. +354 599 6200  
Fax +354 599 6201  
[www.reykjavikuniversity.is](http://www.reykjavikuniversity.is)  
ISSN 1670-8539