You Shall Not Pass!

Measuring, Predicting, and Detecting Malware Behavior

PhD Thesis

Enrico Mariconti

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Security and Crime Science

University College London

April 25, 2019

I, Enrico Mariconti confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

To Valentina,

who started dating me at the beginning of my student path at university,

and married me at the end of it.

Acknowledgments

This is the last section people write on a dissertation, but, actually, many times we think about it in advance. I always spent a bit of time to focus on it. Today it will be the same as the acknowledgements aren't something I have to write, but a real thank you to the people I have shared a part of this path with. If you won't find your name, but you are reading this dissertation not just because I tweeted about it, most probably it is because it is hard to name each and every person and I tried not to forget anyone by using the groups we've been part of.

I moved to London on Saturday the 27th of September 2014 with this PC I am writing from and a lot of question marks on how my future would have been. The day after, Sampdoria won a derby (1-0 with goal of Manolo Gabbiadini). Today, Monday 15th of April 2019 I am writing these acknowledgements, the day after Gregoire Defrel and Fabio Quagliarella secured another derby for 2-0. In the middle a lot of water under the bridges passed (including other 4 derbys won, 3 drew and only 1 lost), a lot of time, a lot of experiences, a lot of emotions.

Speaking about spent time, I will start from the end: from Steven and Thorsten, my examiners, who actually spent a considerable amount of hours reading carefully my work and almost three hours making me questions and carefully evaluating me.

The person who has invested more time in following me in this PhD is Gianluca, who, with his calm and objective evaluations, has always been able to understand how to make me give my best. He is a great supervisor, a model

and a mentor, from which I will take inspiration for my academic career. He managed perfectly all my weird aspects, my most impulsive moments as well as my ups and downs.

I feel blessed from how many good people I found in my PhD path. I'm thanking Gordon, my second supervisor, Emiliano, coauthor and mentor who helped me a lot in my growth as researcher, and all the coauthors I've had, I had the opportunity to learn something from all of you.

Such a path would not be possible without a good environment around me. All the people part of the Information Security and, more in general, of Computer Science that have spent valuable time with me, from social hour, to sports chats with Lucky and Sarah to Jeremiah asking me many times how did I feel after the Ponte Morandi tragedy. All the people in Crime Science, from the admins (it has always been good to enter the department and see the smiles of people like Amy, Andrea, Ellie, and Gavin) to the academic staff and the students. Some of them have been there since 2014, some others joined this crazy journey much more recently, but I am thanking all of you!

It has been great to meet people that became more than collaborators or colleagues: Florian, Patricio, Nadine, Michael, Toby, Andrea, Mirko. From meeting at a random conference to watching football or rugby together; from drinks, dinners, and events to coffee breaks with someone trusted, with who you can talk freely about everything.

One little mention goes to the fencing clubs I've been part of and their members: almost like families in which I tried a new fantastic sport, in which I had a break from the stress of the work and I focused only on landing that point on my opponent. Thanks UCL Fencing Club, Haverstock, and SAM.

Another thank you is for all my friends, both in Italy and those met here, as they have always been the nicest people to me.

What people tend to not mention of a PhD is that it still brings heavy challenges, stress, and anxiety, in some cases it could be even worse. Whether you became my family later, or you share a good amount of genes with me,

whether you're on this earth, or looking after us from somewhere else, thank you for all the priceless support you gave in these years every time it was needed. I know you all will still be there for me and Valentina, and I hope we will be able to give back the same support when needed.

Last, but not least, my wife, a wonder to my eyes, the person I will be next to for our whole lives, thank you for everything, Valentina.

Abstract

Researchers have been fighting malicious behavior on the Internet for several decades. The arms race is far from being close to an end, but this PhD work is intended to be another step towards the goal of making the Internet a safer place. My PhD has focused on measuring, predicting, and detecting malicious behavior on the Internet; we focused our efforts towards three different paths: establishing causality relations into malicious actions, predicting the actions taken by an attacker, and detecting malicious software. This work tried to understand the causes of malicious behavior in different scenarios (sandboxing, web browsing), by applying a novel statistical framework and statistical tests to determine what triggers malware. We also used deep learning algorithms to predict what actions an attacker would perform, with the goal of anticipating and countering the attacker's moves. Moreover, we worked on malware detection for Android, by modeling sequences of API with Markov Chains and applying machine learning algorithms to classify benign and malicious apps. The methodology, design, and results of our research are relevant state of the art in the field; we will go through the different contributions that we worked on during my PhD to explain the design choices, the statistical methods and the takeaways characterizing them. We will show how these systems have an impact on current tools development and future research trends.

Impact of this Work

This dissertation is focusing on three different aspects to tackle malicious behavior on the Internet from different angles.

A first technical chapter focused on extracting malicious behavior and measuring it through causal relationships. This section starts with a rather theoretical contribution related to how to apply the causality framework in the field and a first attempt of practical application as part of the second work. The philosophy behind this part is to change the sandboxing phase of malware analysis: we often study what malware samples are doing by letting them run in a safe environment. In these works we show that sandboxing can induce the malware samples in doing something different, injecting specific actions to trigger the samples to show their complete behavior. The last paper related to this chapter, the system called Ex-Ray, is a proper application of the framework. It is showing how to automatically detect extensions that may leak the browser history. We used a sandboxing infrastructure and refined the experimental methodology to apply the causality framework. Academics have asked more details about the idea that we truly believe can start a different way of approaching the sandboxing analysis.

The second technical chapter is about Tiresias, a system used to predict multi-step attacks that can be flagged as security events. With respect to the previous part, where the impact can be found in the future scenarios of analyzing malware and on the possible academic directions in the topic, this system has a straightforward practical development. Tiresias was developed in collaboration with Symantec researchers that were aiming to find new solutions

to be deployed on their security systems. From an academic point of view, this work is, among others, trying to shift the attention from working only on detection of malicious activities, to prediction of these events, to ultimately prevent them from happening or mitigating their effects. This is a new trend where relevant works are growing and the publication of this paper to such an important venue as Computer and Communication Systems (CCS) 2018 may boost the trend. Tiresias has been particularly appreciated by the Information Security community as it has been one of the finalists for the applied research of the year in the Cyber Security Awareness Week (CSAW) Europe 2018.

The last technical chapter is related to Android Malware Detection. We created a system called MaMaDroid, exploring the opportunities of using Markov Chains to model sequences of API Calls. MaMaDroid is based on static analysis and has been tested over one of the largest Android malware datasets used in research, analyzing how samples change over time and the robustness of the system to the challenges faced by this type of detection systems. From an academic point of view, the importance of these works is already noticeable: the first MaMaDroid Camera Ready version went online the 16/12/2016 on *arxiv* and, in two years, the paper reached 55 citations. Moreover, MaMaDroid has been one of the applied research competition finalists, finishing 2nd in CSAW Europe 2017.

Contents

1 Introduction **27**

1.1 The Malware World 27

1.2 Statistics and Learning Algorithms 29

1.3 The Rationale Behind this Work 30

1.4 Side Projects I Contributed to During my PhD 31

1.5 Contributions Statement 32

1.6 The Document Outline 33

2 Background and Related Work **35**

2.1 Malware Behavior 35

2.2 Defense Systems 37

2.2.1 Intervention Actions and Areas 38

2.2.2 Defensive Tools 39

2.3 Statistical Methods 40

2.3.1 Correlation and Causation 40

2.3.1.1 Counterfactual Analysis Causality 41

2.3.2 Machine Learning and Deep Learning Algorithms . . . 41

2.3.2.1 What is Machine Learning? 41

2.3.2.2 Machine Learning Evaluation 43

2.3.2.3 K-Nearest Neighbors Classifier 46

2.3.2.4 Random Forests 47

2.3.2.5 Deep Learning 48

2.4 Causality in Malware Traffic 50

2.5 Browser Abuse 52
2.6 Prediction of Malicious Activities 55
2.6.1 Security Events Sequences and the Application of Deep Learning Algorithms 56
2.6.1.1 Security Event Forecast 56
2.6.1.2 Recurrent Neural Network Applications in Security Research 57
2.7 Android Malware and Static Analysis 59
2.7.1 Static Analysis, Markov Chains and Malware Classification 60
2.7.1.1 Program Analysis 60
2.7.1.2 Android Malware Detection 61

3 Causality Assessment in Malware Activities Using Counterfactual Analysis 65
3.1 Causality in Malware Traffic 65
3.1.1 Approach Formalization 66
3.1.2 Experimental Environment 66
3.1.3 Sandboxing Background 67
3.1.4 Sandbox Implementation Details 69
3.1.5 Extracting and Labeling Network Conversations 70
3.1.6 Chains of Events 72
3.1.7 Statistical Analysis 73
3.2 Causality Framework Application: Malware Network Traffic . 74
3.2.1 Application of the Methodology 74
3.2.1.1 Employed Dataset 75
3.2.1.2 Instantiation of the Experiments 76
3.2.1.3 Extraction and Labeling Network Conversation 77
3.2.1.4 Labeling and Chains Settings 79
3.2.2 Evaluation 80
3.2.2.1 Labeling Results 81

3.2.2.2 Beta Distributions 82
3.2.2.3 Statistical Evaluation of Causality and Experimental Validity 83
3.2.3 Discussion 84
3.2.3.1 Labeling Results 85
3.2.3.2 Results and Validity 87
3.2.4 Limitations 88
3.3 Causality Framework Application: Browser History Leakage . 88
3.3.1 Linear Regression and Causality Background 89
3.3.2 The Environment 90
3.3.2.1 HTTP URL Honeypot 91
3.3.2.2 Types of Trackers 91
3.3.2.3 Threat Model 92
3.3.3 Ex-Ray Methodology 93
3.3.3.1 Overview 93
3.3.3.2 Application of Counterfactual Analysis . . . 96
3.3.4 Ex-Ray Counterfactual Analysis Evaluation 98
3.3.4.1 Experimental Setting 98
3.3.4.2 Ex-Ray Counterfactual Analysis Results . . 101
3.3.5 Discussion and Limitations 103

4 Predicting Security Alarms due to Malicious Activities Using Deep Learning Algorithms 105
4.1 Motivation 107
4.2 Methodology 108
4.2.1 Architecture Overview 108
4.2.2 Recurrent Memory Array 111
4.3 Employed Dataset 113
4.4 Evaluation 115
4.4.1 Experimental Setup 115
4.4.2 Overall Prediction Results 116

4.4.3 Comparison Study 118
4.4.4 Influence of Training Period Length 119
4.4.5 Stability Over Time 121
4.4.6 Sequence Length Evaluation 124
4.4.7 Tiresias Runtime Performance 127
4.5 Case Studies 127
4.5.1 Predicting Events in a Multi-Step Attack 128
4.5.2 Adjusting the Prediction Granularity 131
4.6 Discussion 133

5 Detecting Malware by Using Markov Chains as Behavioral Models 137
5.1 MaMaDroid: Using Static Analysis to Detect Malware . . . 138
5.1.1 Overview 138
5.1.2 Call Graph Extraction 139
5.1.3 Sequence Extraction 140
5.1.3.1 Abstraction to Classes 142
5.1.4 Markov Chain Based Modeling 143
5.1.5 Classification 145
5.2 Datasets 146
5.2.1 Employed Dataset 146
5.3 MaMaDroid Evaluation 150
5.3.1 Preliminaries 150
5.3.2 Detection Performance 151
5.3.3 Detection Over Time 154
5.3.4 Case Studies of False Positives and Negatives 156
5.3.5 MaMaDroid vs DroidAPIMiner 158
5.3.6 Runtime Performance 160
5.3.7 Finer-Grained Abstraction 162
5.3.8 Reducing the Size of the Problem 163
5.3.9 Class Mode Results 163

5.3.10 Detection Over Time 164
5.4 Discussion 165
5.4.1 Lessons Learned 165
5.4.2 Evasion 167
5.4.3 Limitations 169

6 Discussion 171

7 Ethical Discussion of this Work 177
7.1 Research Analysis and Ethics 177
7.2 Systems Ethics and Implementation in the Wild 180

8 Conclusions and Final Remarks 183

Bibliography 186

List of Figures

2.1 Decision tree example from [1]. 47

2.2 Neural network example from [2]. 49

3.1 Sandbox infrastructure: the host machine simulates a network of VMs, a webserver managing the malware distribution, a mailserver to redirect possible spam campaigns from the infected VMs, and a router that allows the connection among the different internal machines and with the Internet world. 68

3.2 Overview of our approach. 75

3.3 The Beta distributions related to the **Nav tests. The dotted line is the Beta distribution of InfNav tests, the dashed line represents the AdNav tests, and the full line represents the OtNav tests. 82

3.4 The Beta distributions related to the **Log tests. The dotted line is the Beta distribution of InfLog tests, the dashed one represents the AdLog tests, and the full one represents the OtLog tests. 82

3.5 The causality probabilities of **Nav tests varying the used fraction of the dataset. The dotted line is related to the InfNav tests, the dashed line represents AdNav tests, and the full line represents OtNav tests. When the dataset is more than 80% the three lines become stable. 85

3.6 The causality probabilities of **Log tests varying the fraction of used dataset. The dotted line represents the InfLog tests, the dashed one represents AdLog tests, and the full line is related to the OtLog tests. In this case the observations maintain the same values regardless of the used fraction, because the Beta distributions are extremely different. 85

3.7 The causality probabilities of Tests**Nav varying the number of observations. The dotted line is related to Tests InfNav, the dashed line is Tests AdNav, and the full line is Tests OtNav. When the observations are more than 50, the three lines maintain the same values. 86

3.8 The causality probabilities of Tests**Log varying the number of observations. The dotted line is related to Tests InfLog, the dashed line is Tests AdLog, and the full line is Tests OtLog. In this case the observations maintain always the same values because the beta distributions are extremely different. 86

3.9 Extension execution with unique URLs vs. incoming connections to those URLs from the public Internet. These connections confirm that leaked browsing history is used by the receivers, often immediately upon execution. 91

3.10 Ex-Ray architectural overview. A classification system combines unsupervised and supervised methods. After triaging unsupervised results, a vetted dataset is used to classify extensions based on n-grams of API calls. 94

3.11 Comparison in change of traffic between executions leaking history and benign extensions. Each bar displays the change of traffic sent relative to executions with increased history. Sent data projects an ascending slope based on size of history. . . . 95

3.12 Ex-Ray extension execution overview. 99

4.1 Three endpoints undergoing a coordinated attack. $\{e_0, ..., e_{13}\}$ are events involved in the coordinated attack and highlighted in bold. 107

4.2 TIRESIAS collects security events from machines that have installed an intrusion protection product. The sequential events from these machines are collected, preprocessed and then used to build and validate TIRESIAS' predictive model. The optimal model is then used in operations and its performance is monitored to ensure steadily high prediction accuracy. 109

4.3 Summary of the security event datasets used in this paper. . 113

4.4 Experimental setup for TIRESIAS' prediction evaluation (Section 4.4.2) and comparison study with baseline methods (Section 4.4.3). The grey bars indicate data derived from machines used for training while the dotted bars indicate the data used for testing and coming from different machines with respect to the training data. 116

4.5 Precision, Recall, and F1-Measure of overall TIRESIAS's performance. TIRESIAS is trained using one day of data and evaluated on both the same day and the following days until 5 November 2017. 117

4.6 Experimental setup for multiple day evaluation of TIRESIAS (Section 4.4.4). 120

4.7 Experimental setup for TIRESIAS reliability evaluation (Section 4.4.5). 122

4.8 Quantity of successfully and unsuccessfully guessed events. The Y axis on the left of each graph is the occurrence of successes/failures with at least the probability indicated on the X axis according to the system. The Y axis on the right is the ratio between the value on the other Y axis and the total of successes/failures. 125

4.9 The plots show the percentage of the sequences correctly guessed (a) or failed to guess (b) with respect to sequences that share all the events but the last. On the X axis, as for Figures 4.8a and 4.8b there is the confidence level of the sequences used by the system. Figures show that sequences of at least 5 events (sl >=5) are quite unique, therefore long term memory is a crucial factor in the system accuracy. 126

4.10 Step by step visualization of TIRESIAS prediction process in two real systems. TIRESIAS starts with event e_{10} and e_{31} respectively as the initial feed and predicts the upcoming security event step by step. The predictions are colored by their probabilistic scores, where green indicates TIRESIAS returns a correct prediction with probabilistic score greater than 0.5, orange indicates TIRESIAS returns a correct event prediction with probabilistic score less than 0.5 (but remains the largest probabilistic score), and red indicates a wrong prediction (the actual events are shown in parentheses in this case). 129

5.1 Overview of MAMADROID operation. In (1), it extracts the call graph from an Android app, next, it builds the sequences of (abstracted) API calls from the call graph (2). In (3), the sequences of calls are used to build a Markov chain and a feature vector for that app. Finally, classification is performed in (4), labeling the app as benign or malicious. 139

5.2 Code snippet from a malicious app (com.g.o.speed.memboost) executing commands as root. 141

5.3 Call graph of the API calls in the try/catch block of Figure 5.2. (Return types and parameters are omitted to ease presentation). 142

5.4 Sequence of API calls extracted from the call graphs in Figure 5.3, with the corresponding package/family abstraction in square brackets. 143

5.5 Markov chains originating from the call sequence example in Section 5.1.3 when using packages (a) or families (b). 144
5.6 CDF of the number of API calls in different apps in each dataset. 148
5.7 CDFs of the percentage of `android` and `google` family calls in different apps in each dataset. 148
5.8 Positions of benign vs malware samples in the feature space of the first two components of the PCA (family mode). 149
5.9 F-measure of MaMaDroid classification with datasets from the same year (family mode). 151
5.10 F-measure of MaMaDroid classification with datasets from the same year (package mode). 153
5.11 F-measure values in the different test settings. 155
5.12 F-measure values in the different test settings. 156
5.13 F-measure values in the different test settings. 164

List of Tables

2.1 Table explaining the possible actions and tools depending on the attacking phase in [3] 38

3.1 Encoding of the labels. Domains contacted during tests are labeled following this table. Running VMs without any malware infecting them allows to find the conversations labeled as "Trigger", while running an infected VM in idle is how we assign to the conversations the label"Pre-Trigger". When the label to be assigned is "Triggered", it can be assigned only if that domain is not already in previous ones. 71

3.2 Summary of our test cases. 77

3.3 Number of repetitions per test. 79

3.4 Labels encoding per each test. 79

3.5 Labels for the tests in which the VM is navigating to *amazon.com*. 81

3.6 Percentages of the different labels for the tests with Log VMs. 81

3.7 Top five extensions connecting to our honeypot with highest installation numbers which are still available in the Chrome Web Store. 101

4.1 Prediction precision comparison study: Tiresias vs. baseline approaches. 118

4.2 Evaluation of Tiresias' prediction precision between 8th November and 17th November. 122

4.3 Evaluation of Tiresias's prediction precision on every 8th and 23rd of each month. 124

5.1 Overview of the datasets used in our experiments. 146

5.2 F-measure, precision, and recall obtained by MaMaDroid, using Random Forests, on various dataset combinations with different modes of operation, with and without PCA. 151

5.3 Classification performance of DroidAPIMiner [4] vs MaMaDroid (our work). 157

5.4 MaMaDroid's Precision, Recall, and F-measure when trained and tested on dataset from the *same* year in class and package modes. 163

Chapter 1

Introduction

The aim of this dissertation is to explore the application of statistics to the malware world. While the use of statistical techniques, such as Machine Learning, is not novel, we accurately adapted these techniques to approach new problems. Moreover, we applied statistical tests for preliminary detailed analysis that have never before been used in this field. Nevertheless, we do not limit the use of statistics to detecting malicious samples, we also apply them to establish causality relationships and predict what can be the following malicious action.

1.1 The Malware World

History. Internet has its roots back in the 1960s, when the ARPA and ARPANET projects were financed by the United States Department of Defense [5]. It started with the idea of communicating easily among different networks of research institutions, military corps, and companies. Its goal was to be a facilitator and catalyst of ideas and documents exchanges, but security was not a requirement in its creation. Antivirus software and other security systems, as well as the current encryption algorithms have been created to satisfy this requirement. The first malicious software (malware) was a virus, created in 1982 [6]. The name has its origins in biology, where a virus is infecting a body and self replicating into its cells. In informatics, a virus is a file able to self replicate into other files when triggered into a computer. This

name and definition have been officially used for the first time by Fred Cohen in 1985 [7]. From this first sample, the development of malware families increased dramatically and the aims and threats of different samples became extremely varied.

Sir Tim Berners-Lee invented the World Wide Web (commonly known as WWW) in October 1990 and the first website[1] [8] went online on the 6th August 1991. This moment has been the turning point in the history of the Internet that quickly became as we know it now. These dramatic changes affected everything around the Internet as from a tool for a chosen elite, it became a resource for everyone. The economic value of the Internet increased exponentially when new services, such as e-commerce, were created, sensitive information started to be stored in online servers, and the entire communication and telecommunication world was revolutionized. This new scenario changed the malware world as well: until the 1990s hackers who wanted to show off their skills or operated for their personal interests created malware samples. Since this revolution happened, even the malware ecosystem became a delivered service. People started to pay hackers to write malicious code for them or to deliver their own contents through the infected networks controlled by these cybercriminals. This escalation brought to the creation of more than 57 million new malware samples in the third quarter of 2017 [9], complex and dangerous attacks targeting specific victims, and proper cyberwars among countries as, for instance, the events during the US elections show [10].

Current Days. Nowadays the information security community is facing several challenges. The threats are different and malicious software is only one of the fronts the community has to face. The efforts to contrast attacks related to malicious software led to the creation of many defense layers as explained in Section 2.2, but the spectrum of the possible attacks has been enlarged by the opportunities that clever minds have found to perpetrate these attacks. Nowadays, for instance, there are botnets [11], droppers [12], ransomware [13], and

[1] `http://info.cern.ch/hypertext/WWW/TheProject.html`

information stealers [14]. Botnets are networks of infected computers waiting for instructions for possible attacks, droppers are malware samples that are asking to a Command and Control (C&C) Server for another malicious sample. Ransomware samples encrypt the hard disk of the infected PC and ask for a ransom to (maybe) decrypt it. Information stealers are particular pieces of code that are storing sensitive information (credit cards data, login credentials, etc etc) from the infected PC and sending it to a remote server.

In the last years a new kind of attacks emerged: targeted attacks, exploiting specific vulnerabilities that the attackers know are present in a certain system of a specific victim. The Stuxnet [15] case is one of the most famous destructive examples where a malicious software infected Iranian nuclear plants to destroy their centrifuges. Another clear example of how powerful and dangerous malware samples are nowadays is the series of ransomware campaigns carried out in the last two years. The most famous examples are WannaCry and NotPetya; the first one infected 200,000 PCs in over 150 countries, generating revenues for hundreds of thousands of dollars to the hackers and losses to infected companies for more than 4 billions [16]. NotPetya is a ransomware that targeted mainly Ukrainian computers in what has been declared by CIA to be an attack from the Russian secret services [17]. In this scenario of several and varied threats the arms race between defenders and cybercriminals is continuously evolving.

1.2 Statistics and Learning Algorithms

According to the Oxford dictionary of statistical terms, “statistics is the study of the collection, analysis, interpretation, presentation, and organization of data.” From this broad definition, in the information security field, the interpretation and presentation topics have been taken for defensive and offensive actions as I will explain in Section 2. Statistical methods allow to analyze if the collected data is biased or to get some insights on the dataset. It can happen through the dataset representation or through some specific tests such

as Chi Square test, ANOVA, or Cramer Von Mises Test. These tests are used to extract some information on the experimental datasets we collect, allowing some preliminary considerations.

An important part of the statistical methods (and extremely relevant to this thesis) used in the information security field is the set of Machine Learning algorithms, including the Deep Learning ones, a peculiar subset of the Machine Learning algorithms. These methods operate the classification of samples into defined labels, giving a numerical evaluation of the results, such as Accuracy, Precision, and Recall. In our case the classified samples could be pieces of code that we want to recognize, while the label we want to assign as a result of the classification could be "malware" or "benign." This procedure can be done in two ways: (a) by using known samples to elaborate decision rules and, according to these rules, to label the unknown sample (supervised machine learning classification), or (b) by detecting anomalies from an initial samples population and raising alarms when anomalies are identified (anomaly detection systems). Current defensive systems implement their security procedures based on these approaches.

1.3 The Rationale Behind this Work

As just mentioned, statistical methods are broadly used in security systems (e.g., anomaly detection systems); however, this application is not as wide and efficient as it could be. For instance, most people receive at least one spam e-mail a day, containing a malicious URL or attached file and, most of the time, these malicious components are not detected and blocked. The efforts of private companies and the research community are continuous, but the adversaries are leading in this arms race: they find a new way to perpetrate an attack and, only after, the community finds a countermeasure for the specific problem. For instance, when the first ransomware samples were created, there was no existent defense able to identify their activities as our community was not able to identify possible new threats in the future.

This research work has the aim of elaborating systems that are (a) resilient to time evolution or other factors that affect their implementation in the wild (to reasonable extents), and that (b) identify interesting relationships in malware operations or the detection of advanced threats communications. These goals are not the only focus of work: the accurate use of descriptive statistics and, in some cases, tests that have been used in fields different from information security, are important factors of novelty of this work. We decided to

1.4 Side Projects I Contributed to During my PhD

During my PhD I did not work only on the technical contributions explained in this thesis. While working on the core ideas of my PhD I have had the opportunity to work on other projects and collaborate with other researchers.

The first project I will talk about is related to the use of the same principles we designed and used in MaMaDroid, but using dynamic analysis to extract the sequences. The system is a sister of MaMaDroid and for this reason it is called AuntieDroid [18]. AuntieDroid is using different apps stimulators: humans (through CHIMP [19]) and monkeys (automated inputs for dynamic analysis). The project analyzed the advantages of using an hybrid system that brings together the decisions made by MaMaDroid and AuntieDroid.

The sandboxing infrastructure was shared with part of Jeremiah Onaolapo's PhD work. He used it for [20, 21] where I also helped with statistical testing.

Finally, my research work covered some topics related to the social networks world. I am part of the iDrama Lab[2], studying fringe communities behavior as part of the ENCASE H2020 European project. I participated to one of the projects (available on arxiv and currently under submission [22]);

[2] `https://idrama.science/`

we studied a way to identify which videos could be object of coordinated hate attacks from these fringe communities. Another wonderful team work was the study of the profile name reuse phenomenon on Twitter where we studied how it is possible to misuse the handles that have been abandoned by other Twitter users [23, 24].

1.5 Contributions Statement

This thesis is composed by works in which I collaborated with other PhD students. I, therefore, state my contributions that will be explained and shown in this document.

Causality in Malware Activities Chapter. Chapter 3 is constituted by the works on causality in malicious activity. The methodology [25] and the work applied to network activity [26] are coauthored with Jeremiah Onaolapo. Jeremiah and I worked together on the sandboxing infrastructure with which I started working on the causality topic. The same infrastructure that have been used for Jeremiah's works on gmail honeyaccounts [20, 21] and, in its basic structure, the sandbox have been used for [27]. While the sandbox is a shared work between me and Jeremiah, the causality methodology and its application to the network traces themselves are contributions of mine, supervised by Dr Stringhini for the experimental parts and the writing and supervised by Dr Ross on the statistical framework implementation.

Chapter 3 includes another work as application of this methodology: Ex-ray [28]. The sandboxing system was created by the first author, Michael Weissbacher. In [28] we applied linear regression to the data to be able to apply the causality framework already used in the previously cited works; the application of linear regression and the causality framework have been my contributions. Dr Stringhini, Dr Suarez-Tangil, Dr Robertson, and Prof Kirda helped in the writing phase while supervising and periodically checking the experimental phase, giving suggestions on potential changes in the design choices.

Prediction of Security Alarms. Chapter 4 presents our work on prediction of security alarms. It is describing the work on TIRESIAS, the system created in collaboration with researchers from Symantec Research Labs. Yun Shen, first author and Symantec researcher, owned the database and the neural network architecture. Designing the test and the benchmarks and analyzing thoroughly the results and TIRESIAS characteristics has been my main contribution to the work. It results in producing the crucial sections of the work and the evaluation of its impact and practicable employment. Pierre-Antoine Vervier analyzed the case studies while Dr Stringhini closely supervised the work and contributed to the paper writing.

Android Malware Detection. Chapter 5 presents the works over the detection of Android Malware using static analysis and Markov Chain models. Its first work is the MaMaDroid NDSS article [29] where the architecture and its detection performances based on static analysis are presented; an evolution of this work is presented in [30]. Lucky Onwuzurike is the PhD student I collaborated with on these works. He developed all the parts related to static analysis and the extraction of the sequences of API calls. My contribution has been the idea, design, coding, and implementation of the Markov Chain models on the sequences of API calls, the design of the classifiers using Machine Learning algorithms and the evaluation framework for [29]. Lucky used the scripts from [29] in [30], while eliminating a bug on the code abstraction. He selected the datasets for the evaluation while I helped with the results interpretation and the writing. The literature review of these papers has been the contribution of Dr Andriotis. Dr Ross supervised me when designing the Markov Chains, while Dr Stringhini and Dr De Cristofaro supervised the experimental phases and helped writing.

1.6 The Document Outline

This document is divided as follows. Section 2 is a detailed literature review of the explained topics and used tools of the following sections; Section 3 is about

the works on causality models in malware activities. Section 4 is describing the work on predicting security alarms using deep learning models. Section 5 is related to the Android malware detection systems applying Markov Chains to model the behavior of the examined apps. These technical chapters are followed by a global discussion on the work of this thesis (Section 6). There is a specific section dedicated to the ethical issues of my research (Section 7). The conclusions and final remarks (Section 8) completes this document.

Chapter 2

Background and Related Work

In this chapter we are going to explain the several papers that are foundation or in relation with the ones that compose this dissertation. We will first analyze the malware environment and how it evolved. This will be followed by the system of defenses implemented by the security community. As the statistical methods are a crucial part of the works we published, there will be a section related to the theories and algorithms applied before going through the works that are related to the specific fields covered by the dissertation work.

2.1 Malware Behavior

In Chapter 1 we briefly drew a history of Internet and malware. We established that one of the reasons why the arms race between cybercriminals and the security community is so complicated is the fact that security has never been a primary requirement since computers and the Internet were invented. We have also given a hint on how much profit can be done by cybercriminals, mainly due to the fact that we continuously store valuable information on the Internet, control expensive machinery and goods through remote machines, without having enough defenses to prevent systems breaches.

Breaches can have different characteristics and can consist in (a) hackers manually entering a system unauthorized as well as (b) large scale infections through malicious software. While the first case was the most common in the early days, it became less used now: hacking a system may not be profitable as

it is not always possible to reach valuable information or assets; moreover, it is not an approach scalable to operations involving a high number of systems.

Malware infections have caused more relevant issues in the recent period. They can be extremely diverse and can be used for very different purposes; moreover, in some cases malware usage does not require particular skills. For instance, it is now possible to rent botnets portions as-a-service and different botnets collaborate in the same attacking campaigns, as shown in [31]. The attacker renting the botnet does not need technical instructions as they do not manage the botnet. On the other hand, there are very skilled attackers able to create very complex pieces of code to penetrate specific systems and operate stealthy multi-step attacks effectively. Cybercriminals are a variegate set of people: there are still the enthusiast hackers trying to show off their skills like in the 80s and the 90s, but there are also hackers that are using these skills to pursue organizations goals, from cyberactivists like Anonymous (some of these associations are even defined cyberterrorists in some cases) to organized crime groups and state actors.

The variety of possible attacks, targets, and skills require systems able to evaluate a large variety of threats and not focusing on single issues. Moreover, as mentioned in the introduction, there are several malware families, sharing only partially their goals and, in some cases, pieces of code. We mentioned some of the most important families nowadays, but we did not talk about what the behavior of malware is. This word is used in many different context, from what code is used, which functions and which language, to which communications are operated. On a higher level, behavior is sometimes defined by the actions logged by the infected machines or by the network security systems. The behavior definition does not depend only on the malware sample, but on where and in which way we desire to describe the behavior.

In this dissertation we talk about behavior on different levels, in Section 3 we approach the behavior description through the network traffic, looking at whether the malicious sample is going to react differently to the stimulation

received by the environment. Section 4 describes the behavior, the malicious actions, through the events logged by the Intrusion Prevention Systems, while in Section 5 we define behavior as the sequences of API calls that are characterizing each sample actions.

2.2 Defense Systems

This Section will go through the defense systems by using two white papers from Hutchins et al. [32, 3]. The first one presents the Cyber Kill Chain and the second one its applications. The papers go through how attacks work, their different phases and how they can be countered, depending on the defense systems involved.

According to these works, the attacks involve 7 phases:

1. Reconaissance: crawling and scanning on the Internet, looking for possible targets

2. Weaponization: preparation of the payload to use

3. Delivery: transmission of the weapon to the target (email attachments, websites etc etc)

4. Exploitation: exploiting the useful vulnerabilities on the target

5. Installation: installation of elements in the victim system allowing the attacker to enter the system in any moment

6. Command and Control: establishing a system of communications including a C&C server and its communication protocol

7. Actions on Objectives: the intruders can now act against their final targets, whether the target is the victim system itself or another machine in or outside the network

As we will explain, these attacks can be countered in different ways and using different tools. [3] considers 5 of the 6 actions suggested in [32] because "Destroy" does not have any defensive tool in any of the attack phases.

Phase	Detect	Deny	Disrupt	Degrade	Deceive
Reconnaissance	Web Analytics	Firewall ACL			
Weaponization	NIDS	NIPS			
Delivery	Vigilant User	Proxy Filter	In-line AV	Queueing	
Exploitation	HIDS	Patch	DEP		
Installation	HIDS	“chroot” jail	AV		
C&C	NIDS	Firewall ACL	NIPS	Tarpit	DNS redirect
Actions on Objectives	Audit log			Quality of Service	Honeypot

Table 2.1: Table explaining the possible actions and tools depending on the attacking phase in [3]

2.2.1 Intervention Actions and Areas

The attacking phases can be countered in different ways, depending on the defence and the position of this tool. The works from Hutchins et al. [32] identified 5 different counter actions (Table 2.1): Detect, Deny, Disrupt, Degrade, and Deceive.

“Detect” consists into discovering what is happening, proactively (before it happens), online (when it happens), offline (after it happened).

“Deny” is not allowing the action the attacker has to do as part of the specific phase of the chain.

“Disrupt” is interrupting the operation as, while it is happening, it looks suspicious and is blocked by the defensive tool.

“Degrade” consists in those actions that slow down and, as consequence, make less effective the attack operation.

“Deceive” consists in making the attacker or its tools believe they are reaching their goal while the environment they are interacting with is a fake one, controlled by the cybersecurity experts and//or their tools.

The table shows several possible defense tools. However, some of these tools can act against more than one attacker phase and, depending on the phase, they may operate more than one counter action. This is due to the fact that the defensive tools control many aspects of our Internet communications,

they are positioned in different intervention areas and some of these areas are involved in more than one attacking phase.

Company networks are often divided in many subnetworks, according to the kind of machines that are contained in the networks, the administrators implement different levels of defenses. Defenses can be placed at a network level (both on the perimeter and inside the network) as well as on host machines. Each defense can identify several types of attacks and does not prevent from the utilization of other defenses on the same area.

Domestic networks are much smaller and present less defensive tools. They often have a firewall implemented on the network router and, hopefully, host based antivirus software (in some cases it includes a host based firewall).

In recent years, the beginning of the smartphone era made mobiles much more similar to computers. However, the software distribution has a completely different mechanism: instead of going to the vendor website, there are markets managing and selling the apps. It opened a new area of intervention: the markets often have resources to do security checks and preventing malicious apps from being available to the public. However, some markets are much less effective than others in preventing the malicious apps distribution (in several cases even the main ones failed in this task). Computer software is moving in the same direction as people start relying on online stores to find software for their machines. This situation opens a new area of intervention where powerful resources can do complicated checks over, but at the same time it opens a new vector for the delivery attacking phase.

2.2.2 Defensive Tools

In Table 2.1 many defensive tools are mentioned as possible countermeasures to the attacking phases. Some of them operate on the connections done, like firewalls, IDS, and IPS (whether network or host based), the different types of AV analyze the files code and operations, while other parts monitor and log the events that are happening to recognize suspicious events and apply rules that would limit the attack efficiency.

Honeypots (in some applications they are called sandboxes, but they share the same kind of structure) are a completely different defense from the previous ones: they do not act on the actual traffic a part from a specific kind of attacks. It's a trap, where attackers can find what they believe is the aim of their attacks and be observed by the security administrators or, in some cases, the law enforcement.

2.3 Statistical Methods

In this section we go trough the different theories and methods used in the works presented in the dissertation. The first section explains all the theories, definitions, methodologies, and algorithms used in the causality works. The second section details the different algorithms used among when applying machine learning or deep learning to solve classification challenges.

2.3.1 Correlation and Causation

Correlation is the association of variables through statistical tests that can measure quantitatively the relationship among variables [33]. With respect to correlation, causation takes into account the nature of the relationship between variables. Causation is established by a priori knowledge or experimental ground that can assess direct cause of a certain factor (variable) over the variation in others [33].

There is an old saying, widely accepted, stating that "correlation does not imply causation." One of the empirical reasons supporting the concept are spurious correlations: correlations that are not due to any relationship between the two variables. One of the clearest examples of spurious correlations is probably the one between the number of people who drowned by falling into a pool in a year and the movies Nicolas Cage appeared in [34]. Correlation does not imply causation because, as defined earlier, the mere numbers do not take into account context. For instance, two variables may be correlated because their variations have a common cause, rather than one variable's variations are causing the other one to change [35].

2.3.1.1 Counterfactual Analysis Causality

The first time the concept of causality has been explored was in 1748, when David Hume defined the concept of causation and a first definition using a counterfactual example [36]. In Hume's work this concept is evaluated on the philosophical point of view, from this work many different definitions have been formulated, with minimal variations depending on the field of application, including statistical formalizations like the one we will use in this work from Lewis [37] who gave a statistical definition of causality through "counterfactual analysis". The idea is to make a minimal modification to the variables set and observe if the outcome changes or not; if it happens, it is possible to determine whether there is a dependency relation between the changing variable and the outcome.

Other causality models can be found in the literature. The most known is probably Pearl's causality model [38]; Pearl assesses causality by using graphs and Bayesian networks. Notwithstanding the importance of Pearl's work, we decided to rely on a simpler model, as the system did not require formalizations as complicated as Pearl's model.

2.3.2 Machine Learning and Deep Learning Algorithms

This Section explains the different algorithms used in the works that are part of this thesis. To introduce Machine Learning, its concepts and to explain K-Nearest Neighbors we will refer to "Pattern Recognition" from Duda et al. [1]. As the first version of this book has been written in *1973*, we will use different sources for Random Forests ([39] and [2]) and Tiresias' deep learning core engine ([40]).

2.3.2.1 What is Machine Learning?

Machine Learning (ML) is a family of mechanisms and algorithms that is used to classify data samples based on patterns. Such a broad definition allows to think to ML applications on several fields; many of those are not even barely related to Information Security as this is a fairly recent field and machine

learning algorithms have been invented and applied many years before the application to Information Security.

ML algorithms are divided in two main families: supervised and unsupervised. Even though the phases of ML procedures are applied to both families, the remaining part of this section will be dedicated to supervised algorithms as there is not unsupervised algorithms implementation in the work that is part of this thesis. The difference between the two families is that, while the supervised algorithms use a set of known samples (training set) to elaborate the decision model and a set of unknown samples (test set) to evaluate it, unsupervised learning operates clustering operations on the dataset without a priori knowledge of the dataset used to elaborate any model. It is then possible to check whether the decision taken by the algorithm on the test samples is correct or not.

ML is not only about algorithms applied to classify samples, but consists in several phases: sensing, segmentation, feature extraction, classification, and post-processing.

Sensing is the data collection phase: given a problem, we collect data that can be used to find out a solution to the problem.

Segmentation is related to processing data, evaluating if there are, eventually, incomplete records, and prepare the raw data to the features extraction.

Feature extraction is the crucial phase yielding "a representation that makes the job of the classifier trivial." [1] In other words the feature extractor is a system able to take data and transform it into a new representation, keeping, evaluating, measuring, andor transforming the (combination of) variables needed by the classifier. When the feature extraction phase is efficient, the classification phase is much easier; a classifier would have a much more complicated work if a set of features that is not good in representing the differences between samples of two different classes is used.

Classification is the phase where ML algorithms are applied. In this phase the set of features representing the samples of the dataset are used to take

decisions. The classification phase is evaluating whether a data sample could be part of a class or another one.

In the post processing phase it is possible to evaluate several aspects of the classification such as error rate or decisions confidence values. According to the constraints a system may have, while preserving validity characteristics, it is possible to modify parameters that affect the model decision making process and tune it to reach the efficiency system requirements.

The classification and post processing phases are intertwined by this opportunity of tuning the parameters. This operation is particularly delicate as it is important to not add any bias or incur into overfitting. To avoid any mistake of this kind, ML practitioners often use validation methods. Common validation methods that will be used in the following sections of the dissertation are m-fold Cross Validation and the use of a validation set.

M-fold cross validation consists in the division of the samples into m different groups. M-1 folds are used for the training set and the remaining one is used as test set. The folds are then rotated m times in order to have each fold as test set once. If post processing is used to tune parameters with this validation method, it is necessary to use two different groups of data, the first one for tuning and the second one for evaluating.

The validation set use is more straightforward in these phases' division: we first use the trained model to classify the validation set and we tune the model until we reach the efficiency results needed. Once the parameters set is defined, we classify the test set to evaluate the model.

2.3.2.2 Machine Learning Evaluation

Until this moment we decided to use the word efficiency while Accuracy may sound more sensible; however, Accuracy being one of the main evaluation parameters, it can be source of confusion. When evaluating an ML algorithm it is necessary to take into consideration its classification performances as well as other important characteristics that may clash with the requirements. The main ones can be considered to be (a) the RAM and CPU usage and (b) the

training and test time. ML systems often require high RAM and CPU usage, this may limit the usage of such systems into lightweight devices. We often use features selection algorithms to limit this issue; however, the usage of such algorithms may require more time. Time is another important constraint that may affect both training time and test time. Models' training often happens offline, in specialized powerful machines that are designed for such workload; however, some applications may have restrictions on training offline. Usually, the crucial time requirement is the classification time on testing samples: when deployed in real world we need the system to take the decision as fast as possible. ML algorithms often have a quick classification part (less than a second), however it is important to evaluate if it is quick enough: a system used for billions of analysis a day may not be fast enough if every single evaluation lasts a second. All these characteristics have to be taken into account for the evaluation as we cannot violate certain constraints while maintaining high classification performances.

Classification performances are evaluated through several parameters and are built on whether the evaluated test sample has been labeled correctly or not by the algorithm. When the decision is taken between two possible outcomes (classes), we often talk about a positive and a negative class, while when the decision is taken among different classes, we often talk about the relevant class (versus all the other ones). For instance, if we evaluate a malware versus benign classification system, we will use malware as the positive class and benign as negative class; when doing malware families classification, we will have each family as a separate class and calculate the classifier performances by taking each family separate from all the other ones recreating the dichotomy of the two classes case. When a classifier decision is wrong we use the word “False”, while when the decision is correct we use the word “True”. Summarized, there are four possible classifier outcomes:

- True Negative (TN): the classifier correctly decides for the negative class.
- False Negative (FN): the classifier wrongly decides for the negative class.

- True Positive (TP): the classifier correctly decides for the positive class.
- False Positive (FP): the classifier wrongly decides for the positive class.

According to these outcomes it is possible to calculate the evaluation metrics. The main ones used are Accuracy, Precision, Recall, and F1-Measure. Real-world implementations rely on constraints on these metrics (including the percentage of FP and FN) to understand whether the system is reliable or not.

Accuracy is defined as:

$$Acc = \frac{TP+TN}{TP+FP+TN+FN}$$

Precision is defined as:

$$Prec = \frac{TP}{TP+FP}$$

Recall is defined as:

$$Rec = \frac{TP}{TP+FN}$$

F1-Measure is the harmonic mean of Precision and Recall:

$$Acc = \frac{2 \cdot Prec \cdot Rec}{Prec + Rec}$$

Accuracy and F1-Measure are overall metrics: the resulting number summarizes all the aspects of the classification; TP, TN, FP, FN, Precision, and Recall are metrics that are taking into account only certain aspects of the classifier decisions. It is often necessary to use an overall metric and information retrieved from the non overall metrics at the same time to thoroughly evaluate the classifier performances.

The Receiver Operating Characteristics (ROC) curve is another interesting metric of evaluation. Although we decided to not use them in this work, ROC curves are giving important insights on how a binary classifier works. The curve is plotted by evaluating the True Positive Rate (TPR) and the

False Positive Rate (FPR) on different settings. This metric is extremely useful to evaluate the operational point of systems that should be deployed in the wild. For instance, a company may want to deploy an anti-spam system and requires an FPR less than 0.01%. At the same time the TPR cannot fall below 95%. With a ROC curve it is possible to see which settings of a system respect such conditions and, therefore, whether that system could be chosen to be deployed.

2.3.2.3 K-Nearest Neighbors Classifier

The Nearest Neighbors (NN) classifier evaluates which k training samples are the closest to the evaluated test sample in the features space to assign the label to the test sample. For instance, imagine that you have a satellite picture of a field and you know the position of some sunflowers and some roses. You decide that you will assign the label "sunflower" or the label "rose" to the unknown flower according to a majority vote among the three closest known flowers. If among the three closest flowers there are at least two sunflowers, the unknown flower will be considered a sunflower, otherwise it will be considered a rose. This is an example of a 3-NN classifier.

More formally, we define as our training set $D^n = x_1, ..., x_n$ a set of n labeled samples and $x' \in D^n$ the closest labeled sample to a test point x. Then, the nearest neighbor rule to classify x is to assign it the label associated with x'.

k-NN may not necessarily apply Euclidean distance or majority vote to classify samples, however, it is the setting often used as default in its implementations.The k is a crucial parameter: lower levels of k result in a higher level of granularity in the decision, however they may be more subject to noise if there are outliers in the training set. Moreover it is best practice to always use odd numbers as k: if we implement in the previous example a 2-NN and I have a sunflower and a rose as the two closest flowers in the field, the classifier may take a random decision. In case we had a third kind (or more) of flower labeled in that field, it is still best practice to use an odd k as most of the areas in the field will still be populated by only two species.

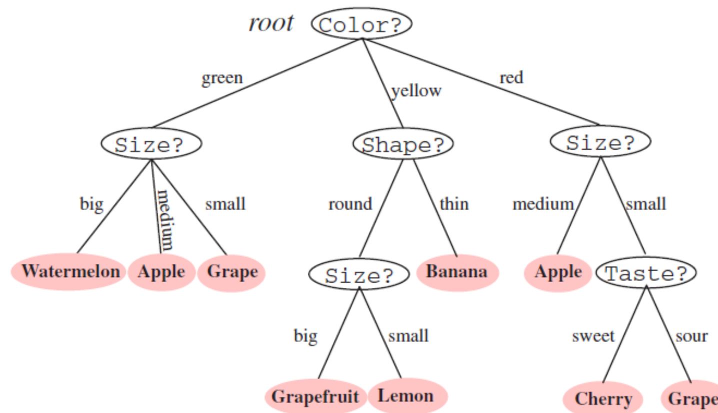


Figure 2.1: Decision tree example from [1].

2.3.2.4 Random Forests

Random Forests is a classifier using trees to take classification decisions. It is one of the newest ML techniques and it is proven to be particularly effective, without overfitting nor introducing biases. Breiman [39] is considered the author of this algorithm even though the final version is using previous work not only from this author [41, 42, 43, 44].

Decision Trees. Decision trees are the classifiers on which Random Forests is based. A tree (Figure 2.1) is a set of nodes corresponding to questions which answers are mutually exclusive but include all possible answers. The answers lead to new nodes with questions on different features until a label is assigned. Once the model is developed, the label assignment for an unknown sample is extremely fast: it consists only in a series of comparisons between the features values of the sample and those used to decide which path to follow.

CART. CART stands for "Classification And Regression Trees" [45] and is one of the most common methodologies to create a decision tree. As described in [1], CART poses 6 questions to build the tree:

1. Should the properties be restricted to binary-valued or allowed to be multivalued? That is, how many decision outcomes or splits will there be at a node?

2. Which property should be tested at a node?

3. When should a node be declared a leaf?

4. If the tree becomes "too large," how can it be made smaller and simpler, that is, pruned?

5. If a leaf node is impure, how should the category label be assigned?

6. How should missing data be handled?

When implementing Random Forests using off-the-shelf libraries like the Python Scikit-learn one[1], these are parameters that we can set and modify depending on the system requirements.

Random Forests takes a number of decision trees set by the user and trains them to take decisions according to the training samples. Moreover, each tree will be assigned a limited number of features to build its model. To increase its efficiency, Random Forests models add noise to estimate which features are the most important ones and find out the model that best separates the classes. Breiman [39] shows that, because of the Law of Large Numbers, Random Forests cannot overfit and the randomness mechanism allows high regression and classification accuracy.

2.3.2.5 Deep Learning

Deep learning algorithms are a specific family of algorithms able to extract the model from sequences of data, thus basically, avoiding the features extraction phase and the human decisions related to its design. They are based on Neural Networks as well as some ML algorithms. The name is due to the fact that Artificial Neural Networks are an interconnected set of nodes, like neurons in the brain. Neural networks are organized in layers (Figure 2.2 from [2]) that are fully connected: each node of a layer is connected to all the nodes of the following layer.

The first layer is the input layer and takes as input the sequences of data. The last layer is called output layer and the quantity of nodes depends strictly

[1] `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`

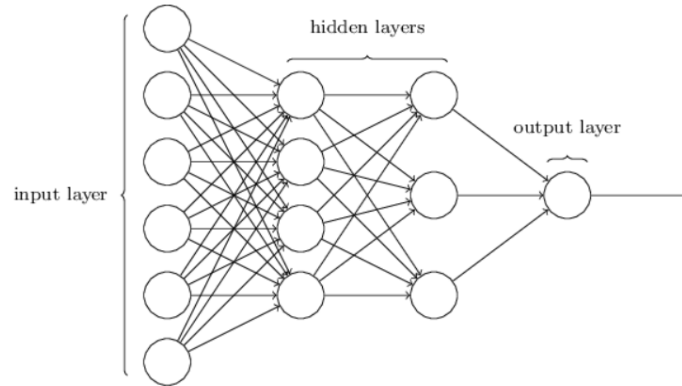


Figure 2.2: Neural network example from [2].

on the amount of possible outcomes. If the outcome is a decision between being a benign or a malicious sample in malware classification, only one neuron node is needed. Depending on whether its output is less or more than 0.5, we will assign a label rather than the other to the tested sample.

In the middle, between the input and the output layer, there are the hidden layers (two in Figure 2.2). The hidden layers are designed by deciding the number of layers and the number of nodes per layer, depending on the user's decisions.

All the layers apply different decision rules. During the training phase, the decisions made by each node are then compared with a target vector to understand which weights to attribute to different nodes.

In Tiresias, we relied on a specific type of Neural Networks: the Recurrent Neural Networks. The RNNs are using a feedback mechanism that adapts the weights while classifying. This mechanism has been found particularly useful when predicting series of events as it tunes the weights depending on the predicted events.

Among them, we used the implementation with Long Short Term Memory (LSTM) array of cells [40]. An LSTM cell has an input gate, a forget gate, and an output gate. The input gate is used to add new information in the cell (such as a new event in time series prediction), the forget gate is used to take out of the cell input data that is not useful to the cell prediction engine, and the output gate is used to send out a specific prediction. The activation of

the gates at the beginning of the classification phase depends on the values of the weights, decided by the algorithm during the training phase. However, the recurrent neural networks modify the weights in their cells while classifying, by using the back propagation mechanism. For instance, if the back propagation feedback is saying that the cells rely too much on short memory (the forget gate is often active, discarding the older inputs), the system might modify the weights to activate the forget gate less frequently. As consequence, the input sequences considered for the output decision will be longer.

2.4 Causality in Malware Traffic

Causality is a complicated phenomenon to assess. As explained in Section 2.3.1, it involves being able to assess a direction (which variable influences the other ones) to the relation between two elements. The first work we focused on has been establishing causal relationships between user actions and malicious activities by malware infecting the machines. To assess these relationships we analyzed the network traffic.

We assessed causality in these relationships by using counterfactual analysis as explained in [37]. The outcome is the presence of new connections operated by the malware and the only thing that is changed is the trigger. If, among the used triggers, there is one trigger that makes the malware generate new connections, then in that test, there is a condition of causality between the trigger and the malware network operations. As we will see later (Section 3.1.5), being in an experimental environment, there are issues related to the sensitivity of the system and to the noise due to unexpected network connections. These issues will be addressed through the use of Beta distributions for the Thompson sampling used in the Bayesian Inference tests.

Although we can use tests to infer correlation, we cannot infer causality because it does not only establishes the presence of a relation between two variables, but its direction. But what does this refers to, when talking about computer science or, more specific, malware research? According to Behi and

Nolan [46] "In circumstances where extraneous variables have been controlled, an experiment is said to have internal validity, and the causal relationship is proven." As the tests and the environment (Section 3.2.1.2) are fully controlled, we can assess causality where the statistical tests are giving evidence of correlation between our incoming variable (the operated test) and the outcome (the presence of triggered conversations).

Correlation and causality topics are widely explored in some fields of computer actions and network communications like services dependency, the most relevant system papers on this topic are Orion [47] and Rippler [48].

However, the topic we tackle is not the causality between services but between events and malware activity. Another important point the previous work analyzed is the malware samples similarity. The number of malware samples is dramatically increasing every day, but new samples are often using part of the code of previous ones and, as consequence, the behavior is similar too. Chakradeo et al. [49] analyzed the phenomenon on mobile malware while Bitshred [50] speeds up malware identification by taking advantage of malware triaging. Malware triaging is important with respect to our work because of the similarities between samples' network behavior (shown in Sigmal [51]); moreover, some samples need a trigger event to do certain operations, a problem tackled by Brumley et al. [52] by automatically analyzing the binaries, while we are looking to the network level of the phenomenon. This research analyzed the binaries similarities, but not the network operations caused by network events.

King et al. assessed causality aspects in malware behavior in an old paper [53] by studying causality in the attacks of the worm Slapper. With regard to this work, the differences are in the approach (i.e. we only look for network communications) and how causality is assessed (i.e. we apply statistical tests).

A more recent work, Zhang et al. [54], is more similar to ours because it looks to malware communications, but limiting itself to HTTP and DNS events to detect malicious triggered events; they assessed causality because

of the detection of the malicious events, but without any test as evidence. Moreover the mentioned work used its own proof-of-concept malicious samples that were written by the researchers to act in a predefined way while a real sample may act differently every time depending on many factors that can affect its reactions to trigger events. The application of statistics as we did, by applying the Bayesian inference, allows a much more precise definition of the correlation between a trigger event and malware operations on the network. Causality can be assessed because the environment is fully controlled during the tests.

As explained previously, we applied the Bayesian inference [55] of the relations between each different test and a certain sequence of labels; thereafter, in order to assess the relation between a test and the sequence, random sampling [56] over the distributions has been used to see if one of them was characterizing more the output sequence. This approach is used in other fields with the same purpose because the randomized sampling applied to a posterior distribution includes in the evaluation the uncertainty due to the noise that may affect the experiments [57]. An example of the use of these methods is Chapelle and Li [58], where the authors used a procedure similar to ours to evaluate the efficiency of Thompson Sampling mechanism with respect to more modern algorithms.

2.5 Browser Abuse

This part of the thesis focuses on establishing whether extensions installed in the browser are leaking or not its history. We used the causal relationships between the amount of history to leak and the dimensions of payloads exchanged between the virtual machine and suspicious IPs. To assess these relationships we analyzed the network traffic and used linear regression.

Like any other web application, browser extensions are third-party code. However, this software operates with elevated privileges and have access to APIs that allow access to all content within the browser. Extensions develop-

ers have been proven to request more permissions than the needed ones, effectively de-sensitivizing users. Heule et al. [59] considered the top 500 Chrome extensions and noticed that 71% use permissions that, if misused, can leak private information. They proposed, as solution, an extension design based on mandatory access control to protect user privacy.

Previous work has found that browser extensions are a relevant issue because of privacy-violations. It was shown that the quality checks made by official extension stores do not identify the malicious extensions. Researchers have manually analyzed extensions and reported the findings in a blog [60] and Starov et al. studied leaks basing their methodology on keyword search [61]. In contrast, Ex-Ray does not look for particular strings and is not affected by the extensions protocols.

IBEX [62] is a research framework that verifies access control and data flow policies of browser extensions through statistical tests. Developers have to author their extensions in high-level type safe languages; .NET and a JavaScript subset are supported. Policies are allowing for finer-grained control than contemporary permission systems; they are specified in Datalog.

Egele et al. [63] operated a dynamic taint analysis approach using the QEMU system emulator in order to detect spyware; they focused on Internet Explorer Browser Helper Objects (BHO). BHOs are relevant as they are classified as malicious whenever they leak sensitive information in their processes.

The first large-scale dynamic analysis of Chrome browser extensions have been done by using a system called Hulk [64]. The authors used what they defined as Honeypages. This technique creates web pages tailored to specific extensions to trigger malicious behaviors.

Authors that may want to monetize through their extensions. To do so, they may inject their own advertisements in the browser. Thomas et al. [65] found that 249 extensions on the Chrome Web Store were substituting the original ads with their own. The authors identified two drops in their mea-

surement of ad injection. They correlate to Chrome blocking side loading of extensions, and introduction of the single purpose rule to the Chrome store.

Websites use several third-party services enabling developers to quickly add functionalities. The downside is that user privacy is often threatened, because when websites are including remote source content, the user trust into a website is delegated. Nickiforakis et al. [66] studied this phenomenon and highlighted that the behavior is dramatically widespreaded. For instance, according to this study, Google Analytics was included in 68% of the top 10,000 websites.

There is a vast literature on third-party trackers on websites. Browsing on seemingly unrelated sites can be observed by third-party trackers and combined into a comprehensive browsing history. Mayer et al. [67] introduced a measurement platform called FourthParty, analyzing the privacy implications, technology, and policy perspectives involved by third-party tracking. Roesner et al. [68] developed defenses based on the client-side to identify malicious extensions and prevent third-party tracking. Recent work (e.g. [69]) has analyzed web tracking during the years by using the Internet Archive's Wayback Machine. In this work, the authors found that the tracking phenomenon has steadily increased since 1996 indicating that tracking on the web has never been as common and aggressive as it is now.

Even though we mainly focused on browsers, they are not the only platforms exploited to leak private data. In PiOS [70], Egele et al. measured applications from iOS app stores. The applications leaking private user data were not a large number, however, more than half of the analyzed apps leaked unique phone identifiers. This information is extremely sensitive and can be used by third parties to profile users. Similarly, AndroidLeaks [71] evaluates Android applications by using data-flow analysis to understand whether the apps leak private information; their dataset consisted in 2,342 applications. Lever et al. [72] highlight in their malware study how the analysis of network traffic is a key factor for early detection.

2.6 Prediction of Malicious Activities

This section illustrates the background and previous work related to [73]. In this project, we focused on the opportunity of predicting events in real world situations. Specifically, we designed TIRESIAS, a system able to predict the following event given the previous ones. We tested our system on real world data from Intrusion Prevention Systems (IPSs) which task was to monitor the different machines in their network.

To illustrate the complexity of keeping track of events across different machines, consider the real-world example in Figure 4.1. We show three endpoints undergoing a coordinated attack to Apache Web servers (see Section 4.1 for the detailed case study), where $\{e_0, ..., e_{13}\}$ are events involved in this attack and are highlighted in bold. This coordinated attack consists of three parts: (i) run reconnaissance tasks if port `80/tcp` (HTTP) is open (e.g., e_4 is default credential login, e_5 is Web server directory traversal), (ii) trigger a list of exploits against the Web application framework Struts (e.g., e_8 is an exploit relating to Apache Struts CVE-2017-12611, e_{11} is an attempt to use Apache Struts CVE-2017-5638, e_{13} tries to exploit Apache Struts CVE-2017-9805, etc.) and (iii) execute a list of exploits against other possible applications running on the system (e.g., e_2 exploits Wordpress arbitrary file download, e_9 targets Joomla local file inclusion vulnerability, etc).

To solve such challenge it is necessary to develop a system that is able to extrapolate the useful events by analyzing the entire sequences of events. We based TIRESIAS on Recurrent Neural Networks as explained in Section 2.6.1 and tested the system over real world data as explained in the following paragraphs (Section 4.3).

2.6.1 Security Events Sequences and the Application of Deep Learning Algorithms

In this section, we broadly reviewed previous literature in both forecasting security events and deep learning, especially recurrent neural networks (RNNs), applications in security analytics.

2.6.1.1 Security Event Forecast

System-level security event forecast. Soska *et al.* [74] proposed a general approach to predict with high certainty if a given website will become malicious in the future. The core idea of this study is building a list of features that best characterize a website, including traffic statistics, filesystem structure, webpage structure & contents and statistics heuristic of dynamic features (e.g., contents). These features are later used to train an ensemble of C4.5 classifiers. This model is able to achieve operate with 66% true positives and 17% false positives given one-year data. Bilge *et al.* [75] proposed a system that analyzes binary appearance logs of machines to predict which machines are at risk of infection. The study extracts 89 features from individual machine's file appearance logs to produce machine profile, then leverages both supervised and semi-supervised methods to predict which machines are at risk. In terms of machine wise infection prediction, RiskTeller can predict 95% of the to-be-infected machines with only 5% false positives; regarding enterprise-wise infection prediction, Riskteller can, on average, achieve 61% TPR with 5% FPR.

Organization-level security event forecast. Liu *et al.* [76] explored the effectiveness of forecasting security incidents. This study collected 258 externally measurable features about an organization's network covering two main categories: mismanagement symptoms (e.g., misconfigured DNS) and malicious activities (e.g., spam, scanning activities originated from this organization's network). Based on the data, the study trained and tested a Random Forest classifier on these features, and are able to achieve with 90% True Positive

(TP) rate, 10% False Positive (FP) rate and an overall accuracy of 90% in forecasting security incidents. Liu *et al.* [77] carried out a follow-up study on externally observed malicious activities associated with network entities (e.g., spam, phishing, malicious attacks). It further proved that when viewed collectively, these malicious activities are stable indicators of the general cleanness of a network and can be leveraged to build predictive models (e.g., using SVM). The study extracts three features: intensity, duration, and frequency, from this activity data. It later trained a SVM model using these features and achieved reasonably good prediction performance over a forecasting window of a few months achieving 62% true positive rate with 20% false positive rate.

Cyber-level security event forecast. Sabottke *et al.* [78] conducted a quantitative and qualitative exploration of the vulnerability-related information disseminated on Twitter. Based on the analytical results, the study designed a Twitter-based exploit detector, building on top 4 categories of features (Twitter Text, Twitter Statistics, CVSS Information and Database Information), for early detection of real-world exploits. This classifier achieves precision and recall higher than 80% for predicting the existence of private proof-of-concept exploits when only the vulnerabilities disclosed in Microsoft products and by using Microsoft's Exploitability Index are considered.

2.6.1.2 Recurrent Neural Network Applications in Security Research

Binary Analysis. Shin *et al.* [79] leveraged recurrent neural networks (RNN) to identify functions (e.g., function boundaries, and general function identification) in binaries. For each training epoch, the RNN model is trained on *N* examples (an example refers to a fixed-length sequence of bytes). The authors used one-hot encoding to convert each byte in a given example into a 256-vector, and associated a function start/end indicator with each byte (i.e., a 256-vector). Once the model is trained, it effectively serves as a binary classifier and outputs a decision for that byte as to whether it begins a function or not. The authors consequently combine the predictions from each model using

simple heuristic rules to achieve aforementioned function identification tasks. It is claimed that this system is capable of halving the error rate on six out of eight benchmarks, and performs comparably on the remaining two. Chua *et al.* [80] presents EKLAVYA, a RNN-based engine to recover function types (e.g., identifying the number and primitive types of the arguments of a function) from x86/x64 machine code of a given function without prior knowledge of the compiler or the instruction set. On the condition that the boundaries of given functions are known, EKLAVYA developed two primary modules - instruction embedding module and argument recovery module - to recover argument counts and types from binaries. The instruction embedding module takes a stream of functions as input and outputs a 256-vector representation of each instructions. After the instructions are represented as vectors, argument recovery module uses these sequences of vectors as training data and trains four RNNs for four tasks relating to function types recovery. The authors reported accuracy of around 84% and 81% for function argument count and type recovery tasks respectively.

Anomaly Detection. Du *et al.* [81] proposed DeepLog, a deep neural network model utilizing Long Short-Term Memory (LSTM), to learn a system's log patterns (e.g., log key patterns and corresponding parameter value patterns) from normal execution. At its detection stage, DeepLog uses both the log key and parameter value anomaly detection models to identify abnormal log entries. Its workflow model provides semantic information for users to diagnose a detect anomaly. The author reported that DeepLog outperformed other existing log-based anomaly detection methods achieving a F-measure of 96% in HDFS data and a F-measure of 98% in OpenStack data.

Password Attack. Melicher *et al.* [82] used artificial neural networks to model text passwords' resistance to guessing attacks and explore how different architectures and training methods impact neural networks' guessing effectiveness. The authors demonstrated that neural networks guessed 70% of 4class8 (2,997 passwords that must contain all four character classes and be at least

eight characters) passwords by 10^{15} guesses, while the next best performing guessing method (Markov models) guesses 57%.

Malware Classification. Pascanu *et al.* [83] model malware API calls as a sequence and use a recurrent model trained to predict next API call, and use the hidden state of the model (that encodes the history of past events) as the fixed-length feature vector that is given to a separate classifier (logistic regression or MLP) to classify malware.

The closest work to this paper is DeepLog [81]. However, DeepLog focused on anomaly detection in regulated environment such as Hadoop and OpenStack with limited variety of events (e.g., 29 events in Hadoop environment and 40 events in OpenStack). In such a very specific log environment, DeepLog was able to use a small fraction of normal log entries to train and achieve good detection results. Our work aims at understanding multi-steps coordinated attacks in a noisy environment with a wide variety of events (i.e., 4,495 unique events in our dataset) and prediction in this setup is a far harder problem comparing to DeepLog. Additionally DeepLog considered an event *abnormal* if such event is *not* with top-g probabilities to appear next. Our work does not employ this relaxed prediction criteria and focuses on the accurate prediction of the upcoming security event (out of 4,495 possible events) for a given system.

2.7 Android Malware and Static Analysis

This section of the chapter focuses specifically on the exiting work on Android malware and static analysis as background that introduces the topics of Section 5.

Mobile Malware is a relatively new phenomenon. In fact, while new computer malware has been created since the 80s as explained earlier, mobile malware is associated to the rise of smartphones. The smartphones allowed users to do many different actions through their phones, storing and using sensitive information that attracted cybercriminals. We can date the first mobile malware samples in 2004, grouping the first mobile malware families in 2009,

as Maslennikov [84] shows. However, mobile malware samples were not more than about a thousand samples at the end of 2010 [84]. In the following report [85], Maslennikov shows how the number of samples sky-rocketed in 2011. In fact, "the total number of threats over just one year increased 6.4 times. In December 2011 alone we uncovered more new malicious programs targeting mobile devices than over the entire period between 2004 and 2010." This trend has reached even higher numbers. According to the McAfee 2018 Q1 report on mobile threats [86], in the third quarter of 2017, for the first time in history, more than 2.5 million new mobile malware samples have been created by cybercriminals. At the end of the quarter the total of mobile malware samples was over 20 millions while one year and a half earlier was still less than 10 millions.

2.7.1 Static Analysis, Markov Chains and Malware Classification

Over the past few years, Android security has attracted a wealth of work by the research community. In this section, we review (i) program analysis techniques focusing on general security properties of Android apps, and then (ii) systems that specifically target malware on Android, with a particular focus on the ones applying Markov models.

2.7.1.1 Program Analysis

Previous work on program analysis applied to Android security has used both static and dynamic analysis. With the former, the program's code is decompiled in order to extract features without actually running the program, usually employing tools such as Dare [87] to obtain Java bytecode. The latter involves real-time execution of the program, typically in an emulated or protected environment.

Static analysis techniques include work by Felt et al. [88], who analyze API calls to identify over-privileged apps, while Kirin [89] is a system that examines permissions requested by apps to perform a lightweight certifica-

tion, using a set of security rules that indicate whether or not the security configuration bundled with the app is safe. RiskRanker [90] aims to identify zero-day Android malware by assessing potential security risks caused by untrusted apps. It sifts through a large number of apps from Android markets and examines them to detect certain behaviors, such as encryption and dynamic code loading, which form malicious patterns and can be used to detect stealthy malware. Other methods, such as CHEX [91], use data flow analysis to automatically vet Android apps for vulnerabilities. Static analysis has also been applied to the detection of data leaks and malicious data flows from Android apps [92, 93, 94, 95].

DroidScope [96] and TaintDroid [97] monitor run-time app behavior in a protected environment to perform dynamic taint analysis. DroidScope performs dynamic taint analysis at the machine code level, while TaintDroid monitors how third-party apps access or manipulate users' personal data, aiming to detect sensitive data leaving the system. However, as it is unrealistic to deploy dynamic analysis techniques directly on users' devices, due to the overhead they introduce, these are typically used offline [98, 99, 100]. ParanoidAndroid [101] employs a virtual clone of the smartphone, running in parallel in the cloud and replaying activities of the device – however, even if minimal execution traces are actually sent to the cloud, this still takes a non-negligible toll on battery life.

Recently, hybrid systems like IntelliDroid [102] have also been proposed that use input generators, producing inputs specific to dynamic analysis tools. Other work combining static and dynamic analysis include [103, 104, 105, 106].

2.7.1.2 Android Malware Detection

A number of techniques have used *signatures* for Android malware detection. NetworkProfiler [107] generates network profiles for Android apps and extracts fingerprints based on such traces, while work by Canfora et al. [108] obtains resource-based metrics (CPU, memory, storage, network) to distinguish malware activity from benign one. Chen et al. [109] extract statistical features,

such as permissions and API calls, and extend their vectors to add dynamic behavior-based features. While their experiments show that their solution outperforms, in terms of accuracy, other antivirus systems, Chen et al. [109] indicate that the quality of their detection model critically depends on the availability of representative benign and malicious apps for training. Similarly, ScanMe Mobile [110] uses the Google Cloud Messaging Service (GCM) to perform static and dynamic analysis on apks found on the device's SD card.

The sequences of system calls have also been used to detect malware in both desktop and Android environments. Hofmeyr et al. [111] demonstrate that short sequences of system calls can be used as a signature to discriminate between normal and abnormal behavior of common UNIX programs. Signature-based methods, however, can be evaded using polymorphism and obfuscation, as well as by call re-ordering attacks [112], even though quantitative measures, such as similarity analysis, can be used to address some of these attacks [113]. MaMaDroid inherits the spirit of these approaches, proposing a statistical method to model app behavior that is more robust against evasion attempts.

In the Android context, Canfora et al. [114] use the sequences of three system calls (extracted from the execution traces of apps under analysis) to detect malware. This approach models specific malware families, aiming to identify additional samples belonging to such families. In contrast, MaMaDroid's goal is to detect previously-unseen malware, and we also show that our system can detect new malware families that even appear years after the system has been trained. In addition, using strict sequences of system or API calls can be easily evaded by malware authors who could add unnecessary calls to effectively evade detection. Conversely, MaMaDroid builds a behavioral model of an Android app, which makes it robust to this type of evasion.

Dynamic analysis has also been applied to detect Android malware by using predefined scripts of common inputs that will be performed when the device is running. However, this might be inadequate due to the low probabil-

ity of triggering malicious behavior, and can be side-stepped by knowledgeable adversaries, as suggested by Wong and Lie [102]. Other approaches include random fuzzing [115, 116] and concolic testing [117, 118]. Dynamic analysis can only detect malicious activities if the code exhibiting malicious behavior is actually running during the analysis. Moreover, according to Vidas and Christin [119], mobile malware authors often employ emulation or virtualization detection strategies to change malware behavior and eventually evade detection.

Aiming to complement static and dynamic analysis tools, machine learning techniques have also been applied to assist Android malware detection [120]. Droidmat [121] uses API call tracing and manifest files to learn features for malware detection, while Gascon et al. [122] rely on embedded call graphs. DroidMiner [123] studies the program logic of sensitive Android/-Java framework API functions and resources, and detects malicious behavior patterns. MAST [49] statically analyzes apps using features such as permissions, presence of native code, and intent filters and measures the correlation between multiple qualitative data. Crowdroid [124] relies on crowdsourcing to distinguish between malicious and benign apps by monitoring system calls. AppContext [125] models security-sensitive behavior, such as activation events or environmental attributes, and uses SVM to classify these behaviors, while RevealDroid [126] employs supervised learning and obfuscation-resilient methods targeting API usage and intent actions to identify their families.

Drebin [127] automatically deduces detection patterns and identifies malicious software directly on the device, performing a broad static analysis. This is achieved by gathering numerous features from the manifest file as well as the app's source code (API calls, network addresses, permissions). Malevolent behavior is reflected in patterns and combinations of extracted features from the static analysis: for instance, the existence of both SEND_SMS permission and the android.hardware.telephony component in an app might indicate an

attempt to send premium SMS messages, and this combination can eventually constitute a detection pattern.

In Section 5.3.5, we will introduce, and compare MaMaDroid against, DroidAPIMiner [4]. This system relies on the top-169 API calls that are used more frequently in the malware than in the benign set, along with data flow analysis on calls that are frequent in both benign and malicious apps, but occur up to 6% more in the latter. As shown in our evaluation, using the most common calls observed during training requires constant retraining, due to the evolution of both malware and the Android API. On the contrary, MaMaDroid can effectively model both benign and malicious Android apps, and perform an efficient classification on them. Compared to DroidAPIMiner, our approach is more resilient to changes in the Android framework than DroidAPIMiner, resulting in a less frequent need to re-train the classifier.

Overall, compared to state-of-the-art systems like Drebin [127] and DroidAPIMiner [4], MaMaDroid is more generic and robust as its statistical modeling does not depend on specific app characteristics, but can actually be run on any app created for any Android API level.

Finally, also related to MaMaDroid are Markov-chain based models for Android malware detection. Chen et al. [128] dynamically analyze system- and developer-defined actions from intent messages (used by app components to communicate with each other at runtime), and probabilistically estimate whether an app is performing benign or malicious actions at run time, but obtain low accuracy overall. Canfora et al. [129] use a Hidden Markov model (HMM) to identify malware samples belonging to previously observed malware families, whereas, MaMaDroid can detect previously unseen malware, not relying on specific malware families.

Chapter 3

Causality Assessment in Malware Activities Using Counterfactual Analysis

This is the first technical section of the dissertation, as mentioned, it refers to the works having causality as topic [25, 26, 28]. In this part we analyze behavior through the reactions of malicious samples to triggers: we simulate the human actions to determine which actions may trigger which malicious samples.

3.1 Causality in Malware Traffic

To determine the causal relationship we use counterfactual analysis from Lewis, as explained in Section 2.3.1. While Mariconti et al. [25] explained the methodology behind using counterfactual analysis, in Mariconti et al. [26] there is a first attempt of reproducing this methodology into a real application: studying whether it is possible to identify causality relationships between user triggers and malware actions by looking only at the network connections operated from the Virtual Machines to other machines. This section takes from the first of these papers to explain the methodology.

We want to infer the type of a malware sample by learning causality relations between user actions and the activity performed by the malware

sample. For this reason, we observe the network packets generated by infected Virtual Machines (VMs) and apply statistical tests to assess causality. In this section, we describe our approach in detail.

Each test regarding a trigger and a malware sample follows this procedure: we record the test's network activity and extract the conversations from the dump file to label them depending on what generated them. From the conversation labels we create a chain of labels every time; we repeat the test to apply Bayesian inference on the chains frequencies of labels assessing if there are relations between labels and tests.

3.1.1 Approach Formalization

In this section we explain and formalize our approach to the problem. In a nutshell, our approach takes into account a set of malicious samples and a set of triggers, and studies if the samples react to the user triggers. Assessing causality needs a complete procedure in which all the details are taken into account, therefore we formalized the approach to the experiments and the different tests. The formalization has to be general and scalable to allow reproducibility and different levels of analysis such as experiments with an arbitrary number of malware families and triggers. More formally, we define a set of malware samples $M_1, ..., M_i, ..., M_K$ and a set of possible trigger events $N_1, ..., N_j, ...N_L$. An experiment consists in running one of the samples M_i in the presence of each trigger N_j, one trigger at a time. This formalization is extremely scalable, in fact, the approach is valid when changing how many malware families or possible triggers we use. Moreover, this formalization can be applied to different malware analysis as the concepts of triggers and malware families are not context dependent.

3.1.2 Experimental Environment

In this Section we explain the principles on which we have built our environment, starting from the sandbox and arriving to all the other phases of

the methodological framework, showing the challenges and how we practically solved them.

3.1.3 Sandboxing Background

We set up a virtual environment in which different VMs are configured to run. The structure is similar to the one created by John et al. [130].

Figure 3.1 shows the configuration of the sandboxing environment that we needed to implement. The webserver manages the download of malware by the VMs and the additional content needed for the experiments (e.g. credentials for the gmail login). Our mailserver is a sinkhole that receives all the SMTP packets the VMs generate, the router redirects those packets. This design avoids our VMs from sending spam to the Internet. To allow the connectivity of the virtual network, a router implements rules of network address translation, redirection of SMTP packets, and bandwidth restrictions, that will mitigate denial of service attacks performed against public servers from the VMs. These are only some of the security measures that we applied by following the guidelines from Rossow et al. [131]. To avoid the detection of the virtual environment by the malware samples, we followed the suggestions of the Pafish tool[132]. Pafish is a tool that operates the same kind of checks malware samples run to understand whether they are in a simulated environment or not. By following the suggestions given by Pafish it is more likely that the malicious sample would not understand that it is a sandbox and not a real environment.

The sandbox has to be secure and efficient: the research environment must not be a threat to the Internet (security) and it should be able to run as many samples as possible without being identified and evaded by the malicious code (efficiency). Most of the suggestions about these two big issues have been taken by Rossow et al. [131] where the authors tried to explain good practices to set up secure experimental environments. Some of them have been already mentioned: the mail redirection (anti spam), the bandwidth restriction (DOS mitigation), and the use of an account without any real information for the

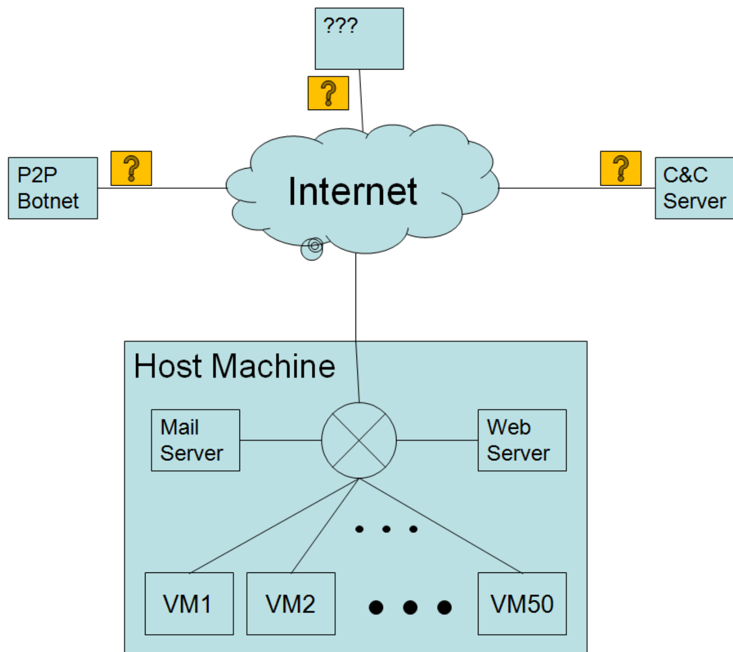


Figure 3.1: Sandbox infrastructure: the host machine simulates a network of VMs, a webserver managing the malware distribution, a mailserver to redirect possible spam campaigns from the infected VMs, and a router that allows the connection among the different internal machines and with the Internet world.

log in. Another important limitation has been in lifetime: the VMs are alive only for 30 minutes in order to avoid any complicate and long attack from the malware samples to their targets. Due to the sensitivity of this research and of the experimental environment, we applied and successfully obtained ethics approval from our institution, under project 6521/001.

The efficiency of the research environment is important as well: malware tries to understand if it is running in a virtual environment designed to analyze it, therefore it is important to deceive it. To this end, we set up the VMs to look as believable as possible, following, among the others, the instructions given by Pafish, a software designed to identify virtual machines from inside the virtualized environment. Similarly, to simulate user activity, we set up a script to open a Firefox browser window, and install a script that periodically moves the mouse cursor. With these instructions it has been possible to deceive a large quantity of the samples. Only a few of the Spambot samples have been able to detect the virtual environment and did not perform any activity. In

the discussion section we will reason on how it would be possible to improve the number of malware samples that run in our environment even further.

3.1.4 Sandbox Implementation Details

In Section 3.1.3 we explained the techniques related to the sandboxing approach.

We prepared three different virtual machines, one for each type of trigger under scrutiny: one that performs no actions, one that connects to a certain website (*https://amazon.com* in our case), and one that is connecting to Gmail and logging into a fake account. Figure 3.1 shows the configuration used for the VM that is not performing any user activity. Because of the elementary configuration of the webserver it was possible to run more than a machine at a time, with a maximum of 50. The other two kind of VMs and their webservers have a more complicated structure that does not allow more than one machine at the same time at the moment.

The VMs are started by scripts in the server, when the virtual environment is ready, it starts some short scripts to change some important characteristics: VirtualBox has specific registry keys and values that are detected by malware samples to evade the sandboxes analysis. After having deleted such values, for the same reasons, the server general script changes the MAC address. In fact, also the MAC address is predefined by VirtualBox; the MAC address is changed to an undetectable one through an algorithm that uses sequential numbers to track which machine receives which samples and instructions.

Ones the VM is set up, it contacts the webserver to download the instructions and the malicious sample. According to the instructions, the VM will execute the malware sample, simulate the human trigger and wait until the VM is shut off and deleted by the main server script. The main server script will be managing the creation of the VMs and, in parallel, the deletion of the ones that reached the maximum lifetime. All the network packets exchanged by the VMs are recorded by a tcpdump instance and, thanks to the modified

MAC address, filtered to divide into different pcap files the packets exchanged by each VM.

3.1.5 Extracting and Labeling Network Conversations

The network dump files collected during our experiments recorded information on the tests. Packets that have in common the tuple (sourceip, sourceport, destinationip, destinationport) are a conversation. The conversations is the key for labeling: we do not need to analyze each packet to identify the malware actions but we still have all the needed details.

For each test, we extract the conversations and then we label them: we determine the protocol and which domain is contacted by the VM for each conversation. Table 3.1 shows the label assigned to the contacted IP addresses depending on the test we are taking into account. For example, domains that are always contacted by the VM that runs trigger event 1, regardless of the specific malware sample that is being tested, are labeled as “event1 Trigger”. These conversations are independent from malware traffic and we will filter them out when we will be looking for traffic generated by malware as a possible response to a user trigger.

We define four possible conversations:

1. Common: those are the conversations the VM performs independently from malware samples or user trigger actions.

2. Pre-Trigger: those communications are performed by a malware sample independently from the user trigger action performed by the VM. The word pre indicates the independence from events happening after a specific user trigger is issued.

3. Trigger: those conversations are part of a user trigger activity, for example the connections generated by visiting a website.

4. Triggered: those conversations that did not appear in any of the previous labels. In this case, we consider the malware sample operating a connection as a consequence to a user trigger event.

We first have to learn which connections belong to the "Common" label: this is done by test Idle, no malware and no trigger. Those conversations are the ones the operative system has to operate to establish the communications and monitor which machine is in the same network as the one of the records. Those contacted domains and IP addresses will always be contacted in every test and will not be important for the analysis; for this reason the label is used only as a filter.

The "Trigger" ones are the labels assigned to the domains contacted by the VMs when no malware is infecting them: the domains that are not excluded by the "Common" label are those related to the trigger event. Using the same logic for "Pre-Trigger", we can label the domains contacted by the malware samples when the VM does not operate any particular action.

The "Triggered" label is given to the domains that are contacted during tests where there are a trigger operation and a malware: this label is given to the domains that are not already in the previous labels. The label "Triggered" will be given to those domains (if there is any) that are not already in "Trigger" and in "Pre-Trigger" labels.

	Doing nothing	**Trigger 1**	...	**Trigger L**
Not infected	Common	Trigger	...	Trigger
Malware type 1	Pre-Trigger	Triggered	...	Triggered
...	...	...	...	...
Other	Pre-Trigger	Triggered	...	Triggered

Table 3.1: Encoding of the labels. Domains contacted during tests are labeled following this table. Running VMs without any malware infecting them allows to find the conversations labeled as "Trigger", while running an infected VM in idle is how we assign to the conversations the label"Pre-Trigger". When the label to be assigned is "Triggered", it can be assigned only if that domain is not already in previous ones.

The labeling phase is the most delicate of this work: we continuously performed an accurate tuning of the translation of the IP addresses to the contacted domains because stealthy malware may be unobserved if they were using the same domains as legitimate traffic or too many "Triggered" labels

were assigned to contacted domains when the identification of the network is too precise. This is due to the presence of large IP spaces and the use of Content Delivery Networks.

3.1.6 Chains of Events

In Figure 3.1 we explained which test was assigning which label to a certain contacted domain. Apart from the test Idle, the tests without infection were giving a different trigger label to their contacted domains, while the domains contacted from tests without trigger are labeled with a pre-trigger label indicating that the samples contact those domains independently from the machine actions.

The tests where a malware sample runs in a VM that operates a trigger event label the domains that are not part of pre-trigger and trigger lists as triggered. This means that each of these tests may have different labels every time they are repeated, depending also on which samples and the current settings of the websites that are visited. The main concept of the work is in this point: every repetition of the test will create a chain of labels given the events (i.e. the connections) that the host machine will record. Each repetition will have a correspondent labels chain, therefore every test will have a number of times where a certain chain is the result while another one related to another chain.

Every test where there is an infection and an action by the VM can have two possible outcomes: "PreTrigger-Trigger-Triggered" in case of some new actions generated by the malware sample and "PreTrigger-Trigger" in case of a sample that does not react to the trigger. In the next section we explain how to study the correlation between the test type and the resulting sequences and why, in case of correlation, we can assess causality.

3.1.7 Statistical Analysis

We use statistical analysis to assess whether there is a connection between what happens in the VM and what is observed on the network and, as consequence, if there is a connection between the VM actions and the malware ones.

After the above labeling procedure has been carried out, the results consist of the frequencies at which the chains of labels have occurred in the tests. For each chain, our goal is to estimate the proportion of times it occurs during the test. This is essentially the task of estimating the proportion parameter θ of a Binomial(θ) distribution based on a sequence of binary observations where the observations are 1 if the sequence occurred, and 0 if it did not occur.

We estimate the proportion parameter using Bayesian inference to allow all uncertainty about its value to be captured. When performing Bayesian inference for the Binomial distribution, it is usual to use the conjugate Beta(α,β) distribution as it models the distribution of a binary outcome (in our case, a specific sequence of events versus all other ones). The α and β parameters in the prior distribution are chosen to take prior information into account, and we use the non-informative setting $\alpha = \beta = 0.5$. In this case, the posterior distribution is Beta($\alpha+N,\beta+M$) where N denotes the number of times the analyzed sequence occurs during the test, and M denotes the number of other sequences that occurred during repetitions of the test [133].

Once the posterior distribution has been obtained, it is possible to detect increases in the proportion parameter θ. This can be done by integrating the joint posterior distribution over the relevant region of space. We use an approach based on Thompson sampling [56] for this purpose. We sample a random value from each of the Beta distributions and note which distribution produced the highest observed value. We repeat this procedure many times and divide the counts of the highest values by the number of repetitions. After the normalization we have a probability of correlation of each test for the sequence analyzed and, as said in [134], because our environment is fully controlled and managed, it is possible to assess causality between the test with the highest

probability and the sequence. In case of this strong relation it is possible to affirm that the malware samples that are part of a certain family are triggered by a certain action in the real world and operate different actions on the network because of the trigger.

3.2 Causality Framework Application: Malware Network Traffic

The goal of our approach is to infer the typology of a malware sample by learning causality relations between user actions (e.g., logging into a website) and the activity performed by the malware sample. To this end, we observe the network activity generated by infected Virtual Machines (VMs) and we apply statistical tests to assess causality.

3.2.1 Application of the Methodology

An overview of our approach is displayed in Figure 3.2. As already mentioned, we run a malware sample in a VM in which we execute a simulated user activity, called trigger. We then record network traffic generated by the VM and separate it between traffic that is relative to the user trigger (e.g., the traffic related to shopping websites), traffic that is generated by the malware sample before the trigger happens, and traffic that is generated by the malware sample after the trigger happens. We then extract the occurrences frequency of a certain activity related to a specific trigger, and perform Bayesian inference to determine correlation between this activity and the corresponding trigger. As explained, our Bayesian inference process involves extracting Beta distributions from the data and performing Thompson sampling to assess the causation probabilities. Note that to reach the confidence required to reliably establish these values we typically repeat each test multiple times. Each single test regarding a malware sample and a triggering action follows a certain procedure; first it is recorded using a `tcpdump` instruction to register the whole network activity. We extract the conversations from the dump file and

label them depending on what generated the conversation (malware sample or trigger event) and why (if malware traffic independent from the trigger, the trigger, and malware traffic triggered by the VM). By using the labels attributed to the conversations we create a chain of labels per each repetition of the test and we apply Bayesian inference on the frequencies of the chains of labels to assess the relation between them and the tests. The decision of using a more complicated method such as Bayesian inference instead of a simpler chi-square test is because chi square only takes into account the proportion between quantities while Bayesian inference also considers the uncertainty in the measurements by using randomic sampling [135, 136].

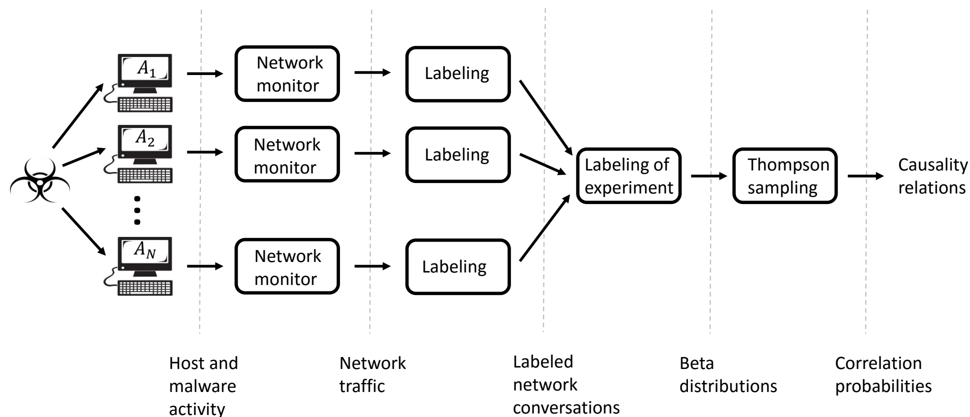


Figure 3.2: Overview of our approach.

3.2.1.1 Employed Dataset

The methodology we explain in Section 3.1.1 can be applied to assess relations between any user trigger and malware type or operation. We also present an effective case study based on malware network activity.

More precisely, we ran 20 Zeus samples [137] as Info-stealers, 10 Shopper-pro and 3 CloudGuard samples as Adware [138], and 20 samples of other families. The use of a limited quantity of samples is due to different reasons, the

most important being that we need active communication between the C&C server and the malware sample for our experiments. To collect the malware samples we periodically downloaded the most recent samples from VirusTotal.

Info stealers. These samples typically try to contact a certain number of C&C Servers to receive instructions about what to do in case of relevant data to steal (i.e., where to upload the stolen data). When relevant data is stolen, the malware communicates with different C&C Servers to upload the stolen information.

Adware. This type of malware operates a few connections to C&C servers to receive instructions about the hosts to contact when a website containing advertisements is visualized by the user. When the user navigates to a website containing advertisements, these are substituted by malicious ones. The sample's goal is immediately reached: the visualization of the malicious ads generates money to the malware operator.

Other. This group of malware samples were mainly Spambot samples. The Spambot samples are operating several different actions: they are contacting different C&C servers by using HTTP, HTTPS, and proprietary protocols; after these communication they start sending emails to victims by using the SMTP protocol. As we mentioned, our mailserver worked as a sinkhole for these emails.

3.2.1.2 Instantiation of the Experiments

We set up our experiment to take into account malware samples from three different types and study their relation to two user trigger events. The malware families that we study are *information stealer* malware (identified as *Inf* in the rest of the paper), *adware* (*Ad*), and *other* malware (*Ot*), where "other" includes malware samples that we typically do not expect to be triggered by user activity (e.g., spambots that send emails regardless of what the owner of the infected computer does). The trigger events that we used are the navigation to popular shopping websites (*Nav*) and the log in event into the Gmail webmail provider (*Log*). To correctly label network conversations at the next

step, we also need to run an infected VM in which no user trigger is executed. We also identify network traffic generated by the operating system regardless of user activity and the malware sample infecting the machine; to this end, we run a not infected VM. We call this test *Idle*. The combinations of malware types M_i and user triggers A_j used in this paper are summarized in Table 3.2.

	Doing nothing	**Navigation**	**Logging in**
Not infected	Idle	Nav	Log
Info Stealer	Inf	InfNav	InfLog
Adware	Ad	AdNav	AdLog
Other	Ot	OtNav	OtLog

Table 3.2: Summary of our test cases.

3.2.1.3 Extraction and Labeling Network Conversation

The network dump files collected during our experiments contain information on the IP addresses contacted by the VMs during each of the tests. We define a *conversation* as the exchange of packets that have in common the tuple formed as source IP address, source port (TCP or UDP), destination IP address, and destination port. Conversations are then used for labeling. This way can be agnostic to the network payloads themselves.

This phase aims at assigning a "label" to the network conversations observed by a certain experiment. The goal is to identify the conversations that compose the user trigger first, and we can then label accordingly the malware activity that happens before and after the trigger.

For each test, we extract the list of network conversations, resolve the DNS domain associated with the destination IP address, and proceed with labeling them. More specifically, we assign four different labels to network conversations:

Common: operating systems such as Linux and Windows perform network traffic as part of their behavior, regardless of any user activity or program running on the machine. Examples of this include automated software updates and synchronization with network shares. To avoid considering this traffic as

part of other labels, we run our VM without any malware sample or user trigger ("Idle" test in Table 3.2) and label any observed traffic as common, filtering it out when elaborating other labels.

Trigger: these conversations are those generated by the VM as part of a user trigger activity, for example the set of connections generated by visiting websites. We label conversations as Trigger if they are observed in the tests when the VM is not infected (marked as "Nav" and "Log" in Table 3.2) and were not marked as Common in the test "Idle."

Untriggered: these conversations are performed by a malware sample independently from the user trigger action. We use this label for conversations that are generated by the malware when no user trigger is present ("Inf," "Ad," and "Ot" tests in Table 3.2)

Triggered: these are the most important conversations for this work, because they are the ones that have the potential to present a correlation with the user trigger. We mark as Triggered any conversation that happens in a test in which a user trigger is happening, and that was not previously marked with any other label.

We first perform the "Idle" tests, followed by the tests in which only malware or user triggers are present, followed by the ones that combine a trigger and a malware sample. As we will explain in the next section, the variability of the set of IP addresses and domains contacted as part of different trigger activities and by different malware samples forced us to re-run our tests multiple times. Table 3.3 reports the number of performed runs.

Table 3.4 shows which test assigned which label to a contacted domain. Apart from the test Idle, the tests without infection were giving a different trigger label to their contacted domains, while the domains contacted from tests without trigger are Untriggered ones, indicating that the samples contact those domains independently from the machine action.

Test	Runs	Test	Runs	Test	Runs
Idle	42	Nav	108	Log	30
Inf	114	InfNav	40	InfLog	60
Ad	87	AdNav	73	AdLog	71
Ot	401	OtNav	159	OtLog	157

Table 3.3: Number of repetitions per test.

	Doing nothing	**Navigation**	**Logging in**
Not infected	Common	Navigation Trigger	Login Trigger
Info Stealer	Info Stealer Untriggered	Triggered	Triggered
Adware	Adware Untriggered	Triggered	Triggered
Other	Other Untriggered	Triggered	Triggered

Table 3.4: Labels encoding per each test.

3.2.1.4 Labeling and Chains Settings

The labeling phase is the most delicate: we continuously performed an accurate tuning of the translation of IP addresses to the contacted domains because stealthy malware may be undetected if it uses the same domains as legitimate traffic. On the other hand, using all the different subdomains may assign too many "Triggered" labels because the network identification was too fine grained.

As mentioned, we map IP addresses to domains when labeling network conversations. This works in most cases, because domains used by malware are not the ones used by legitimate applications. However, in some cases a domain can be used by both malware and legitimate traffic. One example of this is the use of Content Delivery Networks (CDNs). The biggest issue for our experiments were Amazon and Akamai servers: those address spaces are extremely wide and are used by a large variety of clients, from Amazon itself for advertisements on its website to malware samples hosting content to their

domains. It is not possible nor to simply assign `amazonaws.com` a specific label, nor to assign one to the exact IP. Therefore we found a good balance in using the first two octets of the IP addresses and dividing in eight groups the third one, giving the corresponding label to each of these subnetworks.

Another problem occurred when a malware sample was contacting many IP addresses on the same network but not all of them: it happened that the sample contacted different IP addresses in different test runs. A similar issue is given by advertisements used by Amazon: it asks to several addresses the required information and every time a different address can be contacted. For this reason we ran some of the tests more times than others, increasing the labeling reliability.

Our approach assigns a label to each network conversation, whether it happens independently of a user trigger (untriggered), it is part of the trigger itself, or it happens as a consequence of the user trigger (triggered). We run each experiment as a combination of user trigger and malware sample, however, it is composed of multiple activities that generate a multitude of network conversations. To assess whether the malware running inside a certain VM as part of an experiment was triggered or not by a user action, we must "label" the whole experiment as triggered or not. For example, if we observe a new connection after the VM has logged into a website we can mark the experiment as "triggered." Otherwise, if no new activity is generated after the user trigger, we can mark it as "untriggered."

To label an entire experiment, we look at the labels assigned to the single conversations as explained in the previous sections. If any of the conversations is marked as "triggered" then we label the entire experiment as such. Otherwise we label the experiment as untriggered.

3.2.2 Evaluation

In this section we evaluate our system. We present the labeling results on how many tests were triggered by which user triggers. We then describe how

we extracted Beta distributions from the experiments and how we assessed causality, providing evidence on the validity of our work.

3.2.2.1 Labeling Results

Test	Triggered percentage	Untriggered percentage
InfNav	55%	45%
AdNav	64.4%	35.6%
OtNav	22%	78%

Table 3.5: Labels for the tests in which the VM is navigating to *amazon.com*.

Test	Triggered percentage	Untriggered percentage
InfLog	92.6%	7.4%
AdLog	16.9%	83.1%
OtLog	29.2%	70.8%

Table 3.6: Percentages of the different labels for the tests with Log VMs.

In Tables 3.5 and 3.6 we show what fraction of tests presented the "Triggered" or "Untriggered" label. Table 3.5 shows quite high values of triggered Adware samples (64.4%) while info stealers present a lower value (55%). These tests use the VM that navigates to shopping websites, loading the related advertisements, and runs the malware sample. Because of the adware modus operandi, we expect many triggered activities from the adware samples, rather then from the other malware types. Most of the adware samples are triggered by the navigation user trigger, however, a relevant number of info stealer samples seems triggered as well; these labeling errors are ruled out by the statistical tests. In other words, the statistical tests are able to determine that there is no causal link between a user navigating to a website and activity by information stealing malware. On the other hand, it is able to assess that adware is likely triggered by navigation.

Table 3.6 shows the result of the experiments for the tests in which the user trigger is a login event on the Gmail website. There is a high fraction

of triggered Info stealers samples (92%), while only a small quantity of triggered Adware samples are triggered; the Other type reports 29% of its tests as "Triggered," but these triggers are ruled out by the statistical tests.

3.2.2.2 Beta Distributions

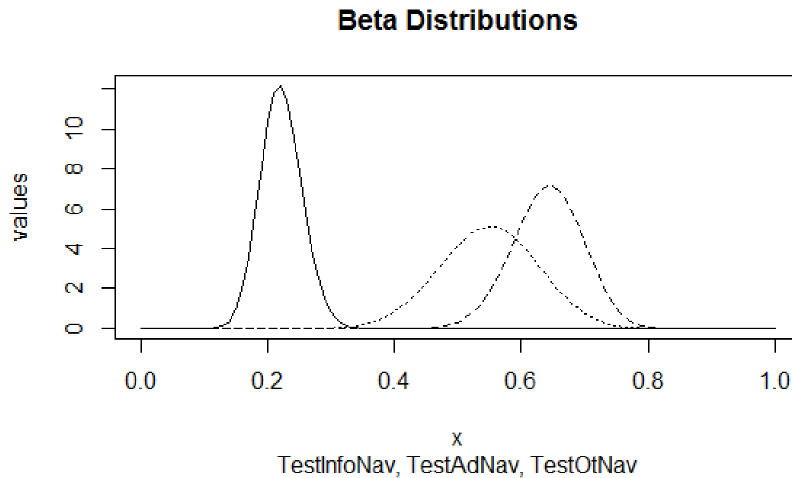


Figure 3.3: The Beta distributions related to the **Nav tests. The dotted line is the Beta distribution of InfNav tests, the dashed line represents the AdNav tests, and the full line represents the OtNav tests.

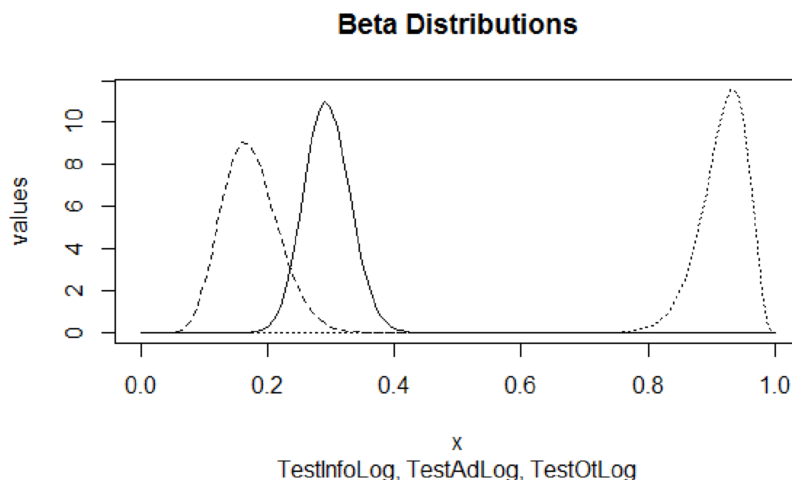


Figure 3.4: The Beta distributions related to the **Log tests. The dotted line is the Beta distribution of InfLog tests, the dashed one represents the AdLog tests, and the full one represents the OtLog tests.

To infer causality through the use of Bayesian inference, the first step is the creation of the Beta distributions from the results presented in the previous section. These results are used to draw the a posterior Beta distributions for each test as ($\beta(NumberOfTriggered + 0.5, NumberOfNotTriggered + 0.5)$. The Beta distributions that we used to model the variables are shown in Figure 3.3 and Figure 3.4. With the Test **Nav (all possible malware types, VM that is doing navigation) we observe a certain similarity between the curves in shape, height and position; these similarities are stronger observing only the Test AdNav and Test OtNav distributions. The distribution for the Test **Log tests show more differences between distributions; in fact the distributions are not close and the curve related to the InfLog tests (Info stealer) is much higher than the others. We can expect that the statistical tests will show a quite balanced situation between Tests **Nav while Tests **Log will give a preference in their results.

3.2.2.3 Statistical Evaluation of Causality and Experimental Validity

We ran Thompson sampling on the Beta distribution 200 times and calculated the average of the results over 10 repetitions; each result is a value between 0 and 1 that represents the probability of a causal relationship between a test and its label ("Triggered" or "Untriggered"). This probability takes into account the uncertainty given by mis-labeling due to the previously-explained issues therefore a big gap between the highest probability and the second one allows to assess causality.

In **Nav tests, Test AdNav is dominant. Test InfNav has 0.157 probability of being the cause of the triggered event among these tests while the probability of AdNav tests is 0.843. In OtNav tests, the triggered cases are not relevant. The mis-labeled OtNav tests did not affect the results of the statistical tests (probability equal to zero). The difference between the highest probability (the AdNav case) and the second one (InfNav) allows to indicate that the navigation user trigger caused the Adware network traffic. The statis-

tical tests for **Log tests have a very clear outcome: the triggered info stealer actions are caused by the login to Gmail user trigger because the probability given by the statistical test is 1, while there is no relation between the login into gmail and the actions of the other malware types (the probability related to the other tests is zero).

Validity of the experiments. For this work we empirically decided the number of samples to use, how many times the tests were repeated and how many observations with random sampling were necessary. We evaluated if the number of repetitions operated for each test can be considered sufficient. We repeated the statistical test using different portions of the operated runs (Figures 3.5 and 3.6). When the **Nav tests were operated, at least 80% of the repetitions was needed for results to achieve enough confidence as with using the full set. On the other hand, **Log tests were derived from beta distributions extremely different, in fact the tests were giving reliable and stable results already with a small percentage of the runs.

Similarly to the procedure used for the test repetitions, we empirically validate the number of observations used during the Thompson sampling phase: starting from two observations, arriving to 200, we observed that more than 50 observations are needed to have stable results on **Nav tests (Figure 3.7), while Figure 3.8 shows that even a minimum amount of observations is enough with **Log tests because the Beta distributions in this case are extremely different and indicate that InfLog tests have a clear correlation with the user trigger.

3.2.3 Discussion

In this section we will discuss the results of our framework in assessing the causality between a user-trigger and network activities performed by malware samples. As explained since the beginning, this has been the first attempt to apply this causality framework to the topic of malware analysis.

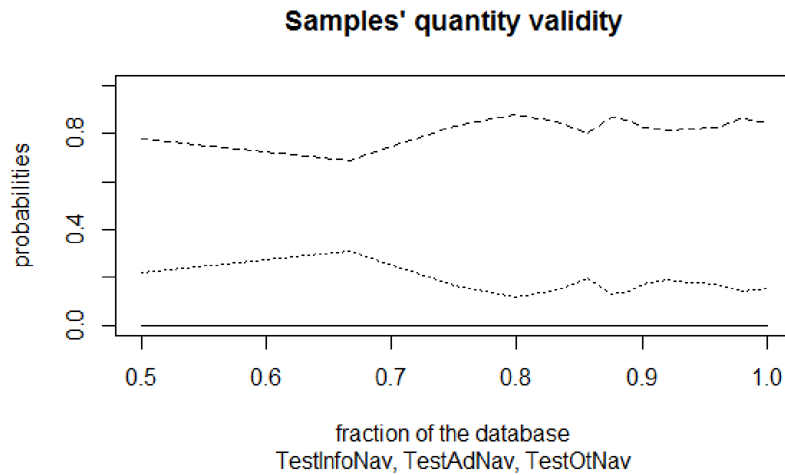


Figure 3.5: The causality probabilities of **Nav tests varying the used fraction of the dataset. The dotted line is related to the InfNav tests, the dashed line represents AdNav tests, and the full line represents OtNav tests. When the dataset is more than 80% the three lines become stable.

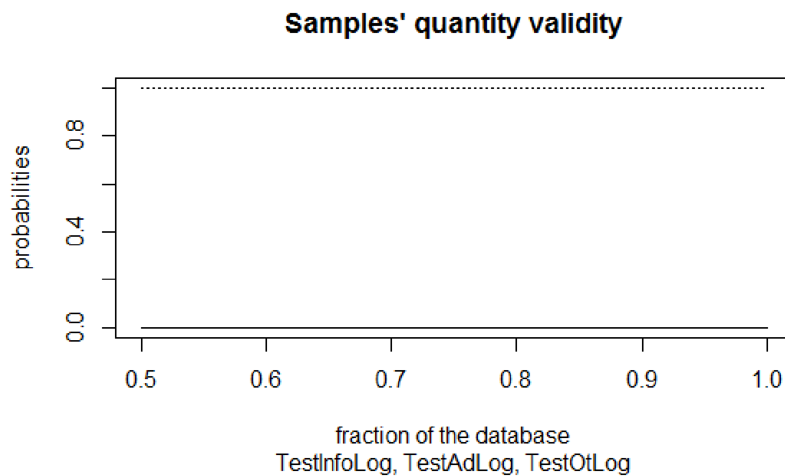


Figure 3.6: The causality probabilities of **Log tests varying the fraction of used dataset. The dotted line represents the InfLog tests, the dashed one represents AdLog tests, and the full line is related to the OtLog tests. In this case the observations maintain the same values regardless of the used fraction, because the Beta distributions are extremely different.

3.2.3.1 Labeling Results

The labeling phase presented many challenges due to the several domains contacted by some malware samples and by the VM navigating to shopping websites. Despite our best attempts, whitelisting websites confirmed to be benign

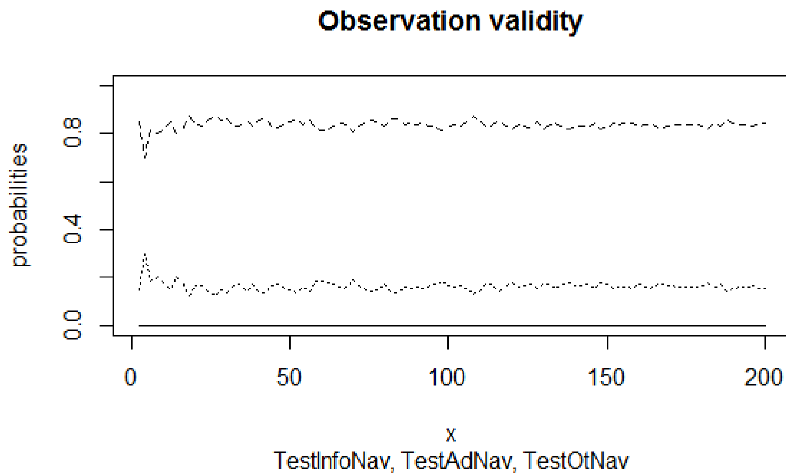


Figure 3.7: The causality probabilities of Tests**Nav varying the number of observations. The dotted line is related to Tests InfNav, the dashed line is Tests AdNav, and the full line is Tests OtNav. When the observations are more than 50, the three lines maintain the same values.

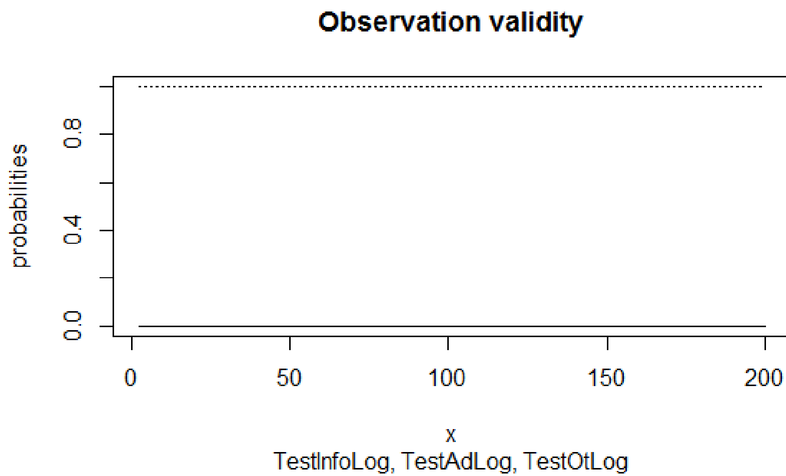


Figure 3.8: The causality probabilities of Tests**Log varying the number of observations. The dotted line is related to Tests InfLog, the dashed line is Tests AdLog, and the full line is Tests OtLog. In this case the observations maintain always the same values because the beta distributions are extremely different.

and the granularity tuning cannot completely remove mislabeling mistakes when large server domains (e.g. `amazonaws`) were contacted. For example, tests with Spambots are considered triggered because of two advertisements domains contacted only during OtNav and OtLog tests.

3.2.3.2 Results and Validity

We ran statistical tests to assess the relation between the user trigger and one of the tests. We expected a strong relation between Adware triggered traffic and the navigation trigger (Test AdNav) and between Info Stealer triggered tests and the login trigger (Test InfLog). Both relations were clearly assessed, even if some mislabeling affected the navigation case. To better understand if this noise influenced the experiments we observed that Test OtNav and Test OtLog do not present different results (Tables 3.5 and 3.6), while the tables show different behaviors with Test InfNav and Test InfLog or with Test AdNav and Test AdLog. These differences indicate that the malware samples are influenced by the user trigger events; in the case of the login, the significance of the result was not affected by the noise when we applied Bayesian Inference while the noise has been more effective in the navigation case, although our statistical analysis was still able to rule out these mislabeling.

The validity paragraphs (Section 3.2.2.3) show that the experiments are not biased by an incomplete dataset or a non sufficient number of observations. Because the validity criteria is respected we can argue that both the statistical tests indicated a causal dependency between the navigation user trigger and adware as well as between the login trigger and information stealing malware.

This work can be improved into a fully-fledged detection system. Malware samples could be run against different user triggers and an alarm could be raised when a type of malware that is considered of particular risk will be generated (e.g., an information stealing malware sample that has the capability of damaging to the company).

By adding a detection system, this methodology can be very effective in practical and real situations where the system should do limited kind of connections (as in delicate and with high security systems): the detection of the anomalous conversation would be immediately correlated to a trigger event and raise the alarm, a few minutes of sandboxing operations would then confirm it while the network administrator is already operating on the issue.

3.2.4 Limitations

The actions that can be detected by the presented system are a large variety and, because the system is content agnostic, this approach may also detect attacks through covert channels. The system is limited in detecting those samples that contact always the same C&C server during different phases of the attack: an Info Stealer sample that communicates the credentials to the same C&C server used in the first phase would not result as Triggered and can be misclassified in a detection system based on this work. At this stage, we cannot use unknown samples because we cannot infer causality through unknown samples; with the development of a detection system based on the causality inference it will be possible to use unknown malware samples.

In its current form, the framework does not take into account information on *when* a network flow appears within a test, but only if it appears or not. Mislabeling could be reduced in the future by using this information.

3.3 Causality Framework Application: Browser History Leakage

The browser has become the primary interface for interactions with the Internet, from writing emails, to listening to music, to online banking. The shift of applications from the desktop to the Web has made the browser the de-facto operating system. To augment this experience browsers offer a powerful interface to access and modify websites. Functionality allows for modification of HTTP requests and responses, injecting content to websites, or executing programs as a background activity. This allows for extensions that manage passwords, remove ads, or store bookmarks in the cloud.

The downside of this powerful interface is that malicious actions at the extension level can lead to problems across all online activities for a user. Extensions can be considered as the “most dangerous code in the browser” [59]. Previous research found extensions to inject or replace ads [64, 139, 140], causing monetary damage to content creators and, in turn, consumers. To detect

privacy-invasive extensions, previous work used dynamic taint analysis to find spyware in Internet Explorer Browser Helper Objects (BHOs) [63]. With previous research in mind, browser vendors can work to restrict malicious extensions.

Google Chrome is considered the state of the art in secure browsing. Chrome extensions can only be installed through a centralized store, and before being admitted they have to pass a review process. Users are prompted for permissions that an extension requests, and can use that information to decide whether they want to install the extension or not. Furthermore, if an extension is considered malicious after admission to the store, it can be remotely removed from clients. With all these security features in mind, privacy in Chrome extensions is still an issue.

To study the topic, we developed the first unsupervised system to detect history stealing browser extensions based on network traffic alone that is also robust against obfuscation. We then quantified the magnitude of user data leakage and introduce a scoring system that is used to triage extensions. Prioritized extensions are manually vetted and the resulting labeled dataset is made available to the research community. In this phase we applied counterfactual analysis to determine whether it was possible to establish a causal relationship between the leakage of browser history from malicious extensions and the size of packages exchanged by the extensions with the remote servers they refer to. We created a machine learning approach to classify extensions that we use on API call traces generated by an instrumented browser. This approach reaches 96.43% F-Measure value and the Recall value is constantly over 99%.

3.3.1 Linear Regression and Causality Background

As for the previous background section, we assessed causality in these relationships by using counterfactual analysis as explained in [37]. The outcome is whether the payloads of the packets exchanged fit the linear regression when the amount of history to leak changes.

We evaluate every extension by installing it in browsers that have different amounts of visited websites in their history. We evaluate whether the volume of packets exchanged with potential C&C IPs follows a linear regression model dependent on the amount of history in the browser. If the data fits the model, there is a causality relationship between the amount of history in the browser and the communications operated by the installed extension. We can therefore deduce that the extension leaks the browser history.

The methodology fits the concepts explained in 2.3.1.1 about counterfactual analysis and, as we will explain in Section 3.3, it follows the controlled experiment conditions [134] mentioned earlier.

To evaluate causality relations between the history on the browser and the network traffic size using counterfactual analysis [37], we apply linear regression in the first step of Ex-Ray. As described earlier, despite its rather simple basic idea, counterfactual analysis is a powerful model. In section 3.3 we will show that the absence of false positives among the extensions that were not leaking browser history simplified the procedure to determine whether there is a relation of causality between the amount of bytes exchanged on the network with specific destinations and being a browser extension leaking browser history or not. Linear regression [141] is a popular tool to establish this kind of relationship between two quantifiable variables, for instance, as an initial step before using machine learning for classification purposes [142], or as an embedded technique as in SVM [143].

3.3.2 The Environment

In this section we take into account the different factors that have to be considered when approaching this topic. We start by describing the infrastructure, we then analyze the types of trackers before finishing by explaining our threat model.

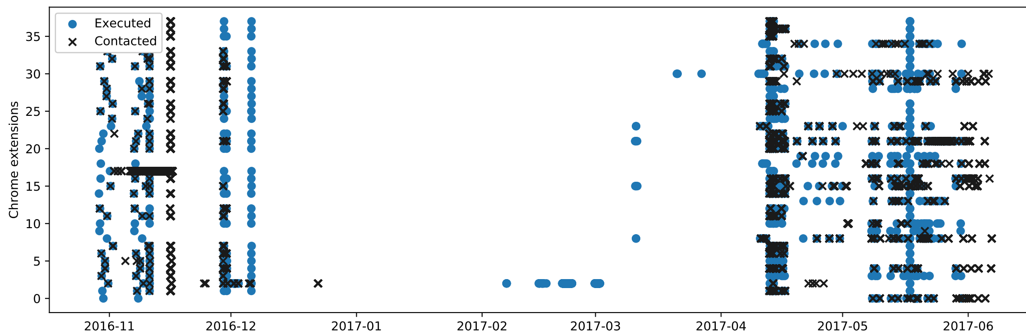


Figure 3.9: Extension execution with unique URLs vs. incoming connections to those URLs from the public Internet. These connections confirm that leaked browsing history is used by the receivers, often immediately upon execution.

3.3.2.1 HTTP URL Honeypot

To gain insight into the environment in which trackers operate, we configured a honeypot. To test whether leaked URLs are accessed after being received by trackers, we exercised extensions with domain names into which we encode their unique extension ID. While executing in our container, extensions only interact with local Web and DNS servers. However, we operate a web server on the public Internet to monitor client connections for such URLs. As these domains are used uniquely for our experiments, HTTP connections indicate leaks linkable to extensions. The connection and execution times are displayed in Figure 3.9. The confirmation that trackers are acting on leaked data motivated further steps in this work. After excluding VPN and proxy extensions, we received incoming connections from 38 extensions out of all Chrome extensions with more than 1,000 users.

3.3.2.2 Types of Trackers

Chrome offers a powerful interface to extensions, and while it can be used for useful tools it can also be misused to violate user privacy. There are multiple ways to collect and exfiltrate browsing history.

Much like trackers that are added to web pages by their authors, extensions can leak history by adding trackers to the body of web pages. An example of third-party tracking is the Facebook “Like” button. These can be

blocked by extensions such as Ghostery. A more robust solution is sending collected history data via requests of extension background scripts. Such requests are not subject to interception by other extensions, and cannot be blocked as tracker objects. Compared to tracking via inserting trackers into pages, better coverage can be achieved.

To acquire browsing data, extensions can intercept requests made by websites via the `chrome.webRequest` API, or poll tabs for the URL using `chrome.tabs`. For past browsing behavior, the `chrome.history` API can be used. Diverse options to collect data render finding a unified way to identify tracking extensions challenging.

3.3.2.3 Threat Model

Based on our honeypot results, we assume the following attacker model. In our scenario the attacker is the owner of, or someone who controls the content of, browser extensions. We assume many users will install these extensions with a cursory reading of the extension's description. While permissions can restrict the behavior of browser extensions, capturing and exfiltrating history can be performed with modest permissions that would not raise suspicion. For instance, the browsing history permission is categorized as *low alert* by Google.

The goal of our attacker is to indiscriminately capture URLs of pages visited by the user while the extension is executed. Furthermore, we assume the adversary collects data with the purpose of analysis or monetization. As the value of traffic patterns decreases over time, we assume the attacker to be inclined to leak sooner rather than later, which seems to be confirmed by our honeypot experiments. A successful attacker would decrease the user's privacy as compared to using a browser without the extension in question.

We exclude from our threat model extensions that openly require the sharing of browsing history as part of their functionality, such as VPNs or online blacklists. Also, we consider leaks purposeful and supposedly accidental as equal, as we cannot reason well about developer intent. As detecting and hiding malicious behavior is an arms race, we prefer to be conservative and as-

sume the attacker could escalate the sophistication of their evasion techniques in the future.

3.3.3 Ex-Ray Methodology

In this section we describe the design of the approach underlying Ex-Ray. The counterfactual analysis is an essential part of this methodology. To identify privacy-violating extensions, we exercise them in multiple stages, varying the amount of private data supplied to the browser, and in turn to the extension under test. Based on the type of extension, the traffic usage can change depending on the number of visited sites. However, the underlying assumption is that benign extension traffic should not be influenced by the size of the browsing history.

3.3.3.1 Overview

A top level view of Ex-Ray is shown in Figure 3.10. The three main components of the system are summarized as follows:

1) **Unsupervised learning**: We use counterfactual analysis to detect history stealing extensions based on network traffic. This component is fully unsupervised and, by definition, prone to misinterpretations.

2) **Triage-based analysis**: We manually vet the output of our unsupervised system, i.e., we verify which extensions are factually leaking and which are not. As the manual verification is costly, we rely on a scoring system that ranks extensions based on how likely they are to be leaking information to aid the process.

3) **Supervised learning**: We systematize the identification of suspicious extensions using supervised learning over the resulting labeled dataset. This component looks at the behavior of the extension and builds a model that detects history leaking (i.e., it looks at the API calls made by the browser extension when executed).

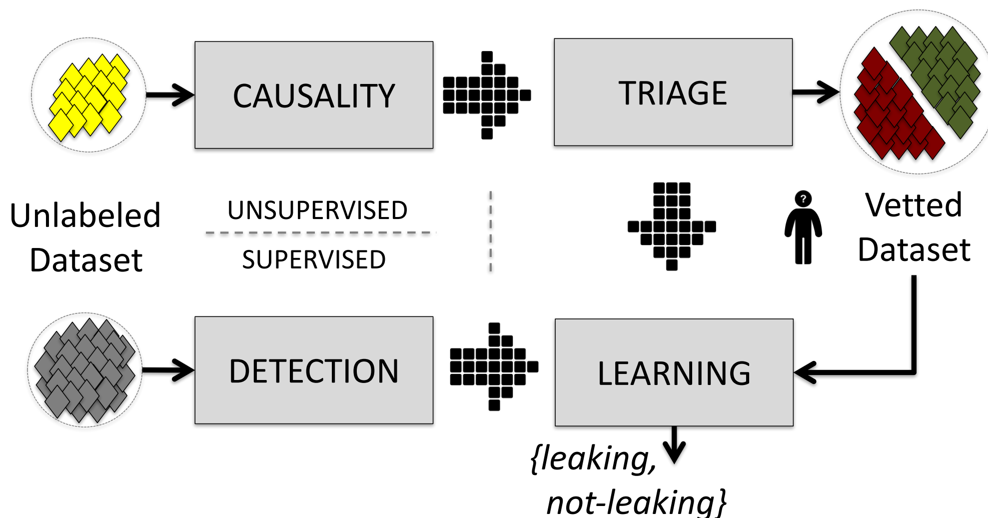


Figure 3.10: EX-RAY architectural overview. A classification system combines unsupervised and supervised methods. After triaging unsupervised results, a vetted dataset is used to classify extensions based on n-grams of API calls.

We see different types of tracking used in browser extensions. Some intercept requests and issue additional requests to trackers. Others transfer aggregated data periodically, while still others insert trackers into every visited page. An integral part of all trackers is transferring data to an external server—simply put, this crucial step is what enables trackers to track.

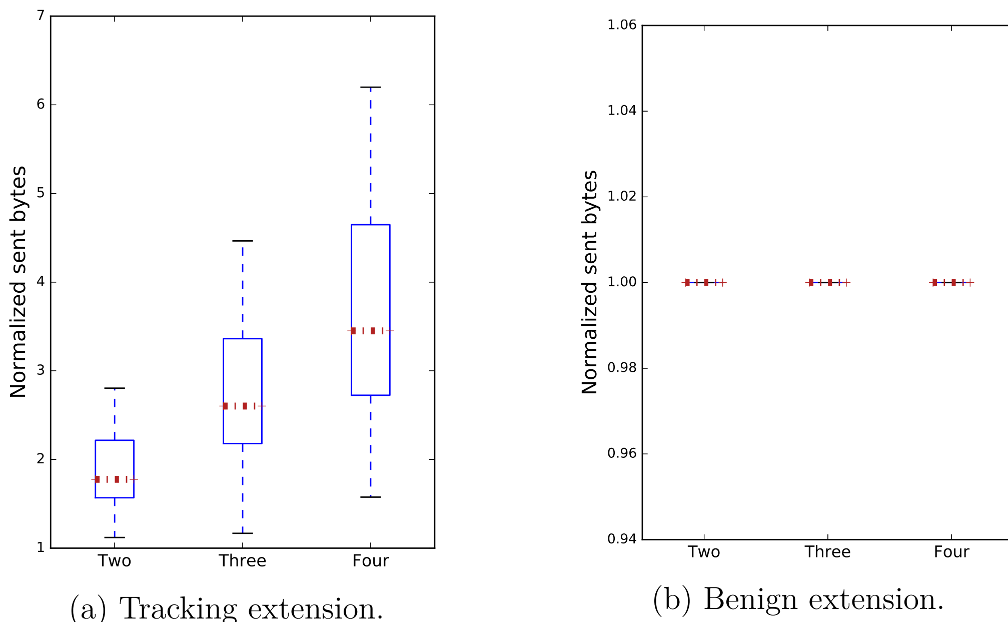
Our work focuses on indiscriminate tracking across all pages. To track, a history item (h_i) generated by the browser will be reported either in isolation or in aggregate. In either case, the size of history items affects network behavior. We argue that network data generated by an effective tracker, independently of protocol and whether plain, encrypted, or otherwise obfuscated has to grow as a function of history.

We execute extensions in multiple stages with increasing amounts of private information. Each h_i should contain less information than the following stage, $h_i < h_{i+1}$. We increase the size of h_i in each stage, extending the length of the testing URLs. For example, `example.com/example/index.html` in stage 0, and `example.com/example/<500characters>/index.html` in stage 10. The expected growth in traffic is h_Δ. This intuition is confirmed from Figures 3.11a and 3.11b where the boxplots clearly show that trackers usually

send more data when there is more history to leak while the amount of data is constant across the different stages for benign extensions.

For deterministic tracking, the traffic deltas of adjacent measurements should project an ascending slope. However, the browser history may be sent compressed in order to send as few bytes as possible and avoid the leak being visible as plain text in the payload. This operation would reduce the number of bytes sent while retaining the same amount of information (entropy). Per information theory, message entropy has an upper bound that cannot be exceeded. As a consequence, the size of compressed messages is lower-bounded as a function of the message entropy. For our experiments, we used compression tools (bzip2, 7zip, xz) to establish a practical lower bound of sent data for each stage as 289 Bytes, 6.9 KB, 14 KB and 30 KB.

Extensions that use trackers establish connections with each execution. Consequently, any group of hosts that results in less measurements than the number of executions will not be considered for further analysis. Examples of hosts that extensions only connect to occasionally are ads.



(a) Tracking extension.

(b) Benign extension.

Figure 3.11: Comparison in change of traffic between executions leaking history and benign extensions. Each bar displays the change of traffic sent relative to executions with increased history. Sent data projects an ascending slope based on size of history.

For this work purposes, we are going to focus on the unsupervised learning part, where counterfactual analysis is applied.

3.3.3.2 Application of Counterfactual Analysis

The goal of this phase is to model the way in which modifications to the browsing history influence observed network traffic. Figure 3.11 presented in the previous section shows that there is a monotonic increase between successive stages of privacy-violating extension tests. Extensions that, on the contrary, are privacy-respecting show no significant difference. One key finding observed during the analysis of the traffic behavior is that privacy-violating extensions might exhibit non-leaking behavior when connecting to certain domains. Thus, it is important to consider individual flows when building our model. Additionally, we also observed that in general variations exhibited in privacy-violating are well-fit by linear regression.

Thus, we use linear regression on each set of flows to estimate the optimal set of parameters that support the identification of history-leaking extensions. We aim to establish a causality relation between two variables: (i) the amount of raw data sent through the network, and (ii) the amount of history leaked to a given domain. For this, we rely on the counterfactual analysis model by Lewis [37], where:

> The model establishes that, in a fully controlled environment, if we have tests in which we change only one input variable, and we observe a change in the output, then the variable and the output are linked by a relation of *causality*.

In our case, the *input* variable is the amount of history, the *output* is the number of bytes sent in the different flows, and the *tests* are run with both goodware and malware. Our framework allows to evaluate this relationship by means of different statistical tests, such as Bayesian inference. This is ideal for situations were there is no deterministic relationship between the variables, such as in targeted advertisement tracking. Although our framework

is designed to model these scenarios, in practice, we observed that leaking extensions behave in a deterministic fashion.

In order to systematically identify the conditions under which the *causality* link is established, we run three steps. The first step is performed before applying linear regression, while the second and third steps are based on the linear regression parameters.

1. **Minimum Intercept**. While the extension might communicate to a domain in all given stages, the content transmitted may not contain a privacy leak. This step verifies whether the amount of data sent exceeds a certain threshold. This threshold is set based on the size of the history compressed as described in Section 3.3.3.1.

2. **Minimum Slope**. In this work we are primarily interested in extensions that actively track users. This type of extensions is expected to leak as much history data as possible from the user. This implies that the relationship between stages is expected to be linear and have a constant variance, modulo any sort of attempt at obfuscation. Based on this, we set a threshold to the slope in order to exclude all those extensions that do not fully meet these two criteria.

3. **Level of Confidence**. Depending on the extension, the regression model fitted might not always follow a strict linear model. We can choose to apply certain bounds (lower, upper, or both) from a fitted model to adjust the precision of the output. Choosing bounds that are very close to the fitted model will give a higher level of confidence in the decision. On the contrary, a very relaxed model will capture boundary cases at the cost of introducing false positives.

We define the term *flagging policy* as the set of parameters used for these checks. A *strict policy* is a policy in which parameters select a restricted area and flag less flows than a *relaxed policy* which flags many more flows as suspected of leaking browsing history.

The notion of confidence described above and the use of the different policies is precisely what motivates our triage system described next.

3.3.4 Ex-Ray Counterfactual Analysis Evaluation

In this section, we describe the evaluation of Ex-Ray counterfactual analysis. To allow comprehension we first describe the whole experimantal environment, but we will focus only on the results related to the components applying the causality framework core of this chapter.

3.3.4.1 Experimental Setting

An overview of our experimental setup is depicted in Figure 3.12.

Extension Containers. As part of our test environment, we created websites that allow scaling the size of a web client's browsing history without otherwise changing the behavior of the websites. We used local web and DNS servers so that the browser could connect to our website without sending information to the public Internet. For each execution, we started the experiment from an empty cache in a Docker container using an instrumented Chromium binary. We exercised each extension four times for five minutes each, capturing all generated network traffic. Capturing traffic on the container level provides a full picture of each extension's network interactions.

To reduce measurement noise, we blocked traffic to Google update services and CRLsets[1] via DNS configuration. We also disabled browser features such as SafeBrowsing and account synchronization.

Considering the maximum URL length of 2,083 characters, we increased the length of URLs by 500 characters between stages. Other than changing the length of URLs used, the pages served to the instrumented browsers did not change between stages. The maximum length of URLs generated by us is below 1,600, leaving sufficient space for trackers that submit URLs as GET arguments to do so. For each execution we open 20 pages; thus, if all URLs

[1] gvt1.com, redirector.gvt1.com, clients1.google.com, clients4.google.com

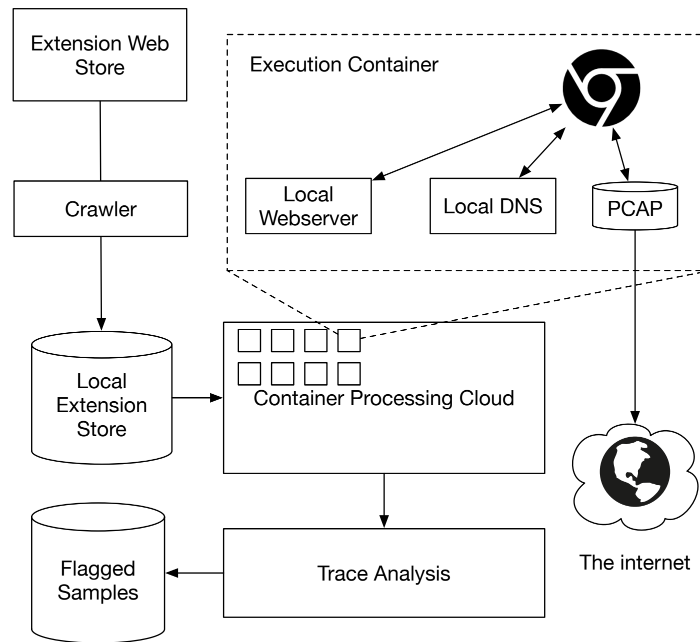


Figure 3.12: Ex-Ray extension execution overview.

were transmitted uncompressed we would expect an increase of 10 KB per stage. We stored DNS information to group IP traffic by hostname.

Extension Dataset. This work focuses on tracking data collected from browsing behavior that is sent to third parties. As opposed to previous work on history leaking browser extensions [61], the system aims to detect leaks regardless of how they are transmitted or collected. We target extensions that are tracking browser history either through background scripts or modification to pages. The main difference between these two approaches is that these trackers are only present on websites in the wild that opt-in to use them. Users are now familiar with tools that remove these trackers and the availability of such tools for free makes them popular. Conversely, tracking in extensions can cover all websites a user visits, and there is no opt-in mechanism. Furthermore, no tools are readily available that would warn a user of such behavior or block it.

Transferring the current host or URL is necessary for certain extensions functionalities – for example, to check against an online blacklist like adult content filters. However, we found that often extensions don't have

the need of transferring URLs, or could be expected by the extension's description, exposing all browsing habits of a user and creating a breach of privacy. Furthermore, developers often hide the specification of such functionality in an extension's description. Users are concerned about how their privacy is impacted [144, 145], without being aware of what a privacy policy is citesmith$_{h}alf_{2}$014.

We crawled the Chrome Web Store and downloaded extensions with 1,000 or more installations. For our analysis, we only consider extensions that can be loaded without crashing. Examples of extensions that could not be loaded are those with manifest files that cannot be parsed or referencing files that are missing from the extension packages. This left 10,691 extensions for Ex-Ray to analyze.

We mainly relied on two approaches to discover extensions:

- **Heuristic search**. We looked for suspicious hostnames, keywords in network traffic, and applied heuristics to traffic patterns. Through manual verification we confirmed 100 benign extensions and 53 privacy-violating extensions. The dataset contains different types of samples, including aggregate data collection and delivery over HTTP(S) and HTTP2.

- **Honeypot probe**. We registered extensions interacting with our honeypot. We verified 38 extensions connecting from the public Internet. Figure 3.9 shows a map of all incoming connections with respect to the time we exercised the extension with unique URLs in the history. Table 3.7 shows the most installed five malicious extensions with domains connecting to the honeypot. Connections often appear immediately after running the extension, but we also detected deferred crawls as well.

We excluded VPN and proxy extensions that redirect traffic through a remote address as these are not part of our threat model. The connecting clients performed no malicious activities we could identify in our log files. The

Extension Name	installations	Connecting from
Stylish - Custom themes	1,671,326	`*.bb.netbynet.ru`, `*.moscow.rt.ru`, `*.spb.ertelecom.ru`
Pop Up Blocker for Chrome	1,151,178	`*.aws.kontera.com`, `176.15.177.229`, `*.bb.netbynet.ru`
Desprotetor de Links	251,016	`*.aws.kontera.com`, `*.moscow.rt.ru`, `*.bb.netbynet.ru`
(Open Tabs)	97,204	`*.dnepro.net`, `109.166.71.185`, `*.k-telecom.org`
Similar Sites	45,053	`*.aws.kontera.com`, `*.moscow.rt.ru`, `*.netbynet.ru`

Table 3.7: Top five extensions connecting to our honeypot with highest installation numbers which are still available in the Chrome Web Store.

hostnames of clients that connected to us varied widely. The most popular one was `kontera.com` with 704 connections, followed by AWS endpoints. Interestingly, we received many requests from home broadband connections, such as `*.netbynet.ru`, often connecting only once. However, we connected four graphs of extensions that were contacted from the same hosts. The biggest graph connected eight extensions with two hosts. The other graphs connected five, two, and two extensions.

12 of these extensions were removed from the Chrome Web Store before our experiments concluded.

3.3.4.2 Ex-Ray Counterfactual Analysis Results

Tuning of the Unsupervised Component. The first step of Ex-Ray consists of applying linear regression for counterfactual analysis. The linear regression test flags flows if they respect the three parameters explained in Section 3.3.3.2. To find the best configuration of these parameters, it is necessary to evaluate the results on a labeled dataset. We used F-Measure as a comparison metric. The strictest policy checks for a minimum of 5 URLs leaked, a

2% minimum slope, and 90% accuracy. This policy results in an F-Measure of 96.9% and no false positives.

This result is showing that causality in this case is not dependent from probabilities. In fact, by having no false positives, it is not necessary to applied statistical methods as in the previous application because there is certainty that, when the linear regression is flagging an extension, there is leakage of browser history.

To obtain better results overall, in our final configuration we used two less strict configurations and flagged as suspicious all flows flagged in both engines. As the goal of the final configuration is to find as many malicious extensions as possible, knowing that the following phases of the system will refine the false positives cases, it is possible to relax the flagging constraints. This constraints relaxation is only possible because we have already shown the causal relationship; without that, it would have been necessary to go through the statistical models first. Both configurations check for a minimum of 2 URLs leaked and 2% minimum slope. However, there is a difference in the last check: while one used 90% accuracy in checking only the lower bound, the other one used 80% accuracy checking both the upper and lower bound. As such, the first and the last checks are less strict, but the F-Measure did not decrease even if a larger area of the feature space can be flagged. The system correctly flagged more flows as with the stricter configuration, but the flows belonged to the same extensions already flagged by the previous system.

Labeling Performance. Ex-Ray flagged 212 extensions out of 10,691 as history leaking using the linear regression on the traffic sent by the extensions. By checking manually, we noticed that not all the extensions flagged were history leaking. Out of 212 samples, 184 were leaking, 2 were goodware, and 26 were unclear. It has not been possible to determine if among those 26 extensions there were ones leaking or not. Therefore, to provide a conservative evaluation, we consider Ex-Ray to have 28 benign extensions wrongly identified as history leaking.

As mentioned earlier, detection systems can be prone to false negatives. To measure this for our system, we spot-checked a representative sample of extensions reported as benign. To establish baseline false negatives we scanned our pcap files for leaks and reimplemented another system used for brute-force searching extension traffic for obfuscated strings with a fixed set of algorithms [61]. This system flagged 367 extensions which we used for our dataset. The false negative samples we subjected to examination numbered 178. These results lead to a precision of 87%, a recall value equal to 50.13%, and an F1-Measure value equal to 63.66%. The overall accuracy value is 98.03%. These values are reached using only the first step of Ex-Ray that is a completely unsupervised algorithm. These results are further improved by the next phases of the system that are building on the results of the phase where counterfactual analysis is used.

3.3.5 Discussion and Limitations

The paper presenting Ex-Ray is using counterfactual analysis as a first step towards detection. In fact, inferring causality is used to determine whether the approach is working and after a first unsupervised flagging operation based on this technique, the system implements other elements to detect with high accuracy extensions leaking browser history.

The rationale behind our approach is intuitive: if an extension is leaking browser history, it has to transmit the leaked information. Following this intuition we set up the environment and decide to model the leakage as a linear relationship between the amount of history and size of exchanged packets. The presence of linear relationship corresponds to the Triggered flag in the previous application, while the absence corresponds to the Untriggered case. Instead of having different malicious families, in this case we have leaking and non-leaking extensions.

The experiments have shown how the system was extremely precise and no non-leaking extension was flagged. This result shows causal relationship without the use of the whole statistical framework. Achieving this result meant

tuning the different factors that were evaluating the linear regression by taking into consideration the noise due to the experimental setup, the fallacy of Internet communications, and the possible benign behaviors to avoid misclassifications. For detection purposes, after having determined causality, we have relaxed the constraints as possible misclassifications could have been ruled out by the following system steps.

Evasion of Linear Regression. A system based on linear regression is exploiting a rather simple habit of trying to communicate in the most immediate and simplest way. We took into evaluation whether the leakage was happening through encrypted messages as they would have still respected the linear relationship between size of history and size of the exchanged packets, however, we did not take into evaluation all other possible evasion methods. Breaking into several packets is not effective as the system looks at the aggregated quantities, but for instance the malicious extension could try to pad the shortest messages to mix the length of the packages or using non linear compression that would result in non linear increase of the size of the payloads. These are effective evasion examples for the causality step, however Ex-Ray is not a stand alone system, some of these behaviors would immediately be flagged by other security systems.

Sandboxing. Malware evasion is a well-explored area and is part of the arms race between attackers and defenders. Examples of this include fingerprinting analysis environments or creating more stealthy programs. While no ultimate solutions exist for these problems, Ex-Ray addresses tracking at a fundamental level.

Another approach would be to lay dormant and only leak at a later point in time. However, we have seen with our honeypot experiments that if leaks are utilized, this often happens immediately. Furthermore, there is an economic incentive on the part of attackers to obtain and monetize leaked history as quickly as possible before its value begins to degrade.

Chapter 4

Predicting Security Alarms due to Malicious Activities Using Deep Learning Algorithms

This chapter is focusing on the opportunity of predicting which malicious event may happen next. It is based on the CCS 2018 paper describing the system called Tiresias [73]. My main tasks in the project have been related to the design of the experimental setup, deciding which tests to apply, the benchmarks, and thoroughly analyze the results of all the different evaluations done. It has been important to understand why Tiresias was particularly efficient, which factors of the infrastructure were contributing and whether the requirements related to the possible system deployment were respected.

This chapter is analyzing the opportunity to study sequences of events flagged by security systems through a system we called TIRESIAS. The main difference with the other technical contributions is that, in this case, we are not recognizing the malicious actions, identifying and extrapolating the behaviors, but predicting the actions according to the behaviors observed in previous sequences. This approach differs from the previous ones because of its proactivity. In the previous section we have measured and studied behavior by running files in a safe environment, in the next one we will look at the code of malicious apps to detect malicious ones before they could be on the market.

Both technical contributions aim to prevent and detect malicious elements, but in this section we aim to predict what an active malicious actor is going to do to stop it while perfectly functioning in a real online environment.

Tiresias aims at predicting the actions that are part of attacks to computer systems and networks. The techniques used by adversaries to attack computer systems and networks have reached an unprecedented sophistication. Attackers use multiple steps to reach their targets [146, 147] and these steps are of heterogeneous nature, from sending spearphishing emails containing malicious attachments [148], to performing drive-by download attacks that exploit vulnerabilities in Web browsers [149, 150], to privilege escalation exploits [151]. After the compromise, miscreants can monetize their malware infections in a number of ways, from remotely controlling the infected computers to stealing sensitive information [152, 153] to encrypting the victim's data and holding it hostage [154, 155].

Traditionally, the computer security community has focused on *detecting* attacks by using a number of statistical techniques [149, 156, 157, 158, 159, 160]. While this is inherently an arms race, detection systems provide the foundation for network and system defense, and are therefore very important in the fight against network attacks. More recently, the attention of the community switched to *predicting* malicious events. Recent work focused on predicting whether a data breach would happen [76], whether hosts would get infected with malware [75], whether a vulnerability would start being exploited in the wild [78], and whether a website would be compromised in the future [74]. These approaches learn the attack history from previous events (e.g., historical compromise data) and use the acquired knowledge to predict future ones. Being able to predict whether an attack will happen or not can be useful in a number of ways. This can for example inform law enforcement on the next target that will be chosen by cybercriminals, enable cyber insurance underwriters to assess a company's future security posture, or assist website administrators to prioritize patching tasks.

$\mathbf{S_1}$ $e_{14} e_{15}$ **...** $\mathbf{e_{10}}$ e_{20} $\mathbf{e_{11} e_{8} e_{12} e_{4} e_{5}}$ **...** $\mathbf{e_{12} e_{11} e_{0} e_{3} e_{9}}$ e_{23} $\mathbf{e_{4} e_{9} e_{3} e_{4} e_{3} e_{9}}$ e_{23} $\mathbf{e_{3} e_{9}}$ $e_{19} e_{24} e_{25} e_{26}$ $\mathbf{e_{12} e_{13}}$

$\mathbf{S_2}$ $\mathbf{e_{4}}$ e_{27} $\mathbf{e_{10} e_{11} e_{12}}$ e_{28} $\mathbf{e_{5}}$ e_{21} $\mathbf{e_{7}}$ **...** $\mathbf{e_{4}}$ $e_{19} e_{30} e_{25} e_{24} e_{31}$ $\mathbf{e_{12}}$

$\mathbf{S_3}$ $\mathbf{e_{4}}$ e_{41} **...** $\mathbf{e_{5}}$ $e_{22} e_{21}$ $\mathbf{e_{7} e_{12}}$ **...** $\mathbf{e_{9} e_{3} e_{9} e_{3}}$ **...** $\mathbf{e_{6}}$ $e_{23} e_{19} e_{30} e_{25} e_{24}$ $\mathbf{e_{12}}$

Figure 4.1: Three endpoints undergoing a coordinated attack. $\{e_0, ..., e_{13}\}$ are events involved in the coordinated attack and highlighted in bold.

4.1 Motivation

We started explaining the motivations for this work in Section 2.6 and continued explaining it when introducing this chapter; however, we did not describe in details the challenges related to such problem.

The first challenge that we can immediately notice in Figure 4.1 is that even though those three endpoints are going through the same type of attack, there is not an obvious pattern in which a certain event e_i would follow or precede another event e_j given $e_i, e_j \in \{e_0, ..., e_{13}\}$. For example, e_{12} (Malicious OGNL Expression Upload) can be followed by e_4 (HTTP Apache Tomcat UTF-8 Dir Traversal) and e_{13} (Apache Struts CVE-2017-9805) in s_1, yet, it is followed by e_7 (Wordpress Arbitrary File Download) and e_{11} (Apache Struts CVE-2017-5638) in s_2.

The second challenge is that the endpoints may observe other security events not relating to the coordinated attack. For example, in s_3, we can observe a subsequence $\{e_4, e_{41}, ..., e_5, e_{22}, e_{21}, e_7\}$ in which e_5 is followed by a number of unrelated events including e_{41} (WifiCam Authentication Bypass) before reaching e_5. Note that the other noisy events are omitted for the sake of clarity. Between e_5 and e_7, there were two other noisy event e_{22} (Novell ZENWorks Asset Management) and e_{21} (ColdFusion Remote Code Exec).

More interestingly, some of these endpoints may potentially observe different attacks from various adversary groups happening at the same time. For example, we observe $\{e_9, e_{19}, e_{24}, e_{25}, e_{26}, e_{12}\}$ in s_1, $\{e_4, e_{19}, e_{30}, e_{25}, e_{24}, e_{31}, e_{12}\}$ in s_2, and $\{e_6, e_{23}, e_{19}, e_{30}, e_{25}, e_{24}, e_{12}\}$ in s_3. It is possible that e_{19} (SMB Validate Provider Callback CVE-2009-3103), e_{25} (SMB Double Pulsar Ping), and e_{24} (Microsoft SMB MS17-010 Disclosure Attempt) could be part of another

coordinated attack. Facing these challenges, it is desirable to have a predictive model that is able to understand noisy events, recognize multiple attacks given different contexts in a given endpoint, and correctly forecast the upcoming security event. This is a more complex and difficult task than detecting each malicious event passively.

Problem Formulation. We formalize our security event prediction problem as follows. A security event $e_j \in E$ is a timestamped observation recorded at timestamp j, where E denotes the set of all unique events and $|E|$ denotes the size of E. A security event sequence observed in an endpoint s_i is a sequence of events ordered by their observation time, $s_i = \{e_1^{(i)}, e_2^{(i)}, ..., e_n^{(i)}\}$. We define the to-be-predicted event as *target event*, denoted as e_{tgt}. Each target event e_{tgt} is associated with a number of already observed security events, denoted as l. The problem is to learn a sequence prediction function $f : \{e_1, e_2, ..., e_l\} \rightarrow e_{tgt}$ that accepts a *variable-length* input sequence $\{e_1, e_2, ..., e_l\}$ and predicts the target event e_{tgt} for a given system. Note that our problem definition is a significant departure from previous approaches that accept only fixed-length input sequences. We believe that a predictive system should be capable of understanding and making predictions given *variable-length* event sequences as the contexts, hence our problem definition is a better formulation inline with real world scenarios.

4.2 Methodology

In this section we describe the system architecture and the technical details behind Tiresias.

4.2.1 Architecture Overview

The architecture and workflow of Tiresias is depicted in Figure 4.2. Its operation consists of four phases: ❶ data collection and preprocessing, ❷ model training & validation, ❸ security event prediction, and ❹ prediction performance monitoring.

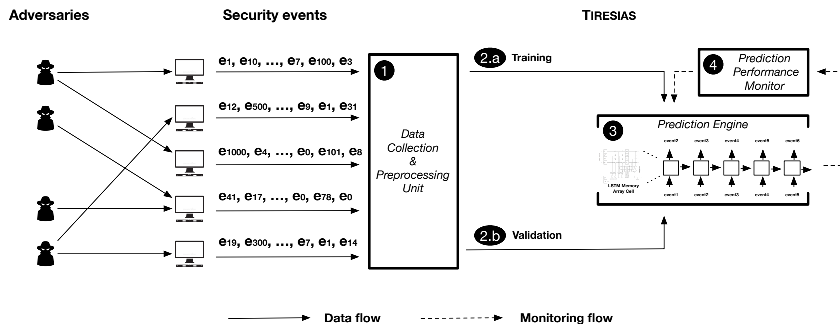


Figure 4.2: TIRESIAS collects security events from machines that have installed an intrusion protection product. The sequential events from these machines are collected, preprocessed and then used to build and validate TIRESIAS' predictive model. The optimal model is then used in operations and its performance is monitored to ensure steadily high prediction accuracy.

Data collection and preprocessing (❶). TIRESIAS takes as input a sequence of security events generated by endpoints (e.g., computers that installed a security program). The goal of the data collection and preprocessing module is to prepare both the training and validation data to build TIRESIAS' predictive model. TIRESIAS then consumes that raw security event data generated from millions of machines that send back their activity reports. The collection and preprocessing module reconstructs the security events observed on a given machine s_i as a sequence of events ordered by timestamps, in the format of $s_i = \{e_1^{(i)}, e_2^{(i)}, ..., e_n^{(i)}\}$. The output of the data collection and preprocessing module is $D = \{s_1, s_2, ..., s_m\}$ where m denotes the number of machines. Finally, we build the training data D_T and validation data D_V from D for the next stage, where $D_T \cap D_V = \emptyset$.

Model training and validation. The core of TIRESIAS consists of the training of a recurrent neural network with recurrent memory cells (❷.a, see Section 4.2.2 for technical details about recurrent memory cells). Essentially, TIRESIAS specifies a probability distribution of e_{w+1} possible events given historical observed events $\{e_1, ..., e_w\}$, where w refers to the rollback window size, by applying an affine transformation to the hidden layer followed by a $softmax$,

$$Pr(e_{w+1}|e_{1:w}) = \frac{\exp(h^w \cdot p^j + q^j)}{\sum_{j' \in E} \exp(h^w \cdot p^{j'} + q^{j'})} \quad (4.1)$$

where p^j is the j-th column of output embedding $P \in R^{m \times |V|}$ and q^j is a bias term. Given the training data D_T, TIRESIAS' training objective is therefore to minimize the negative log-likelihood $\mathscr{L}$ of all the event sequences:

$$\mathscr{L} = -\sum_{t=1}^{|D_T|} Pr(e_t|e_{1:t-1} : \theta) \quad (4.2)$$

We use the validation data D_V to verify if the parameters θ identified during the training phase can achieve reasonable prediction performance (❷.b). It is important to note that D_T and D_V come from different machines so as to verify the general prediction capability of TIRESIAS on the endpoints that are not part of the training data.

Security event prediction (❸). Once the model is trained, TIRESIAS takes the historical events $\{e_0, ..., e_i\}$ as the initial input (i.e., a *variable-length* input sequence inline with the real-world scenario) and predicts the probability distribution of e_{i+1} given E as $Pr[e_{i+1}|e_{0:i}] = \{e_1 : p_1,\ e_2 : p_2,\ ...,\ e_{|E|} : p_{|E|}\}$. Our strategy is to sort $Pr[e_{i+1}|e_{0:i}]$ and choose the event with the *maximum probabilistic score*. TIRESIAS then verifies with the actual event sequence whether e_{i+1} is the correct prediction. In case of a wrong prediction, TIRESIAS updates its contextual information accordingly. Section 4.5 provides a detailed case study of the security event prediction phase in a real-world scenario.

Prediction performance monitoring (❹). Finally, in an effort to maintain the prediction accuracy as high as possible, the prediction performance monitor tracks and reports the evolution of different metrics, such as the Precision, Recall, and F1 of the current model. It is possible to elaborate such metrics on TIRESIAS' implementation in the wild as it is immediately possible to see whether TIRESIAS predicted the right event. If the predictions precision is dropping below a certain threshold, the system would automatically understand that is necessary to retrain the model.

4.2.2 Recurrent Memory Array

Long short-term memory (LSTM) and variants such as gated recurrent units (GRU) are the most popular recurrent neural network models for sequential tasks, such as in character-level language modeling [161]. One common approach to deal with complex sequential data is using a stacked RNNs architecture. Essentially, stacking RNNs creates a multi-layer feedforward network at each time-step, i.e., the input to a layer being the output of the previous layer. In turn, stacking RNNs automatically creates different time scales at different levels, and therefore a temporal hierarchy [162]. This approach has been proven practical and achieving good accuracy in various cases, such as log prediction [81], binary function recognition [79], and function type recovery [80]. Nevertheless, despite the proven success of stacked RNNs, one complication incurred by such strategy is the lack of generalization to new data, e.g., stacking mechanisms chosen and tuned for current training data require vigorous evaluation and may not adapt well to the new data at run time [163]. Therefore, rather than stacking multiple layers of RNNs, it would be ideal to build more complex memory structures inside a RNN cell to retain temporal memories while keeping a single layer RNN network to maintain computational efficiency when training. To achieve both goals, we propose to leverage the recurrent memory array by Rocki [164]; this is doable by modifying LSTM architectures, while it is not available on GRU architectures.

Following the notation in Rocki [164], we can formally define the recurrent memory array as follows in Eq. 4.3.

$$
\begin{aligned}
f_k^t &= \sigma(W_{fk}x^t + U_{fk}h^{t-1} + b_{fk}) \\
i_k^t &= \sigma(W_{ik}x^t + U_{ik}h^{t-1} + b_{ik}) \\
o_k^t &= \sigma(W_{ok}x^t + U_{ok}h^{t-1} + b_{ok}) \\
\tilde{c}_k^t &= tanh(W_{ck}x^t + U_{ck}h^{t-1} + b_{ck}) \\
c_k^t &= f_k^t \odot c_k^{t-1} + i_k^t \odot \tilde{c}_k^t \\
h^t &= \sum_k o_k^t \odot tanh(c_k^t)
\end{aligned} \tag{4.3}
$$

where f denotes forget gates, i denotes inputs, o denotes outputs, c denotes cell states, and h denotes the hidden states. Here, $\odot$ represents element-wise multiplication. It is straightforward to notice that parameter k directly controls the number of cell memory vectors, which enables the recurrent memory array to build an array-like structure similar to the structure of the cerebellar cortex [164].

To deal with noisy sequential input data (Section 4.3) as observed in the real world, we follow the **stochastic** design outlined in [164] by treating initial output gate activations as inputs to a $softmax$ output distribution, sampling from this distribution, and selecting the most likely memory cell to activate (see Eq. 4.4).

$$
\begin{aligned}
p(i=k) &= \frac{e^{o_k^t}}{\sum_k o_k^t} \\
h^t &= o_i^t \odot tanh(c_i^t)
\end{aligned} \tag{4.4}
$$

Eq. 4.4 identifies the probability of a memory cell i to be activated and update h^t accordingly using this cell while the rest of memory cells are deactivated. Hence, instead of summarizing all cell memory (see Eq. 4.3), only one output is used in this stochastic design that is resilient to noisy input. We refer

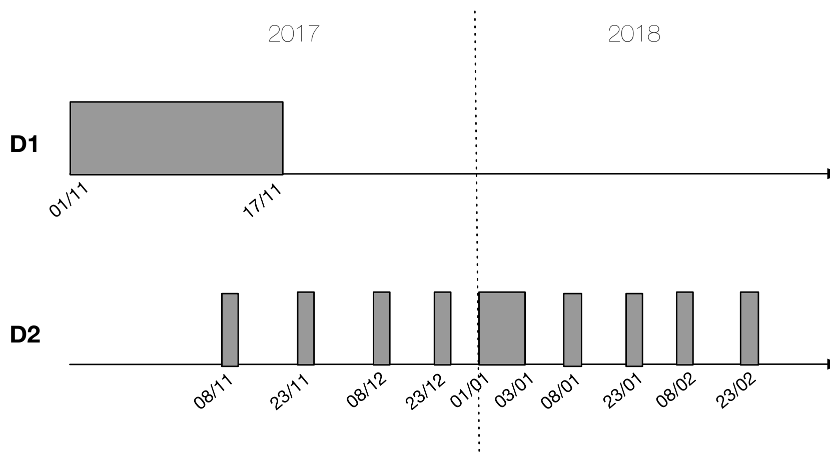


Figure 4.3: Summary of the security event datasets used in this paper.

interested readers to the work from rocki [164] for theoretical proofs and empirical comparison studies with the other state-of-the-art RNN architectures.

4.3 Employed Dataset

Tiresias is a generic system that can be used to predict security events on different protection systems. To evaluate its performance in this paper, we focus specifically on security event data collected from Symantec's intrusion prevention product. Symantec offers end users to explicitly opt in to its data sharing program to help improving its detection capabilities. To preserve the anonymity of users, client identifiers are anonymized and it is not possible to link the collected data back to the users that originated it. Meta-information associated with a security event is recorded when the product detects network-level or system-level activity that matches a pre-defined signature. From this data we extract the following information: anonymized machine ID, timestamp, security event ID, event description, system actions, and other information (e.g., file SHA2) if any. Note that we use the anonymized machine ID to reconstruct a series of security events detected in a given machine and discard it after the reconstruction process is done.

To thoroughly investigate the effectiveness, stability and reusability of Tiresias, we collected 27 days of data, summarized in Figure 4.3. We compile

two separate datasets. The first one, which we call **D1**, spans a period of 17 days in November 2017 (1 November – 17 November), and is composed of over 2.2 billion security events. We use the first five days (1 November – 5 November) of **D1** to validate our approach and for a comparison study against three baseline methods (see Section 4.4). We later use the first seven days (1 November – 7 November) of **D1** to build models with varied length of training period, and study the stability of our approach by evaluating the prediction accuracy for the rest of the 10 days of data (8 November – 17 November) from **D1**. We also compile another dataset, which we call **D2**. This dataset is composed of 1.2 billion security events collected on the 8th and 23rd day of each month between November 2017 and February 2018, and the first three days in January 2018. **D2** is used to understand whether the system retains its accuracy even in a longer term scale: training sets based on **D1** are months older than part of the data in **D2**. We use the first three days in January 2018 to build new models and compare them to the models built with data from **D1** (1 November - 7 November) and study their prediction performance with a focus on Tiresias' reusability (see Section 4.4.5). On average, we collect 131 million security events from 740k machines per day, roughly 176 security events per machine per day. In total, the monitored machines generated 4,495 unique security events over the 27 day observation period.

Data Limitations. It is important to note that the security event data is collected passively. That is, these security events are recorded only when corresponding signatures are triggered. Any events preemptively blocked by other security products cannot be observed. Additionally, any events that did not match the predefined signatures are also not observed. Hence the prediction model used in this paper can only predict the events observed by Symantec's intrusion prevention product. We discuss more details on the limitations underlying the data used by Tiresias in Section 4.6.

4.4 Evaluation

In this section we describe the experiments operated to evaluate TIRESIAS. We designed a number of experiments that allow us to answer the following research questions:

- What is TIRESIAS performance in identifying the upcoming security event (Section 4.4.2) and how does its performance compare to the baseline and state-of-the-art methods (Section 4.4.3)?
- How do variations in the model's training period affect the performance (Section 4.4.4)?
- Can we reuse a trained TIRESIAS model for a given period of time and when do we need to retrain the model (Section 4.4.5)?
- What is the influence of the long-term memory of Recurrent Neural Network models to achieve security event prediction (Section 4.4.6)?

4.4.1 Experimental Setup

Implementation. We implemented TIRESIAS in Python 2.7 and TensorFlow 1.4.1. Experimentally, we set the number of unrolling w to 20, the training batch size to 128, the number of memory array k (see Section 4.2.2) to 4 and the number of hidden LSTM Memory Array units to 128. We find these parameters offering the best prediction performance given our dataset. All experiments were conducted on a server with 4 TITAN X (Pascal) 12GB 1.5G GPUs with the CUDA 8.0 toolkit installed. All baseline methods are implemented in Python 2.7 and experimented on a server with a 2.67GHz Xeon CPU X5650 and 128GB of memory.

Evaluation setup. To form a concrete evaluation setup, for both TIRESIAS and other baseline methods experiments, we split the input data and use 80% for training, 10% for validation, and 10% for test. We strictly require that training, validation and test data to come from different machines so as to verify TIRESIAS' general prediction capability in the endpoints that are not part

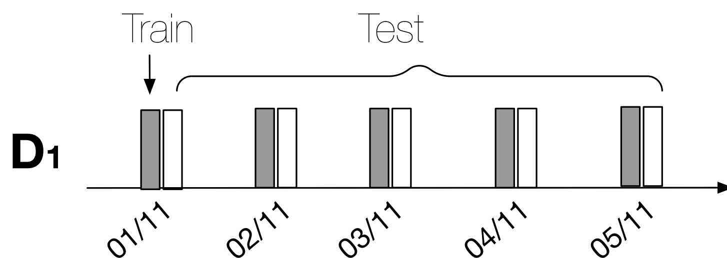


Figure 4.4: Experimental setup for TIRESIAS' prediction evaluation (Section 4.4.2) and comparison study with baseline methods (Section 4.4.3). The grey bars indicate data derived from machines used for training while the dotted bars indicate the data used for testing and coming from different machines with respect to the training data.

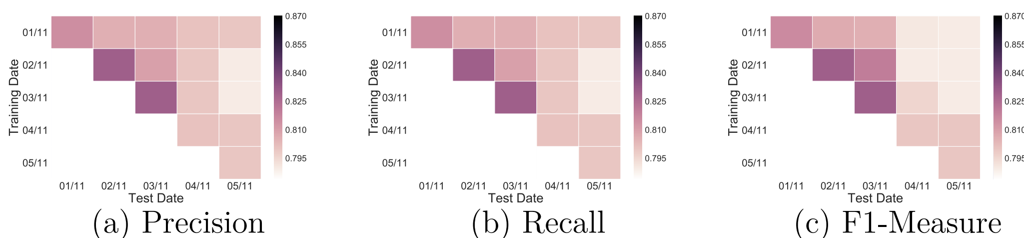
of the training data. Specifically, we train TIRESIAS for 100 epochs, validate model performance after *every* epoch and select the model that provides the best performance on validation data.

Evaluation metrics. We use the precision, recall, and F1 metrics to evaluate prediction results from the models. In our experimental setup, we calculate these metrics globally by counting the total true positives, false negatives and false positives. It is important to note that TIRESIAS accepts *variable-length* security event sequences. We specially hold out the *last* event as the prediction target e_{tgt} for evaluation purposes. Section 4.5 showcases how TIRESIAS can be leveraged to accomplish step-by-step prediction with a single event as the initial input.

4.4.2 Overall Prediction Results

In this section we evaluate the performance of our security event forecast model in predicting the exact upcoming event. This is a challenging task that a predictive system for security events aims at resolving due to the fact that there are 4,495 security events as possible candidates in our dataset (see Section 4.3) and an exact event should be correctly predicted.

Experiment setup. We use the experimental setup as illustrated in Figure 4.4 for TIRESIAS' performance evaluation. From **D1**, we train our predictive model using **one day** of data and evaluate TIRESIAS on both the same day



(a) Precision (b) Recall (c) F1-Measure

Figure 4.5: Precision, Recall, and F1-Measure of overall TIRESIAS's performance. TIRESIAS is trained using one day of data and evaluated on both the same day and the following days until 5 November 2017.

and the following days until 5 November 2017. For example, we train TIRESIAS using data from 2 November and evaluate its prediction performance from 2 November to 5 November.

Experiment results. Following our general experimental evaluation setup, we randomly select $14,396$ machines from the first days of November that are not part of the data used in the training set of the initial model. We focus on predicting the last event occurring on a machine given the sequence of previously-observed events. As shown in Figure 4.5, TIRESIAS is able to achieve over 80% precision, recall, and F1-measure in predicting the exact upcoming security event when evaluating on the same day test data. Figure 4.5 shows that, when training on one day, and testing on the same day and the following ones, the values of the Precision, Recall and F1 do not decrease dramatically. When it does, it decreases, in the worse case, of less than 0.05. The Figure also shows that Precision (Figure 4.5a) and Recall (Figure 4.5b) are well balanced and have very similar values and exactly the same scale (from 0.87 to 0.795). This result shows that TIRESIAS can offer good prediction results. There is no security event dominating in our dataset, which may lead to biased but better prediction performance. The top 3 events in our training data are: (i) Microsoft SMB MS17-010 Disclosure Attempt (19.8%), (ii) SMB Double Pulsar Ping (16.4%), and (iii) Unimplemented Trans2 Subcommand (16.1%). The top 3 events in our test data are ranked as follows: (i) Microsoft SMB MS17-010 Disclosure Attempt (9.85%), (ii) HTTP PE Download (6.3%), and (iii) DNS Lookup Failures (3.5%). Interestingly, the dominant events in

	Test Date (Evaluation Metric - Precision)				
Method	**01/Nov**	**02/Nov**	**03/Nov**	**04/Nov**	**05/Nov**
Spectral	0.05	0.031	0.023	0.013	0.02
Markov Chain	0.62	0.56	0.56	0.53	0.52
3-gram	0.67	0.54	0.61	0.592	0.601
TIRESIAS	**0.83**	**0.82**	**0.83**	**0.82**	**0.81**

Table 4.1: Prediction precision comparison study: TIRESIAS vs. baseline approaches.

training and test are different, which makes TIRESIAS' prediction results even stronger.

Over the days, we observe a trend that the prediction performance of TIRESIAS drops slightly in terms of all three evaluation metrics. Take the model trained on 2 November for example, its prediction precision drops by 4% from 0.83 to 0.79. In Section 4.4.4 we study if variations (e.g., a longer training period) in the model's training data would offer better performance and how stable the trained TIRESIAS performs over consecutive days. Note that 'micro'-averaging in a multi-class setting produces equal Precision, Recall and F1-Measure. For the rest of the evaluation process, we therefore use precision as the main evaluation metric.

4.4.3 Comparison Study

In this section we aim at studying whether the higher complexity of Recurrent Neural Networks is required for the task of predicting security events, or whether simpler baseline methods would be enough for the task at hand. For comparison purposes, we implemented *first-order* Markov Chain [165] and 3-gram model [166] (equivalent to the second order Markov Chain model) in Python 2.7.1. Note that it is natural to consider a higher order (e.g., n-order where $n > 2$) Markov Chain model for security event prediction, however, due to the exponential states issue associated with high order Markov Chain models, it is computationally costly to build such a high order model for 4,495 events. Finally we use the sp2learning [167] package to build a spectral learning model [168] for sequence prediction as the third baseline prediction model. These three methods are often used to model sequences of elements in several

fields and, being simpler than our RNN models and widely used in sequence prediction, they are relevantly good baselines to compare Tiresias with.

Experiment setup. The comparison study uses daily data (1 November - 5 November) from **D1**. To evaluate Tiresias in this case, all training, validation, and test data come from the same day.

Comparison study results. Table 4.1 shows the precision of Tiresias compared to simpler systems. Table 4.1 shows that Tiresias outperforms the baseline methods but also that 3-grams perform better than Markov Chains, and Markov Chains perform better than the spectral learning method. This particular order shows the importance of sequence memory as the system that performs best among the baselines is the 3-grams. However, 3-grams are less effective than Tiresias. This is due to two of the main characteristics of neural networks: the capacity of filtering noise and the longer term memory. As Table 4.1 shows, Tiresias has precision values higher than 0.8 in all the five days of tests showing a very good level of reliability. In Section 4.4.6 we show that the long-term memory that is an important feature of RNNs plays a key role in correctly predicting security events. Note that we didn't report a comparison of the computation time among the methods due to the fact that Tiresias leverages GPUs to train RNN models and the baseline methods rely on traditional CPUs, and therefore Tiresias is in general much quicker to run. For example, our 3-gram implementation took over 10 days for training, yet Tiresias requires only ∼10 minutes per epoch using GPUs.

4.4.4 Influence of Training Period Length

In this section we look at whether training Tiresias on longer periods of time achieves better prediction performance.

Experiment setup. We use the experimental setup as illustrated in Figure 4.6 for Tiresias' performance evaluation. From **D1**, we train our predictive model using **one day** of data and evaluate on the test data from 8 November to 17 November. For example, we train Tiresias using data from 2 November and evaluate its prediction performance on test data from 8 November

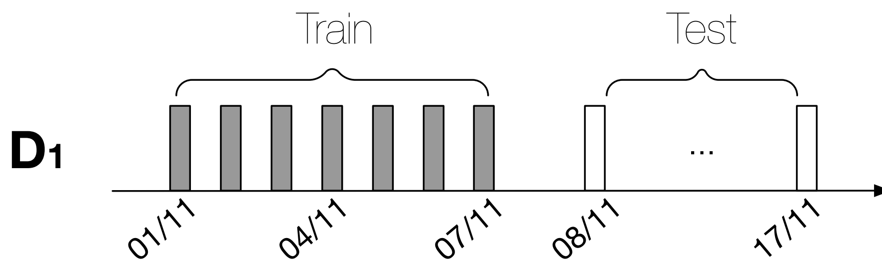


Figure 4.6: Experimental setup for multiple day evaluation of TIRESIAS (Section 4.4.4).

to 17 November. To evaluate if a longer training period can offer better prediction performance, we also train our predictive model using **one week** of data (from 1 November to 7 November) and evaluate its performance in the aforementioned period.

Experiment results. In this experiment we evaluate the performance of our security event forecast model in predicting the exact upcoming event several days after the initial model was trained. The goal is to determine how well our predictive model ages in the short term and to make sure that it remains effective in predicting security events without the need to re-tune it after this period of time.

The question that this experiment is trying to answer is whether there is a difference in training the models over longer periods of time, such as one week, rather than one day. Table 4.2 provides some insights into this question. First, we used the first **five** days of November on their own to build five models. Second, we built one single model from the first **seven days** of the same month. We then tested the six different models (five based on one day of data and one based on one week of data) on **ten days** of data from 8 November to 17 November. *Overall, the training over one week of data produces similar results as those obtained using training over only one day of data.* On average, TIRESIAS trained with one week data can achieve a precision score of 0.819, which is 0.3% higher than that of the models trained with one day data.

These results demonstrate that TIRESIAS can offer good accuracy with stable performance over time since the standard deviation of precision scores

over the measurement period of 10 days is small (∼0.02). However, on 8 November and on 16 November the results are slightly different, exhibiting a higher accuracy for the week-long trained model. While in the first case (8 November) it is probably due to the proximity of the test day to the training week, the second case (16 November) appears to be an outlier. We further observe that the time proximity of the training and test data appears to have a positive impact on the prediction accuracy. Indeed, we can see that the model trained over one day of data is as efficient as the one trained over one week of data when tested on alerts generated only a few days later, probably due to the similarity among attack behaviors observed within a few days. The week-long trained model appears to be more efficient in the presence of deviating, or outlying attack behaviors in the test phase. This can easily be explained by the fact that the more data is used to build a model the more complete the model is. Hence it can better deal with rare events or deviating attack behaviors.

One of the reasons why Tiresias' prediction precision might suddenly decrease is if the set of alerts significantly changes from a day to another, for example because a new vulnerability starts being exploited in the wild, a system patch fixes an existing one, or a major version of a popular software gets released. For this reason, in our architecture discussed in Section 4.2 we included a component that monitors the performance of Tiresias and can trigger a re-training of the system if it is deemed necessary. In the experiment discussed in Table 4.2, for example, the precision performance on 16 November drops by 6.9% on average from 8 November. This could indicate to the operator that something significant changed in the monitored systems and that Tiresias needs to be retrained. As we will show in Section 4.4.7, this can be done in batch and it takes well less than a day to complete.

4.4.5 Stability Over Time

In this Section we evaluate Tiresias' prediction accuracy when the training data is several months older than the test data. Our goal is to evaluate the

	Test Date (Evaluation Metric: Precision)									
Model Training Date	**08/Nov**	**09/Nov**	**10/Nov**	**11/Nov**	**12/Nov**	**13/Nov**	**14/Nov**	**15/Nov**	**16/Nov**	**17/Nov**
01/Nov	0.815	0.823	0.822	0.794	0.789	0.814	0.817	0.816	0.746	0.774
02/Nov	0.821	0.827	0.826	0.801	0.792	0.82	0.819	0.82	0.76	0.79
03/Nov	0.822	0.827	0.826	0.80	0.794	0.82	0.82	0.817	0.742	0.769
04/Nov	0.820	0.828	0.827	0.797	0.797	0.822	0.823	0.82	0.75	0.77
05/Nov	0.817	0.825	0.823	0.791	0.791	0.818	0.815	0.815	0.747	0.775
01/Nov - 07/Nov	0.836	0.83	0.823	0.82	0.801	0.815	0.816	0.812	0.783	0.773

Table 4.2: Evaluation of TIRESIAS' prediction precision between 8th November and 17th November.

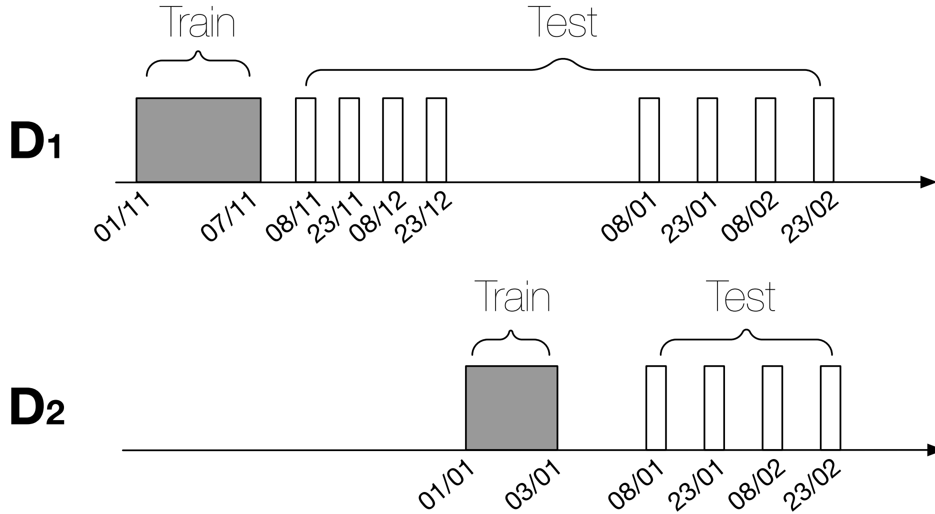


Figure 4.7: Experimental setup for TIRESIAS reliability evaluation (Section 4.4.5).

reliability of the model in case there is no retraining for several months. As we discussed, TIRESIAS is able to detect when it needs to be retrained, however this operation does not come for free and therefore it is desirable to minimize it as much as possible.

Experiment setup. The experimental setup is illustrated in Figure 4.7. We train our predictive model using both **one day** of data (from 1 November to 5 November respectively) and **one week** of data (from 1 November to 7 November) from **D1**. Additionally for comparison purposes, we train three more predictive models using **one day** of data (from 1 January to 3 January respectively) from **D2**. The test data consists of two days per month (on the 8th and the 23rd) so as to obtain a representative dataset from November 2017 until February 2018.

Experiment results. Table 4.3 shows the results obtained using the same training sets as in the previous Section augmented with three days in January, i.e., one day-long model for each of the first five days of November 2017, one week-long model for the first seven days of the same month and one day-long model for each of the first three days of January 2018. The prediction precision results presented in Table 4.3 show consistency through the different training sets and a good level of stability, as the performance does not decrease dramatically over time. Moreover, the week-long training set does not show increased accuracy compared to the day-long ones. These new results thus confirm those from Section 4.4.2 and show that *(i) the model quickly converges towards high accuracy with only one or a few days of training data, and (ii) the model ages very well even months after it was built.*

December discontinuity. Table 4.3 shows a particular behavior between 8 December and 23 December: Tiresias' precision increases. We would normally expect the system's precision to slightly decrease over time, possibly following a pattern, while in this case the precision increases. To investigate this phenomenon, we looked for potential differences in the raw data and noticed that the test data collected after 8 December exhibits a significant deviation with respect to one specific security event ID: the presence of one of the top three recorded alarms decreased by an order of magnitude, having a comparable number of occurrences to alerts occupying the 4th to 10th position. The alarm is related to DoublePulsar, a vulnerability disclosed in the first half of 2017. Such change may be due to different reasons. The most probable reason, however, could be the installation of patches: Microsoft releases monthly updates for Windows every 2nd Tuesday of the month (e.g., 12 December 2017) and many software- and hardware-related companies release patches immediately following Microsoft's. Finally, a small change to the IPS signatures or to the attack modus operandi can heavily impact the hit rate of a given alarm.

Comparison study. To further investigate this December discontinuity phenomenon we decided to assess the impact of the training data on the model

Model Training Date(s)	Test Date (Evaluation Metric: Precision)							
	2017				2018			
	08/Nov	23/Nov	08/Dec	23/Dec	08/Jan	23/Jan	08/Feb	23/Feb
01/Nov	0.815	0.785	0.832	0.899	0.899	0.921	0.93	0.921
02/Nov	0.821	0.8	0.835	0.895	0.896	0.921	0.931	0.918
03/Nov	0.822	0.782	0.835	0.898	0.899	0.923	0.93	0.922
04/Nov	0.820	0.793	0.834	0.901	0.898	0.922	0.929	0.921
05/Nov	0.817	0.79	0.833	0.9	0.898	0.921	0.929	0.92
01/Nov-07/Nov	0.836	0.788	0.829	0.895	0.892	0.917	0.925	0.915
01/Jan	-	-	-	-	0.905	0.927	0.931	0.926
02/Jan	-	-	-	-	0.908	0.926	0.930	0.924
03/Jan	-	-	-	-	0.914	0.933	0.935	0.929

Table 4.3: Evaluation of TIRESIAS's prediction precision on every 8th and 23rd of each month.

accuracy. To this end, we considered the training sets from data collected on the first three days of January and tested on the January and February dates (bottom part of Table 4.3). We can see that TIRESIAS trained in January performs slightly better than when trained in November. These results show that the results by TIRESIAS remain reliable even months after the system was trained. Nevertheless, in the case of a sudden decrease in precision due to an adverse change in the data (e.g., the emergence of a new attack), TIRESIAS would be able to detect this and prompt a retraining, as discussed in Section 4.2.

4.4.6 Sequence Length Evaluation

In Section 4.4.3 we showed that TIRESIAS outperforms simpler systems that do not take advantage of long-term memory in the same way as the RNN model used by our approach. In general, understanding how Deep Learning models work is challenging, and they are often treated as black boxes. To make matters more complex, RNNs do not only rely on long-term memory, but also on short-term memory, in particular to filter out noise.

In this section we aim at understanding whether long-term memory is more influential in making decision than short-term memory or vice versa. With relying on short-term memory we mean a system that relies on a few elements of the sequence to make its decision, that is, the ones closest to the element that the system is trying to guess. With relying on long-term memory we mean when the system uses the whole sequence or a large part of it to take

its decision on what the next security event could be. Intuitively, if short-term memory was predominant, we would not expect the performance of TIRESIAS to increase with the number of observed events.

As looking into the Neural Network weights may be a complicated way to understand which type of memory is more important for the model, we decided to focus on the occurrences of successfully and unsuccessfully guessed events. Every event guessed by TIRESIAS has a probability (confidence score) associated to it. First, we look at the distribution of the confidence scores among successfully guessed events (Figure 4.8a) and unsuccessfully guessed ones (Figure 4.8b). As it can be seen, both types of events present a very skewed distribution in their confidence scores, with a negligible number of events being predicted with a probability of less than 0.5.

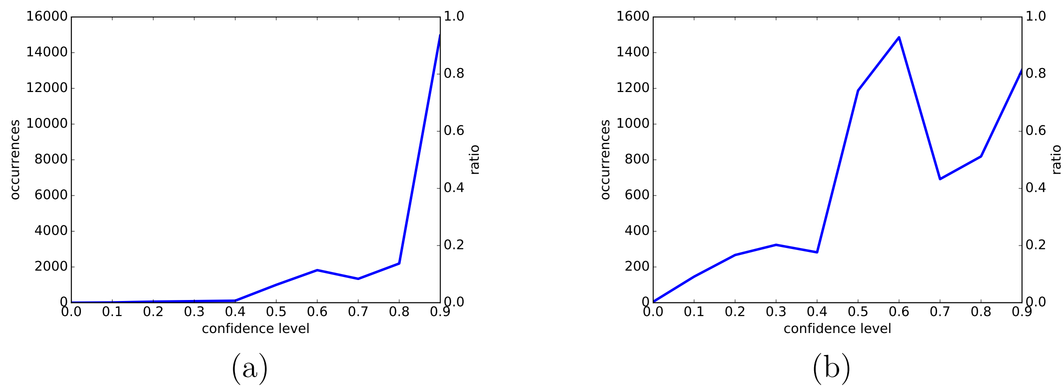


Figure 4.8: Quantity of successfully and unsuccessfully guessed events. The Y axis on the left of each graph is the occurrence of successes/failures with at least the probability indicated on the X axis according to the system. The Y axis on the right is the ratio between the value on the other Y axis and the total of successes/failures.

To better understand if TIRESIAS' results are mainly due to the use of long-term rather than short-term memory, we checked how unique the sequences on which TIRESIAS makes its decisions are. These quantitative results can hint at which kind of approach is used by the algorithm. We try to evaluate the occurrences of the sequence in which the system tried to guess the last event compared to all those that differ from it for the last event (the one that TIRESIAS tried to guess). This analysis is carried out for sequences

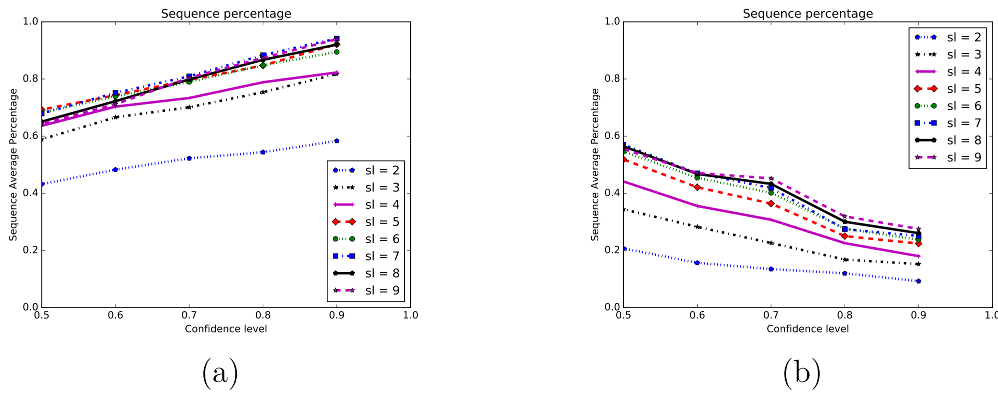


Figure 4.9: The plots show the percentage of the sequences correctly guessed (a) or failed to guess (b) with respect to sequences that share all the events but the last. On the X axis, as for Figures 4.8a and 4.8b there is the confidence level of the sequences used by the system. Figures show that sequences of at least 5 events (sl >=5) are quite unique, therefore long term memory is a crucial factor in the system accuracy.

of length $i+1, (i=2,...,9)$ where i represents the number of events before the last we take into consideration. For all the successful/unsuccessful sequences we calculate the ratio between the times in which we had those i events and Tiresias predicted the last event correctly and the times in which we had the same i events followed by any event (included the right one). According to the probability value of the guessed event, we calculate the average probability.

Figure 4.9a shows the data for all the sequences for which the last event has been correctly guessed by Tiresias. Note that the X axis starts at 0.5 because, as Figures 4.8a and 4.8b showed, the number of predictions with a lower confidence is very low. The values for i (sequence length) less than 5 show that the system's prediction is not very confident. Longer sequences (i greater than 5), instead, are more unique and often correctly predict the last event. The opposite happens when we evaluate the sequences involving events not guessed correctly by our system (Figure 4.9b). In fact, the left part of the graph presents sequences where the wrongly guessed event was rarely the one following the i previous events. This may mean that in those cases there are sequences that differ only for the last event and a few events are quite frequent.

Takeaways. Long sequences including the guessed event are more frequent when we analyze the successful guesses. This situation is more common than the unsuccessful guesses as the system reaches high accuracy. Therefore, according to the graphs the system seem to rely more on long-term memory than short-term memory.

4.4.7 Tiresias Runtime Performance

We now discuss the specific characteristics of the system and its runtime performance.

The training phase is the longest one: building a Tiresias model is a long process that can be performed offline. Tiresias takes around 10 hours to retrain the model. Considered the stability of the model, which as shown in Section 4.4.5 can be effective for long periods of time, rebuilding the model does not have to be done every day. We also showed that it is possible to identify when the system needs retraining because of a discontinuity in the distribution of events (see Section 4.4.4). Once trained, Tiresias takes 25ms to 80ms to predict the upcoming event using the variable-length security events in a given system.

Tiresias' predictive model trained using one day of data is about 31MB. It can be easily pushed to the endpoints with limited network footprint. Note that with the advance of deep learning libraries, especially the recent development of TensorFlow, it is feasible for Tiresias to be deployed not only in traditional endpoints (e.g., PCs) but in mobile and embedded devices as well. This is another aspect that exemplifies the general applicability of Tiresias.

4.5 Case Studies

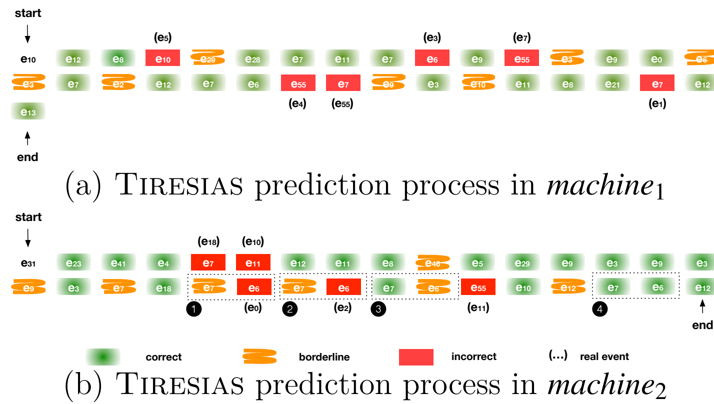
In this section we describe a set of case studies showcasing the capabilities of Tiresias in different real-world scenarios. We first show how Tiresias can be used to detect a coordinated multi-step attack against a Web server (Section 4.5.1). We then provide a number of real-world settings in which

TIRESIAS' prediction labels can be modified to achieve specific goals, for example predicting entire classes of attacks (Section 4.5.2).

4.5.1 Predicting Events in a Multi-Step Attack

The first scenario where TIRESIAS' prediction capability can be leveraged is when facing multi-step, coordinated attacks, i.e., a single attack involving multiple steps performed sequentially or in parallel by an attacker and resulting in multiple alerts being raised by the IPS. The difficulty of identifying such attacks originates from the fact that some of the intermediate steps of a multi-step attack can be considered benign when seen individually by an IPS engine. Moreover, most attacks observed in the wild are the result of automated scripts, which are essentially programmed to check for some precondition on the victim systems and subsequently trigger the adequate exploit(s). For instance, an attack might consists of the following steps: (i) run reconnaissance tasks if port `80/tcp` (HTTP) is open, (ii) trigger a list of exploits against the Web application framework, e.g., Apache and (iii) execute a list of exploits against other possible applications running on top of it. Therefore, we may not observe all steps of an attack on every victim system, depending on which branches of the attack scripts were executed. This variability of observed events across systems hinders the identification of the global multi-step attack.

To identify candidate multi-step attacks in our dataset of IPS events we used the following approach. For each event e_i observed on any of the monitored machines we compute its frequency of occurrence across all machines. We then consider a candidate multi-step attack any sequence of events e_i occurring at the same frequency with an error margin of 10% to capture the variability in attacks as explained above. We also set a *support* threshold on the number of machines exhibiting that sequence so as to avoid a biased frequency obtained from too few samples. To uncover the case study presented here we empirically set this threshold to 1000 machines. For network-sourced attack steps, we also extract the source IP address to determine the likelihood of the global event sequence to be generated by a single attacker.



(a) TIRESIAS prediction process in *machine*$_1$

(b) TIRESIAS prediction process in *machine*$_2$

Figure 4.10: Step by step visualization of TIRESIAS prediction process in two real systems. TIRESIAS starts with event e_{10} and e_{31} respectively as the initial feed and predicts the upcoming security event step by step. The predictions are colored by their probabilistic scores, where green indicates TIRESIAS returns a correct prediction with probabilistic score greater than 0.5, orange indicates TIRESIAS returns a correct event prediction with probabilistic score less than 0.5 (but remains the largest probabilistic score), and red indicates a wrong prediction (the actual events are shown in parentheses in this case).

We present the case of a multi-step attack captured by the IPS and which was successfully learned by the prediction tool. The attack consists of multiple attempts to exploit a Web server and the Web application running on top of it. First, the attacker checks the Web server software for several vulnerabilities. In this case, it quickly identifies the server as running Apache and then attempts to exploit several vulnerabilities, such as Struts-related vulnerabilities. It then switches to checking the presence of a Web application running on top and then fingerprinting it. In this step, the attacker triggers different exploits against known vulnerabilities in various Content Management Systems, such as Wordpress, Drupal, Joomla. Eventually, the attack appears to fail as the various steps are individually blocked by the IPS.

To be able to visualize the decision process and explain how TIRESIAS operates given the aforementioned multi-step attack, we feed TIRESIAS a list of security events from a machine that was under the coordinated attack. By putting TIRESIAS in this real-world environment, we are able to visualize how

TIRESIAS predicts the upcoming events as illustrated in Figure 4.10. Note that events $\{e_0, ..., e_{13}\}$ belong to the coordinated attack.

The process operates as follows. Take $machine_1$ in Figure 4.10a for example, TIRESIAS takes event e_{10} as the initial feed and predicts the upcoming event e_{12}. It then verifies with the actual event to check if e_{12} is the correct prediction. In our case, e_{12} is the correct prediction with a confidence score higher than 0.5 (therefore e_{12} in a green box in Figure 4.10a). TIRESIAS automatically leverages both e_{10} and e_{12} as "contextual information" to enable its internal memory array cells to better predict the next event. The same step is repeated: e_8 is correctly predicted, TIRESIAS uses e_{10}, e_{12}, and e_8 to prepare its internal cells. In the case that TIRESIAS makes a wrong prediction, e.g., it predicts e_{10} instead of the actual observed event e_5 (e_{10} is enclosed in a red box in Figure 4.10a), TIRESIAS uses the actual event e_5 together with previous historical events. This enables TIRESIAS to stay on track with the observed events and predict events that are closely relating to those of the coordinated attack. This may lead TIRESIAS to incorrectly predict some random attacks the system experienced. For example, e_{55} is a 'PHP shell command execution' attack which was observed in the security event sequences, but not part of this particular coordinated attack. It is important to notice that TIRESIAS is able to correctly predict e_{13} (an attack relating to CVE-2017-9805) that was not presented in the previous events, even thought its predecessors, such as e_{12}, e_3, e_{10}, appeared multiple times. We consider this a good example of TIRESIAS using long-term memory to carry out the correct prediction as detailed in Section 4.4.6. It is also interesting to see how TIRESIAS adapts itself during its operation as shown in Figure 4.10b (multiple e_7 and e_6 in dashed line rectangles). We can observe that TIRESIAS did not correctly predict e_6 twice when given e_7 (see ❶ and ❷). Nevertheless, TIRESIAS is capable of leveraging the contextual information (i.e., the actual observed events) to rectify its behavior. As we can observe in ❸, TIRESIAS is able to make a borderline prediction and in ❹ TIRESIAS makes a confident and correct prediction of e_6.

To further exemplify Tiresias' sequential prediction capability in the above setup, we run it on 8 February (2018) test data using the model trained on 3 January (2018) and predict all upcoming events of 200 randomly selected machines (this effectively generates 32,391 sequences due to the step-by-step setup) with a precision of 94%, and against 8 December (2017) test data using a model trained on 4 November (2017) and obtain a precision of 80.89%. These experimental results provide additional evidence of Tiresias' prediction capability in a real-world environment.

4.5.2 Adjusting the Prediction Granularity

The goal of Tiresias is primarily to accurately determine the next event that is going to occur on a given monitored system. In some cases multiple security events might share some common traits. For example, multiple IPS events can be used to describe different attacks against a particular software application, network protocol, etc. These shared traits can then be used to categorize such events. This categorization is specific to the security application that generated these events. Also, the categorization process undoubtedly results in a compressed and coarser-gained set of security events. In this section we describe several cases where such a categorization can be leveraged when the system fails to predict the exact security event but successfully predicts the exact traits, or categories of the attacks, such as the targeted network protocol and software application, or the attack type. To begin with, we extracted categories from the IPS signature labels and descriptions. These categories correspond to characteristics of attacks described by these signatures and are defined as follows. Whenever possible, we identify the *verdict* of the signature, the *severity* of the attack, the *type of attack*, e.g., remote command execution (RCE), SQL injection, etc, the *targeted application*, if any, the *targeted network protocol*, if any, and whether the attack exploits a particular *CVE*. There is thus a one-to-many relationship between each signature and the categories it belongs to. We then uncover machines for which the categories of a mistakenly

predicted event matches exactly the categories of the correct prediction. About 3.5% of failed prediction results exhibit this pattern.

For our first example we consider a machine that was targeted by the Shellshock exploits targeting the Unix shell BASH. Several vulnerabilities were uncovered in the context of these infamous attacks, namely CVE-2014-6271, CVE-2014-6277, CVE-2014-6278, CVE-2014-7169, CVE-2014-7186 and CVE-2014-7187. These six vulnerabilities translate into six IPS signatures. These signatures all belong to the same categories, which include (i) `block`, (ii) `high`, (iii) `RCE`, (iv) `bash` and (v) `CVE`. These categories mean that the exploit attempt is meant to be (i) *blocked* because its potential security impact on the targeted machine is of (ii) *high* severity. This verdict is explained by the fact that, if successful, the exploit would enable the attacker to perform a *remote code execution (or RCE)* by exploiting a known *vulnerability* (with an assigned CVE identifier) against the Unix shell *BASH*. In this case study, a machine was targeted by several of the Shellshock exploits. After observing an attempt to exploit CVE-2014-6271, the system predicted another attempt to exploit the same vulnerability, instead of the correct prediction for CVE-2014-6278. While the event-level prediction result is wrong, the category-level prediction successfully identify an attempt to exploit a Shellshock vulnerability.

The second example is related to a machine that has apparently visited or have been triggered to visit a website distributing fake anti-virus software. Several IPS signatures have been defined to capture different aspects of these malicious websites, for instance, regular expressions matching specific HTML content, suspicious JavaScript code, etc. In this example, the system predicted that the machine would be redirected to a fake AV website containing a particular piece of malicious HTML code while in reality, the machine was redirected to a fake AV website containing a malicious piece of JavaScript code.

Additionally, we evaluate the performance of our security event forecast model in predicting if the upcoming event should be blocked or allowed, a relaxation as the aim is not to determine the exact event that will happen,

but if it is a low-priority alarm (that is still allowed by the product we receive the data from) or if it is a high-priority one (that is blocked immediately). This is one of the essential tasks that a predictive intrusion prevention system needs to resolve. Our experiment shows that the proposed predictive model is able to achieve 88.9% precision in predicting if the upcoming event should be blocked or allowed. This represents a 8% precision increase comparing to the exact event prediction on the same day. Nevertheless, The added value of adjusting the prediction granularity obviously depends on the accuracy of the categorization and the expected level of granularity of the prediction.

4.6 Discussion

Limitations of Tiresias. A recurrent neural network, broadly speaking, is a statistical model. The more the model "sees" (i.e., the more training data) the better the prediction performance is. For rare events, since the model does not have enough training samples, Tiresias may not correctly predict these rare intrusion attempts. Existing statistical and machine learning methods are yet to offer a satisfactory solution to this problem [169, 170, 171]. It would also be interesting to understand whether the recent work by Kaiser *et al.* [172] that makes deep models learn to remember rare words can be applied to predict rare intrusion attempts. DeepLog, a previously proposed system [81], focused on anomaly detection in regulated environments, such as Hadoop and OpenStack, with limited variety of events. In such a specific log environment, DeepLog is able to use a small fraction of normal log entries to train and achieves good detection results. Nevertheless, DeepLog still requires a small fraction of normal log entries would generate enough representative events and patterns. Another limitation following rare events prediction is model retraining when new security events (e.g., new signatures) are created. This retraining is inevitable because machine learning models can only recognize events they have been trained upon. Our experimental results (Section 4.4.3) show that Tiresias takes around 10 hours to retrain and can be redeployed in a timely

fashion in a real-world scenario. As mentioned in Section 4.4.6 the nodes that are activated in an LSTM are not easy to examine. For this reason we cannot guarantee that the system does not take into consideration spurious correlations. At the same time we tried to limit this issue by extensively evaluate Tiresias over a large amount of data and in different settings.

Data limitations. For its operation, Tiresias relies on a dataset of pre-labeled security events. An inherent limitation of this type of data is that an event can be labeled only if it belongs to a known attack class. If, for example, a new zero-day vulnerability started being exploited in the wild, this would not be reflected in the data until a signature is created for it. To reduce the window between when an attack is being run and when it starts being detected by an intrusion prevention system security companies typically use threat intelligence systems and employ human specialists to analyze unlabeled data looking for new attack trends.

Tiresias performance. Sections 4.4 and 4.5 show the effectiveness of the system. The prediction of a security event in such a complicated environment is an important challenge. Tiresias shows the ability of effectively tackling this challenge, showing stability, even when the training set is months older than the test set, and robustness to noise while detecting multistage attacks. We evaluated Tiresias over different time periods to thoroughly prove its qualities; as we discussed, the system may need retraining only in case the data presents radical changes, while its precision does not decrease quickly if the training set is older than the test set. The system can support different dimensions of the training set as it has been tested using one day or one week of data. The differences are minimal: performance is extremely similar, but weekly training seems slightly more robust to anomalies on a specific day of data. However, weekly training sets require more time to build the model. Sections 4.4.6 and 4.5 show how long-term memory and noise filtering are both important factors in the precision of the neural networks, explaining why the baseline methods used in Section 4.4.3 are less precise than Tiresias.

Deployment. The architecture of TIRESIAS enables it to be reasonably flexible in terms of real-world deployment. TIRESIAS can be deployed for each endpoint to proactively defend against attacks as we can see in Section 4.5. At the same time, TIRESIAS can be tailored to protect an enterprise by training with the data coming from that enterprise only and thus better deal with the attacks targeting that enterprise. Note that TIRESIAS' predictive model trained using one day of data is about 31MB. It can be easily pushed to the endpoints with limited network footprint. Together with the mobile TensorFlow library, it is practically feasible for TIRESIAS to protect mobile/embedded devices by training with security event data coming from those devices only. For example, TIRESIAS can be trained using the data collected by smart routers with an IPS installed and deployed in these routers to protect smart home environments.

Evasion. TIRESIAS may be subject to evasion techniques from malicious agents. A vulnerability of deep learning systems is that while the system is classifying samples, it adapts its rules. Therefore, it may be subject to poisoning attacks from a criminal who influences the decision rules using fake actions before attacking the victim. However, to achieve such evasion, the attackers must apply such fake actions at a massive scale and target thousands of machines. A technique that could be used by adversaries is mimicry attacks, i.e., evading security systems by injecting many irrelevant events to cover the alerts generated by a real attack. We argue that TIRESIAS has the potential to be resistant to these attacks. Indeed, we have seen in the case studies that TIRESIAS is able to filter out the noise from the sequences of events observed on the machines, and detect the important events correctly. An interesting future work would be to be able to quantify the amount of events necessary to evade systems like TIRESIAS. Zero day attacks may be difficult to detect: a zero day attack may replicate known sequences of actions to exploit new vulnerabilities, but that would still be detected; when the zero day is applying

a new kind of multi-step attack that has a different sequence of events, it may not be detected.

Chapter 5

Detecting Malware by Using Markov Chains as Behavioral Models

This section is describing the third phase on which we operated in this work: detection. It is describing how we developed a new approach to Android malware detection using API calls. As explained in Section 2.7 there has been a research gap in how to model sequences of API calls to identify malicious apps and distinguish them from benign ones. In fact, our attempt of modeling the sequences of API calls through Markov Chains is novel and aims to take into account the fact that different apps may use different kind of API calls, and may use API calls in different orders.

Detecting malware on mobile devices presents additional challenges compared to desktop/laptop computers: smartphones have limited battery life, making it infeasible to use traditional approaches requiring constant scanning and complex computation [173]. Therefore, Android malware detection is typically performed by Google in a centralized fashion, i.e., by analyzing apps submitted to the Play Store using a tool called Bouncer [174]. However, many malicious apps manage to avoid detection [175], and anyway Android's openness enables manufacturers and users to install apps that come from third-

party market places, which might not perform any malware checks at all, or anyway not as accurately [99].

As a result, the research community has devoted significant attention to malware detection on Android (Section 2.7). Previous work has often relied on the permissions requested by apps [89, 176], using models built from malware samples. This strategy, however, is prone to false positives, since there are often legitimate reasons for benign apps to request permissions classified as dangerous [89]. Another approach, used by DroidAPIMiner [4], is to perform classification based on API calls frequently used by malware. However, relying on the most common calls observed during training prompts the need for constant retraining, due to the evolution of malware and the Android API alike. For instance, "old" calls are often deprecated with new API releases, so malware developers may switch to different calls to perform similar actions, which affects DroidAPIMiner's effectiveness due to its use of specific calls.

5.1 MaMaDroid: Using Static Analysis to Detect Malware

5.1.1 Overview

We now introduce MaMaDroid, a novel system for Android malware detection. MaMaDroid characterizes the transitions between different API calls performed by Android apps – i.e., the sequence of API calls. It then models these transitions as Markov chains, which are in turn used to extract features for machine learning algorithms to classify apps as benign or malicious. MaMaDroid does not actually use the sequence of *raw* API calls, but abstracts each call to either its package or its family. For instance, the API call getMessage() is parsed as:

$$\underbrace{\overbrace{\underbrace{\text{java}}_{\textbf{family}}\text{.lang}}^{\textbf{package}}\text{.Throwable: String getMessage()}}_{\textbf{API call}}$$

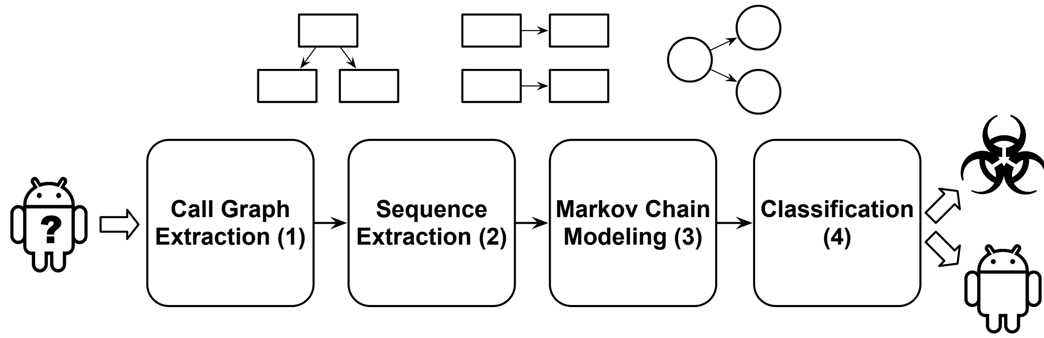


Figure 5.1: Overview of MaMaDroid operation. In (1), it extracts the call graph from an Android app, next, it builds the sequences of (abstracted) API calls from the call graph (2). In (3), the sequences of calls are used to build a Markov chain and a feature vector for that app. Finally, classification is performed in (4), labeling the app as benign or malicious.

Given these two different types of abstractions, we have two modes of operation for MaMaDroid, each using one of the types of abstraction. We test both, highlighting their advantages and disadvantages — in a nutshell, the abstraction to family is more lightweight, while that to package is more fine-grained.

MaMaDroid's operation goes through four phases, as depicted in Figure 5.1. First, we extract the call graph from each app by using static analysis (1), next we obtain the sequences of API calls for the app using all unique nodes in the call graph and associating, to each node, all its child nodes (2). As mentioned, we abstract a call to either its package or family. Finally, by building on the sequences, MaMaDroid constructs a Markov chain model (3), with the transition probabilities used as the feature vector to classify the app as either benign or malware using a machine learning classifier (4). In the rest of this section, we discuss each of these steps in detail.

5.1.2 Call Graph Extraction

The first step in MaMaDroid is to extract the app's call graph. We do so by performing static analysis on the app's apk.[1] Specifically, we use a Java

[1] The standard Android archive file format containing all files, including the Java bytecode, making up the app.

optimization and analysis framework, Soot [177], to extract call graphs and FlowDroid [92] to ensure contexts and flows are preserved.

To better clarify the different steps involved in our system, we employ a "running example," using a real-world malware sample. Specifically, Figure 5.2 lists a class extracted from the decompiled apk of malware disguised as a memory booster app (with package name com.g.o.speed.memboost), which executes commands (rm, chmod, etc.) as root [178]. To ease presentation, we focus on the portion of the code executed in the try/catch block. The resulting call graph of the try/catch block is shown in Figure 5.3. Note that, for simplicity, we omit calls for object initialization, return types and parameters, as well as implicit calls in a method. Additional calls that are invoked when getShell(true) is called are not shown, except for the add() method that is directly called by the program code, as shown in Figure 5.2.

5.1.3 Sequence Extraction

Next, we extract the sequences of API calls from the call graph. Since MaMaDroid uses static analysis, the graph obtained from Soot represents the sequence of functions that are potentially called by the program. However, each execution of the app could take a specific *branch* of the graph and only execute a subset of the calls. For instance, when running the code in Figure 5.2 multiple times, the Execute method could be followed by different calls, e.g., getShell() in the try block only or getShell() and then getMessage() in the catch block.

In this phase, MaMaDroid operates as follows. First, it identifies a set of entry nodes in the call graph, i.e., nodes with no incoming edges (for example, the Execute method in the snippet from Fig. 5.2 is the entry node if there is no incoming edge from any other call in the app). Then, it enumerates the paths reachable from each entry node. The sets of all paths identified during this phase constitutes the sequences of API calls which will be used to build a Markov chain behavioral model and to extract features (see Section 5.1.4).

```
package com.fa.c;

import android.content.Context;
import android.os.Environment;
import android.util.Log;
import com.stericson.RootShell.execution.Command;
import com.stericson.RootShell.execution.Shell;
import com.stericson.RootTools.RootTools;
import java.io.File;

public class RootCommandExecutor {
  public static boolean Execute(Context paramContext) {
    paramContext = new Command(0, new String[] { "cat " + Environment.
         getExternalStorageDirectory().getAbsolutePath() + File.separator + Utilities.
         GetWatchDogName(paramContext) + " > /data/" + Utilities.GetWatchDogName(paramContext)
         , "cat " + Environment.getExternalStorageDirectory().getAbsolutePath() + File.
         separator + Utilities.GetExecName(paramContext) + " > /data/" + Utilities.GetExecName
         (paramContext), "rm " + Environment.getExternalStorageDirectory().getAbsolutePath() +
          File.separator + Utilities.GetWatchDogName(paramContext), "rm " + Environment.
         getExternalStorageDirectory().getAbsolutePath() + File.separator + Utilities.
         GetExecName(paramContext), "chmod 777 /data/" + Utilities.GetWatchDogName(
         paramContext), "chmod 777 /data/" + Utilities.GetExecName(paramContext), "/data/" +
         Utilities.GetWatchDogName(paramContext) + " " + Utilities.
         GetDeviceInfoCommandLineArgs(paramContext) + " /data/" + Utilities.GetExecName(
         paramContext) + " " + Environment.getExternalStorageDirectory().getAbsolutePath() +
         File.separator + Utilities.GetExchangeFileName(paramContext) + " " + Environment.
         getExternalStorageDirectory().getAbsolutePath() + File.separator + " " + Utilities.
         GetPhoneNumber(paramContext) });
    try {
      RootTools.getShell(true).add(paramContext);
      return true;
    }
    catch (Exception paramContext) {
     Log.d("CPS", paramContext.getMessage());
    }
    return false;
  }
}
```

Figure 5.2: Code snippet from a malicious app (com.g.o.speed.memboost) executing commands as root.

Abstracting Calls to Families/Packages. Rather than analyzing raw API calls, we build MaMaDroid to work at a higher level, and operate in one of two modes by abstracting each call to either its package or family. This allows the system to be resilient to API changes and achieve scalability. In fact, our experiments, presented in Section 5.2, show that, from a dataset of 44K apps, we extract more than 10 million unique API calls, which would result in a very large number of nodes, with the corresponding graphs (and feature vectors) being quite sparse. Since as we will see the number of features used by MaMaDroid is the square of the number of nodes, having more than 10 million nodes would result in an impractical computational cost.

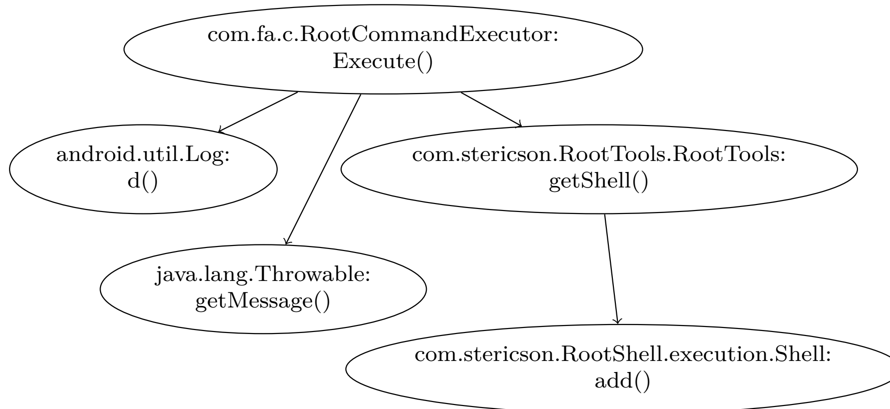


Figure 5.3: Call graph of the API calls in the try/catch block of Figure 5.2. (Return types and parameters are omitted to ease presentation).

When operating in package mode, we abstract an API call to its package name using the list of Android packages[2], which as of API level 24 (the current version as of September 2016) includes 243 packages, as well as 95 from the Google API.[3] Moreover, we abstract developer-defined packages (e.g., com.stericson.roottools) as well as obfuscated ones (e.g. com.fa.a.b.d), respectively, as `self-defined` and `obfuscated`. Note that we label an API call's package as obfuscated if we cannot tell what its class implements, extends, or inherits, due to identifier mangling [179]. When operating in family mode, we abstract to nine possible families, i.e., `android`, `google`, `java`, `javax`, `xml`, `apache`, `junit`, `json`, `dom`, which correspond to the android.*, com.google.*, java.*, javax.*, org.xml.*, org.apache.*, junit.*, org.json, and org.w3c.dom.* packages. Again, API calls from developer-defined and obfuscated packages are abstracted to families labeled as `self-defined` and `obfuscated`, respectively. Overall, there are 340 (243+95+2) possible packages and 11 (9+2) families.

5.1.3.1 Abstraction to Classes

Families and Packages abstractions give two different levels of granularity and, even though the packages abstraction is differentiating among API calls with

[2] `https://developer.android.com/reference/packages.html`
[3] `https://developers.google.com/android/reference/packages`

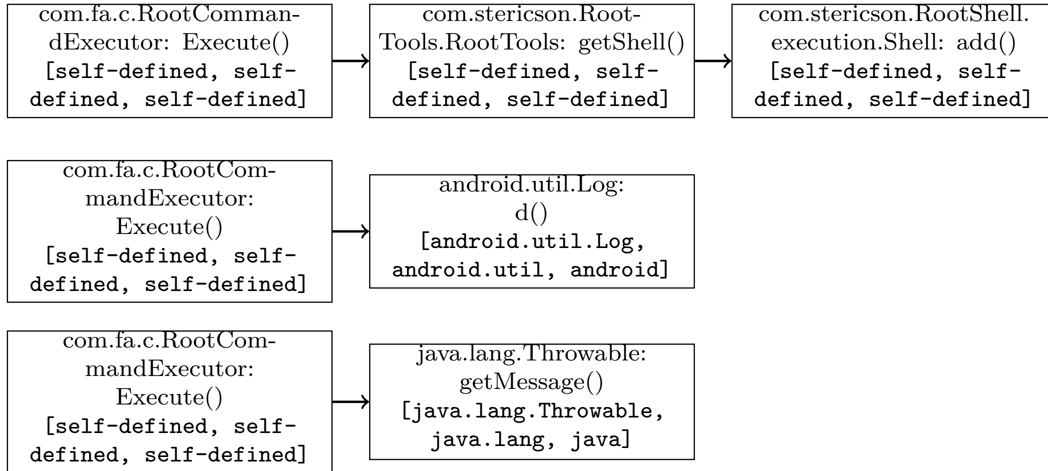


Figure 5.4: Sequence of API calls extracted from the call graphs in Figure 5.3, with the corresponding package/family abstraction in square brackets.

more than an order of magnitude, the information loss about each API is still high. For this reason we designed the class abstraction, aiming to limit the information loss and increase the granularity.

In class mode, we abstract each call to its class name using a whitelist of all class names in the Android and Google APIs, which consists respectively, 4,855 and 1,116 classes.[4]

In Figure 5.4, we show the sequence of API calls obtained from the call graph in Figure 5.3. We also report, in square brackets, the family, the package, and the class to which the call is abstracted.

5.1.4 Markov Chain Based Modeling

Next, MaMaDroid builds feature vectors, used for classification, based on the Markov chains representing the sequences of extracted API calls for an app. Before discussing this in detail, we review the basic concepts of Markov chains.

Markov chains are memoryless models where the probability of transitioning from a state to another only depends on the current state [165]. Markov chains are often represented as a set of nodes, each corresponding to a different state, and a set of edges connecting one node to another labeled with the prob-

[4] https://developer.android.com/reference/classes.html

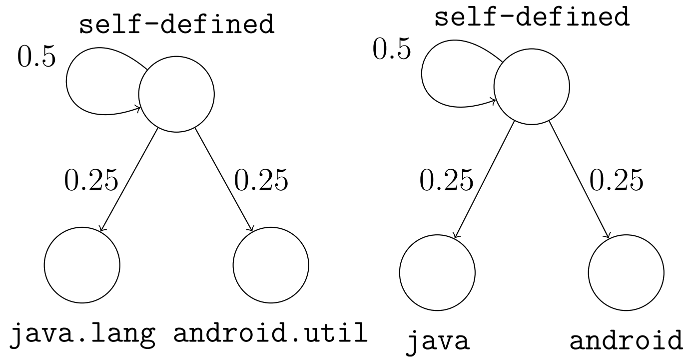


Figure 5.5: Markov chains originating from the call sequence example in Section 5.1.3 when using packages (a) or families (b).

ability of that transition. The sum of all probabilities associated to all edges from any node (including, if present, an edge going back to the node itself) is exactly 1.The set of possible states of the Markov chain is denoted as $\mathscr{S}$. If S_j and S_k are two connected states, P_{jk} denotes the probability of transition from S_j to S_k. P_{jk} is given by the number of occurrences (O_{jk}) of state S_k after state S_j, divided by O_{ji} for all states i in the chain, i.e., $P_{jk} = \frac{O_{jk}}{\sum_{i \in \mathscr{S}} O_{ji}}$.

Building the model. MaMaDroid uses Markov chains to model app behavior, by evaluating every transition between calls. More specifically, for each app, MaMaDroid takes as input the sequence of abstracted API calls of that app – i.e., packages or families, depending on the selected mode of operation – and builds a Markov chain where each package/family is a state and the transitions represent the probability of moving from one state to another. For each Markov chain, state S_0 is the entry point from which other calls are made in a sequence. As an example, Figure 5.5 illustrates the two Markov chains built using packages and families, respectively, from the sequences reported in Figure 5.4.

We argue that considering single transitions is more robust against attempts to evade detection by inserting useless API calls in order to deceive signature-based systems (see Section 2). In fact, MaMaDroid considers all possible calls – i.e., all the branches originating from a node – in the Markov

chain, so adding calls would not significantly change the probabilities of transitions between nodes (specifically, families or packages, depending on the operational mode) for each app.

Feature Extraction. Next, we use the probabilities of transitioning from one state (abstracted call) to another in the Markov chain as the feature vector of each app. States that are not present in a chain are represented as 0 in the feature vector. Also note that the vector derived from the Markov chain depends on the operational mode of MaMaDroid. With families, there are 11 possible states, thus 121 possible transitions in each chain, while, when abstracting to packages, there are 340 states and 115,600 possible transitions and with classes, there are 5,973 states therefore, 35,676,729 possible transitions.

We also apply Principal Component Analysis (PCA) [180], which performs feature selection by transforming the feature space into a new space made of components that are a linear combination of the original features. The first components contain as much variance (i.e., amount of information) as possible. The variance is given as percentage of the total amount of information of the original feature space. We apply PCA to the feature set in order to select the principal components, as PCA transforms the feature space into a smaller one where the variance is represented with as few components as possible, thus considerably reducing computation/memory complexity. Furthermore, the use of PCA could also improve the accuracy of the classification, by taking misleading features out of the feature space, i.e., those that make the classifier perform worse.

5.1.5 Classification

The last step is to perform classification, i.e., labeling apps as either benign or malware. To this end, we test MaMaDroid using different classification algorithms: Random Forests [39], 1-Nearest Neighbor (1-NN) [1], 3-Nearest Neighbor (3-NN) [1], and Support Vector Machines (SVM) [181]. Each model is trained using the feature vector obtained from the apps in a training sample.

Category	Name	Date Range	#Samples	#Samples (API Calls)	#Samples (Call Graph)
Benign	`oldbenign`	Apr 2013 – Nov 2013	5,879	5,837	5,572
	`newbenign`	Mar 2016 – Mar 2016	2,568	2,565	2,465
		Total Benign:	*8,447*	*8,402*	*8,037*
Malware	`drebin`	Oct 2010 – Aug 2012	5,560	5,546	5,512
	`2013`	Jan 2013 – Jun 2013	6,228	6,146	6,091
	`2014`	Jun 2013 – Mar 2014	15,417	14,866	13,804
	`2015`	Jan 2015 – Jun 2015	5,314	5,161	4,451
	`2016`	Jan 2016 – May 2016	2,974	2,802	2,555
		Total Malware:	*35,493*	*34,521*	*32,413*

Table 5.1: Overview of the datasets used in our experiments.

Results are presented and discussed in Section 5.3, and have been validated by using 10-fold cross validation.

Also note that, due to the different number of features used in family/-package modes, we use two distinct configurations for the Random Forests algorithm. Specifically, when abstracting to families, we use 51 trees with maximum depth 8, while, with packages, we use 101 trees of maximum depth 64. To tune Random Forests we followed the methodology applied in Bernard et al. [182].

5.2 Datasets

5.2.1 Employed Dataset

In this section, we introduce the datasets used in the evaluation of MaMaDroid, which include 43,940 apk files – 8,447 benign and 35,493 malware samples. We include a mix of older and newer apps, ranging from October 2010 to May 2016, as we aim to verify that MaMaDroid is robust to changes in Android malware samples as well as APIs. To the best of our knowledge, we are leveraging the largest dataset of malware samples ever used in a research paper on Android malware detection.

Benign Samples. Our benign datasets consist of two sets of samples: (1) one, which we denote as oldbenign, includes 5,879 apps collected by PlayDrone [183]

between April and November 2013, and published on the Internet Archive[5] on August 7, 2014; and (2) another, `newbenign`, obtained by downloading the top 100 apps in each of the 29 categories on the Google Play store[6] as of March 7, 2016, using the googleplay-api tool.[7] Due to errors encountered while downloading some apps, we have actually obtained 2,843 out of 2,900 apps. Note that 275 of these belong to more than one category, therefore, the `newbenign` dataset ultimately includes 2,568 unique apps.

Android Malware Samples. The set of malware samples includes apps that were used to test Drebin [127], dating back to October 2010 – August 2012 (5,560), which we denote as `drebin`, as well as more recent ones that have been uploaded on the VirusShare[8] site over the years. Specifically, we gather from VirusShare, respectively, 6,228, 15,417, 5,314, and 2,974 samples from `2013`, `2014`, `2015`, and `2016`. We consider each of these datasets separately for our analysis.

API Calls and Call Graphs. For each app in our datasets, we extract the list of API calls, using Androguard [184], since, as explained in Section 5.3.5, these constitute the features used by DroidAPIMiner [4], against which we compare our system. Due to Androguard failing to decompress some of the apks, bad CRC-32 redundancy checks, and errors during unpacking, we are not able to extract the API calls for all the samples, but only for 40,923 (8,402 benign, 34,521 malware) out of the 43,940 apps (8,447 benign, 35,493 malware) in our datasets.

Also, to extract the call graph of each apk, we use Soot. Note that for some of the larger apks, Soot requires a non-negligible amount of memory to extract the call graph, so we allocate 16GB of RAM to the Java VM heap space. 2,472 (364 benign + 2,108 malware) samples, Soot is not able to complete the extraction due to it failing to apply the `jb` phase as well as reporting an error

[5] `https://archive.org/details/playdrone-apk-e8`
[6] `https://play.google.com/store`
[7] `https://github.com/egirault/googleplay-api`
[8] `https://virusshare.com/`

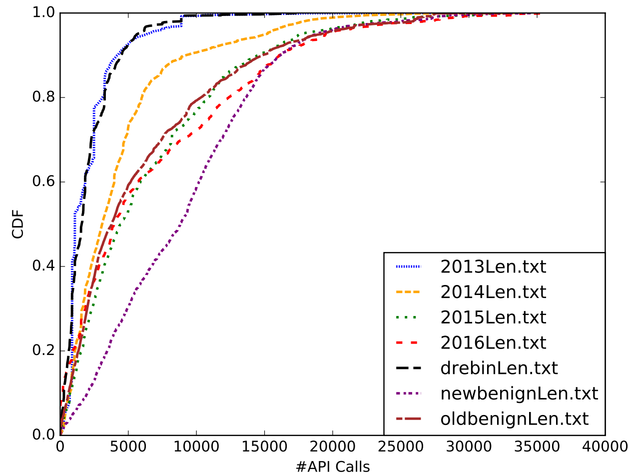


Figure 5.6: CDF of the number of API calls in different apps in each dataset.

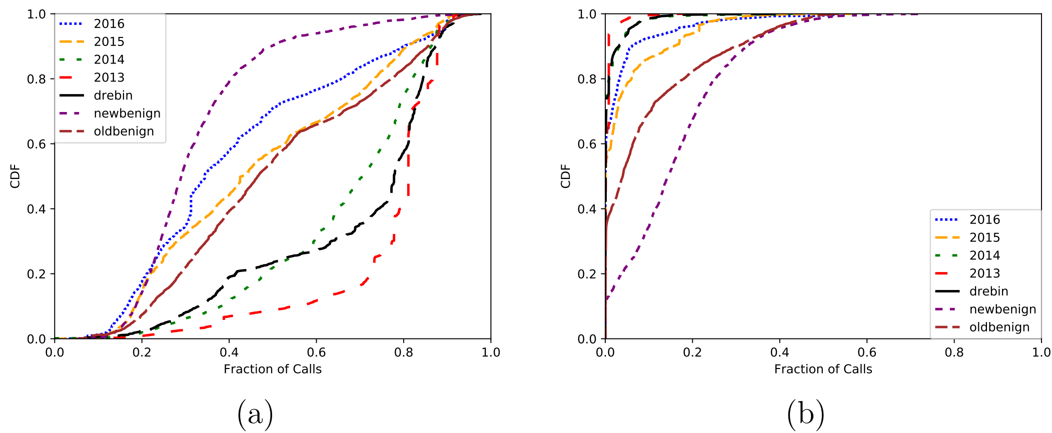


Figure 5.7: CDFs of the percentage of `android` and `google` family calls in different apps in each dataset.

in opening some zip files (i.e., the apk). The `jb` phase is used by Soot to transform Java bytecode into jimple intermediate representation (the primary IR of Soot) for optimization purposes. Therefore, we exclude these apps in our evaluation and discuss this limitation further in Section 5.4.3.

In Table 5.1, we provide a summary of our seven datasets, reporting the total number of samples per dataset, as well as those for which we are able to extract the API calls (second-to-last column) and the call graphs (last column).

Characterization of the Datasets. Aiming to shed light on the evolution of API calls in Android apps, we also performed some measurements over our datasets. In Figure 5.6, we plot the Cumulative Distribution Function (CDF)

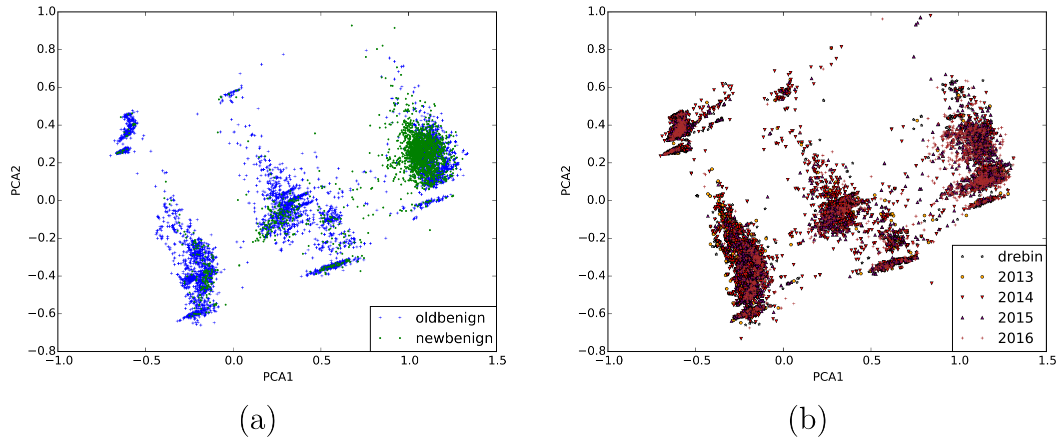


Figure 5.8: Positions of benign vs malware samples in the feature space of the first two components of the PCA (family mode).

of the number of unique API calls in the apps in different datasets, highlighting that newer apps, both benign and malicious, are using more API calls overall than older apps. This indicates that as time goes by, Android apps become more complex. When looking at the fraction of API calls belonging to specific families, we discover some interesting aspects of Android apps developed in different years. In particular, we notice that API calls to the `android` family become less prominent as time passes (Figure 5.7a), both in benign and malicious datasets, while `google` calls become more common in newer apps (Figure 5.7b).

In general, we conclude that benign and malicious apps show the same evolutionary trends over the years. Malware, however, appears to reach the same characteristics (in terms of level of complexity and fraction of API calls from certain families) as legitimate apps with a few years of delay.

Principal Component Analysis. Finally, we apply PCA to select the two most important PCA components. We plot and compare the positions of the two components for benign (Figure 5.8a) and malicious samples (Figure 5.8b). As PCA combines the features into components, it maximizes the variance of the distribution of samples in these components, thus, plotting the positions of the samples in the components shows that benign apps tend to be located in different areas of the components space, depending on the dataset, while

malware samples occupy similar areas but with different densities. These differences highlight a different behavior between benign and malicious samples, and these differences should also be found by the machine learning algorithms used for classification.

5.3 MaMaDroid Evaluation

We now present a detailed experimental evaluation of MaMaDroid. Using the datasets summarized in Table 5.1, we perform four sets of experiments: (1) we analyze the accuracy of MaMaDroid's classification on benign and malicious samples developed around the same time; (2) we evaluate its robustness to the evolution of malware as well as of the Android framework by using older datasets for training and newer ones for testing (and vice-versa); (3) we measure MaMaDroid's runtime performance to assess its scalability; and, finally, (4) we compare against DroidAPIMiner [4], a malware detection system that relies on the frequency of API calls.

5.3.1 Preliminaries

When implementing MaMaDroid in family mode, we exclude the `json` and `dom` families because they are almost never used across all our datasets, and `junit`, which is primarily used for testing. In package mode, to avoid mislabeling when `self-defined` APIs have "android" in the name, we split the `android` package into its two classes, i.e., `android.R` and `android.Manifest`. Therefore, in family mode, there are 8 possible states, thus 64 features, whereas, in package mode, we have 341 states and 116,281 features (cf. Section 5.1.4).

As discussed in Section 5.1.5, we use four different machine learning algorithms for classification – namely, Random Forests, 1-NN, 3-NN, and SVM. Since both accuracy and speed are worse with SVM than with the other three algorithms, we omit results obtained with SVM. To assess the accuracy of the classification, we use the F-measure metric.

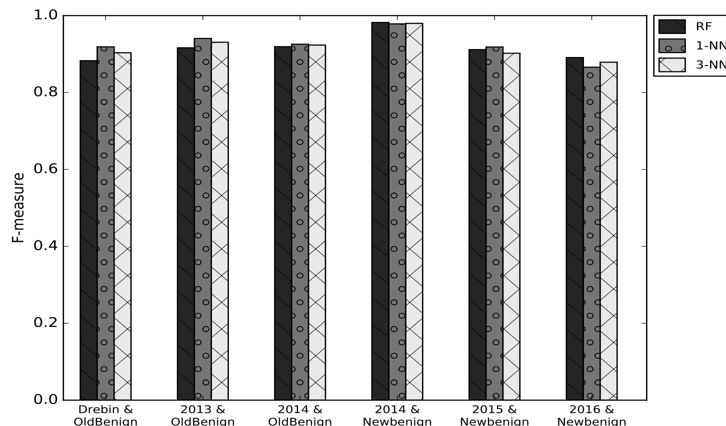


Figure 5.9: F-measure of MaMaDroid classification with datasets from the same year (family mode).

Dataset / Mode	[Precision, Recall, **F-measure**]																	
	`drebin` & `oldbenign`			2013 & `oldbenign`			2014 & `oldbenign`			2014 & `newbenign`			2015 & `newbenign`			2016 & `newbenign`		
Family	0.82	0.95	**0.88**	0.91	0.93	**0.92**	0.88	0.96	**0.92**	0.97	0.99	**0.98**	0.89	0.93	**0.91**	0.87	0.91	**0.89**
Package	0.95	0.97	**0.96**	0.98	0.95	**0.97**	0.93	0.97	**0.95**	0.98	1.00	**0.99**	0.93	0.98	**0.95**	0.92	0.92	**0.92**
Family (PCA)	0.84	0.92	**0.88**	0.93	0.90	**0.92**	0.87	0.94	**0.90**	0.96	0.99	**0.97**	0.87	0.93	**0.90**	0.86	0.88	**0.87**
Package (PCA)	0.94	0.95	**0.94**	0.97	0.95	**0.96**	0.92	0.96	**0.94**	0.97	1.00	**0.99**	0.91	0.97	**0.94**	0.88	0.89	**0.89**

Table 5.2: F-measure, precision, and recall obtained by MaMaDroid, using Random Forests, on various dataset combinations with different modes of operation, with and without PCA.

Finally, note that all our experiments perform 10-fold cross validation using at least one malicious and one benign dataset from Table 5.1. In other words, after merging the datasets, the resulting set is shuffled and divided into ten equal-size random subsets. Classification is then performed ten times using nine subsets for training and one for testing, and results are averaged out over the ten experiments.

5.3.2 Detection Performance

We start our evaluation by measuring how well MaMaDroid detects malware by training and testing using samples that are developed around the same time. To this end, we perform 10-fold cross validations on the combined dataset composed of a benign set and a malicious one. Table 5.2 provides an overview of the detection results achieved by MaMaDroid on each combined dataset, in the two modes of operation, both with PCA features and without. The reported F-measure, precision, and recall scores are the ones obtained with Random Forest, which generally performs better than 1-NN and 3-NN.

Family mode. In Figure 5.9, we report the F-measure when operating in family mode for Random Forests, 1-NN and 3-NN. The F-measure is always at least 88% with Random Forests, and, when tested on the `2014` (malicious) dataset, it reaches 98%. With some datasets, MaMaDroid performs slightly better than with others. For instance, with the `2014` malware dataset, we obtain an F-measure of 92% when using the `oldbenign` dataset and 98% with `newbenign`. In general, lower F-measures are due to increased false positives since recall is always above 91%, while precision might be lower, also due to the fact that malware datasets are larger than the benign sets. We believe that this follows the evolutionary trend discussed in Section 5.2.1: while both benign and malicious apps become more complex as time passes, when a new benign app is developed, it is still possible to use old classes or re-use code from previous versions and this might cause them to be more similar to old malware samples. This would result in false positives by MaMaDroid. In general, MaMaDroid performs better when the different characteristics of malicious and benign training and test sets are more predominant, which corresponds to datasets occupying different positions of the feature space.

Package mode. When MaMaDroid runs in package mode, the classification performance improves, ranging from 92% F-measure with `2016` and `newbenign` to 99% with `2014` and `newbenign`, using Random Forests. Figure 5.10 reports the F-measure of the 10-fold cross validation experiments using Random Forests, 1-NN, and 3-NN (in package mode). The former generally provide better results also in this case.

With some datasets, the difference in performance between the two modes of operation is more noticeable: with `drebin` and `oldbenign`, and using Random Forests, we get 96% F-measure in package mode compared to 88% in family mode. These differences are caused by a lower number of false positives in package mode. Recall remains high, resulting in a more balanced system overall. In general, abstracting to packages rather than families provides better results as the increased granularity enables identifying more differences

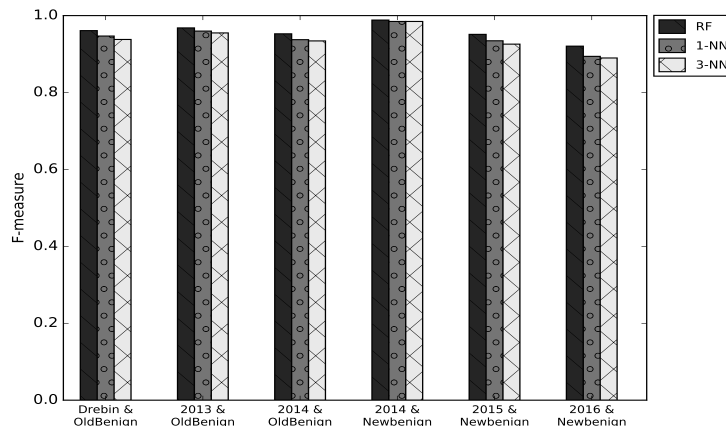


Figure 5.10: F-measure of MaMaDroid classification with datasets from the same year (package mode).

between benign and malicious apps. On the other hand, however, this likely reduces the efficiency of the system, as many of the states deriving from the abstraction are used a only few times. The differences in time performance between the two modes are analyzed in details in Section 5.3.6.

Using PCA. As discussed in Section 5.1.4, PCA transforms large feature spaces into smaller ones, thus it can be useful to significantly reduce computation and, above all, memory complexities of the classification task. When operating in package mode, PCA is particularly beneficial, since MaMaDroid originally has to operate over 116,281 features. Therefore, we compare results obtained using PCA by fixing the number of components to 10 and checking the quantity of variance included in them. In package mode, we observe that only 67% of the variance is taken into account by the 10 most important PCA components, whereas, in family mode, at least 91% of the variance is included by the 10 PCA Components.

As shown in Table 5.2, the F-measure obtained using Random Forests and the PCA components sets derived from the family and package features is only slightly lower (up to 3%) than using the full feature set. We note that lower F-measures are caused by a uniform decrease in both precision and recall.

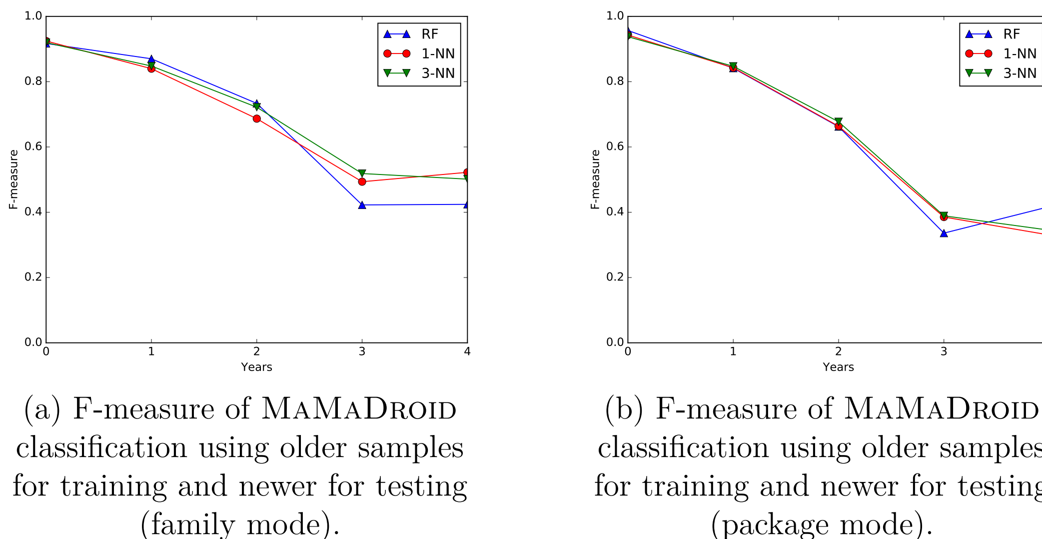
5.3.3 Detection Over Time

As Android evolves over the years, so do the characteristics of both benign and malicious apps. Such evolution must be taken into account when evaluating Android malware detection systems, since their accuracy might significantly be affected as newer APIs are released and/or as malicious developers modify their strategies in order to avoid detection. Evaluating this aspect constitutes one of our research questions, and one of the reasons why our datasets span across multiple years (2010–2016).

As discussed in Section 5.1.2, MaMaDroid relies on the sequence of API calls extracted from the call graphs and abstracted at either the package or the family level. Therefore, it is less susceptible to changes in the Android API than other classification systems such as DroidAPIMiner [4] and Drebin [127]. Since these rely on the use, or the frequency, of certain API calls to classify malware vs benign samples, they need to be retrained following new API releases. On the contrary, retraining is not needed as often with MaMaDroid, since families and packages represent more abstract functionalities that change less over time. Consider, for instance, the `android.os.health` package: released with API level 24, it contains a set of classes helping developers track and monitor system resources.[9] Classification systems built before this release – as in the case of DroidAPIMiner [4] (released in 2013, when Android API was up to level 20) – need to be retrained if this package is more frequently used by malicious apps than benign apps, while MaMaDroid only needs to add a new state to its Markov chain when operating in package mode, while no additional state is required when operating in family mode.

To verify this hypothesis, we test MaMaDroid using older samples as training sets and newer ones as test sets. Figure 5.11a reports the F-measure of the classification in this setting, with MaMaDroid operating in family mode. The x-axis reports the difference in years between training and test

[9] `https://developer.android.com/reference/android/os/health/package-summary.html`



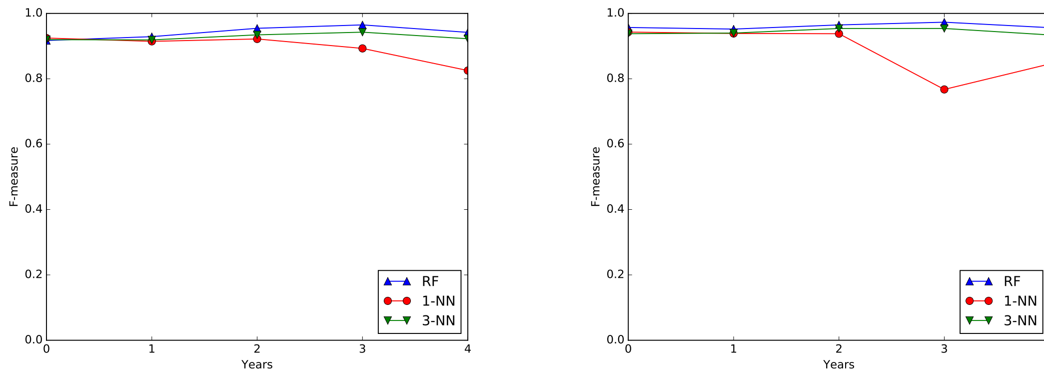
(a) F-measure of MAMADROID classification using older samples for training and newer for testing (family mode).

(b) F-measure of MAMADROID classification using older samples for training and newer for testing (package mode).

Figure 5.11: F-measure values in the different test settings.

data. We obtain 86% F-measure when we classify apps one year older than the samples on which we train. Classification is still relatively accurate, at 75%, even after two years. Then, from Figure 5.11b, we observe that the F-measure does not significantly change when operating in package mode. Both modes of operations are affected by one particular condition, already discussed in Section 5.2.1: in our models, benign datasets seem to "anticipate" malicious ones by 1–2 years in the way they use certain API calls. As a result, we notice a drop in accuracy when classifying future samples and using `drebin` (with samples from 2010 to 2012) or `2013` as the malicious training set and `oldbenign` (late 2013/early 2014) as the benign training set. More specifically, we observe that MAMADROID correctly detects benign apps, while it starts missing true positives and increasing false negatives — i.e., achieving lower recall.

We also set to verify whether older malware samples can still be detected by the system—if not, this would obviously become vulnerable to older (and possibly popular) attacks. Therefore, we also perform the "opposite" experiment, i.e., training MAMADROID with newer datasets, and checking whether it is able to detect malware developed years before. Specifically, Figure 5.12a and 5.12b report results when training MAMADROID with samples from a



(a) F-measure of MaMaDroid classification using newer samples for training and older for testing (family mode).

(b) F-measure of MaMaDroid classification using newer samples for training and older for testing (package mode).

Figure 5.12: F-measure values in the different test settings.

given year, and testing it with others that are up to 4 years older: MaMaDroid retains similar F-measure scores over the years. Specifically, in family mode, it varies from 93% to 96%, whereas, in package mode, from 95% to 97% with the oldest samples.

5.3.4 Case Studies of False Positives and Negatives

The experiment analysis presented above show that MaMaDroid Android malware with high accuracy. As in any detection system, however, the system makes a small number of incorrect classifications, incurring some false positives and false negatives. Next, we discuss a few case studies aiming to better understand these misclassifications. We focus on the experiments with newer datasets, i.e., `2016` and `newbenign`.

False Positives. We analyze the manifest of the 164 apps mistakenly detected as malware by MaMaDroid, finding that most of them use "dangerous" permissions [185]. In particular, 67% of the apps write to external storage, 32% read the phone state, and 21% access the device's fine location. We further analyzed apps (5%) that use the READ_SMS and SEND_SMS permissions, i.e., even though they are not SMS-related apps, they can read and send SMSs as part of the services they provide to users. In particular, a *"in case of*

	Testing Sets									
	drebin & oldbenign		2013 & oldbenign		2014 & oldbenign		2015 & oldbenign		2016 & oldbenign	
Training Sets	[4]	Our Work	[4]	Our Work	[4]	Our Work	[4]	Our Work	[4]	Our Work
drebin & oldbenign	0.32	**0.96**	0.35	**0.95**	0.34	**0.72**	0.30	**0.39**	0.33	**0.42**
2013 & oldbenign	0.33	**0.94**	0.36	**0.97**	0.35	**0.73**	0.31	**0.37**	**0.33**	0.28
2014 & oldbenign	0.36	**0.92**	0.39	**0.93**	0.62	**0.95**	0.33	**0.78**	0.37	**0.75**
	drebin & newbenign		2013 & newbenign		2014 & newbenign		2015 & newbenign		2016 & newbenign	
Training Sets	[4]	Our Work	[4]	Our Work	[4]	Our Work	[4]	Our Work	[4]	Our Work
2014 & newbenign	0.76	**0.98**	0.75	**0.98**	0.92	**0.99**	0.67	**0.85**	0.65	**0.81**
2015 & newbenign	0.68	**0.97**	0.68	**0.97**	0.69	**0.99**	0.77	**0.95**	0.65	**0.88**
2016 & newbenign	0.33	**0.96**	0.35	**0.98**	0.36	**0.98**	0.34	**0.92**	0.36	**0.92**

Table 5.3: Classification performance of DroidAPIMiner [4] vs MaMaDroid (our work).

emergency" app is able to send messages to several contacts from its database (possibly added by the user), which is a typical behavior of Android malware in our dataset, ultimately leading MaMaDroid to flag it as malicious.

False Negatives. We also check the 114 malware samples missed by MaMaDroid when operating in family mode, using VirusTotal.[10] We find that 18% of the false negatives are actually not classified as malware by any of the antivirus engines used by VirusTotal, suggesting that these are actually legitimate apps mistakenly included in the VirusShare dataset. 45% of MaMaDroid's false negatives are *adware*, typically, repackaged apps in which the advertisement library has been substituted with a third-party one, which creates a monetary profit for the developers. Since they are not performing any clearly malicious activity, MaMaDroid is unable to identify them as malware. Finally, we find that 16% of the false negatives reported by MaMaDroid are samples sending text messages or starting calls to premium services. We also do a similar analysis of false negatives when abstracting to packages (74 samples), with similar results: there a few more adware samples (53%), but similar percentages for potentially benign apps (15%) and samples sending SMSs or placing calls (11%).

In conclusion, we find that MaMaDroid's sporadic misclassifications are typically due to benign apps behaving similarly to malware, malware that do not perform clearly-malicious activities, or mistakes in the ground truth labeling.

[10] https://www.virustotal.com

5.3.5 MaMaDroid vs DroidAPIMiner

We also compare the performance of MaMaDroid to previous work using API features for Android malware classification. Specifically, we compare to DroidAPIMiner [4], because: (i) it uses API calls and its parameters to perform classification; (ii) it reports high true positive rate (up to 97.8%) on almost 4K malware samples obtained from McAfee and Genome [186], and 16K benign samples; and (iii) its source code has been made available to us by the authors.

In DroidAPIMiner, permissions that are requested more frequently by malware samples than by benign apps are used to perform a baseline classification. Since there are legitimate situations where a non-malicious app needs permissions tagged as dangerous, DroidAPIMiner also applies frequency analysis on the list of API calls, specifically, using the 169 most frequent API calls in the malware samples (occurring at least 6% more in malware than benign samples) —leading to a reported 83% precision. Finally, data flow analysis is applied on the API calls that are frequent in both benign and malicious samples, but do not occur by at least, 6% more in the malware set. Using the top 60 parameters, the 169 most frequent calls change, and authors report a precision of 97.8%.

After obtaining DroidAPIMiner's source code, as well as a list of packages used for feature refinement, we re-implement the system by modifying the code in order to reflect recent changes in Androguard (used by DroidAPIMiner for API call extraction), extract the API calls for all apps in the datasets listed in Table 5.1, and perform a frequency analysis on the calls. Androguard fails to extract calls for about 2% (1,017) of apps in our datasets as a result of bad CRC-32 redundancy checks and error in unpacking, thus DroidAPIMiner is evaluated over the samples in the second-to-last column of Table 5.1. We also implement classification, which is missing from the code provided by the authors, using k-NN (with k=3) since it achieves the best results according to the paper. We use 2/3 of the dataset for training and

1/3 for testing as implemented by Aafer et al. [4]. A summary of the resulting F-measures obtained using different training and test sets is presented in Table 5.3.

We set up a number of experiments to thoroughly compare DroidAPIMiner to MaMaDroid. First, we set up three experiments in which we train DroidAPIMiner using a dataset composed of `oldbenign` combined with one of the three oldest malware datasets each (`drebin`, `2013`, and `2014`), and testing on all malware datasets. With this configuration, the best result (with `2014` and `oldbenign` as training sets) amounts to 62% F-measure when tested on the same dataset. The F-measure drops to 33% and 39%, respectively, when tested on samples one year into the future and past. If we use the same configurations in MaMaDroid, in package mode, we obtain up to 97% F-measure (using `2013` and `oldbenign` as training sets), dropping to 73% and 94%, respectively, one year into the future and into the past. For the datasets where DroidAPIMiner achieves its best result (i.e., `2014` and `oldbenign`), MaMaDroid achieves an F-measure of 95%, which drops to respectively, 78% and 93% one year into the future and the past. The F-measure is stable even two years into the future and the past at 75% and 92%, respectively.

As a second set of experiments, we train DroidAPIMiner using a dataset composed of `newbenign` combined with one of the three most recent malware datasets each (`2014`, `2015`, and `2016`). Again, we test DroidAPIMiner on all malware datasets. The best result is obtained with the dataset (`2014` and `newbenign`) used for both testing and training, yielding a F-measure of 92%, which drops to 67% and 75% one year into the future and past respectively. Likewise, we use the same datasets for MaMaDroid, with the best results achieved on the same dataset as DroidAPIMiner. In package mode, MaMaDroid achieves an F-measure of 99%, which is maintained more than two years into the past, but drops to respectively, 85% and 81% one and two years into the future.

As summarized in Table 5.3, MaMaDroid achieves significantly higher performance than DroidAPIMiner in all but one experiment, with the F-measure being at least 75% even after two years into the future or the past when datasets from 2014 or later are used for training. Note that there is only one setting in which DroidAPIMiner performs slightly better than MaMaDroid: this occurs when the malicious training set is much older than the malicious test set. Specifically, MaMaDroid presents low recall in this case: as discussed, MaMaDroid's classification performs much better when the training set is not more than two years older than the test set.

5.3.6 Runtime Performance

We envision MaMaDroid to be integrated in offline detection systems, e.g., run by Google Play. Recall that MaMaDroid consists of different phases, so in the following, we review the computational overhead incurred by each of them, aiming to assess the feasibility of real-world deployment. We run our experiments on a desktop equipped with an 40-core 2.30GHz CPU and 128GB of RAM, but only use one core and allocate 16GB of RAM for evaluation.

MaMaDroid's first step involves extracting the call graph from an apk and the complexity of this task varies significantly across apps. On average, it takes 9.2s±14 (min 0.02s, max 13m) to complete for samples in our malware sets. Benign apps usually yield larger call graphs, and the average time to extract them is 25.4s±63 (min 0.06s, max 18m) per app. Note that we do not include in our evaluation apps for which we could not successfully extract the call graph.

Next, we measure the time needed to extract call sequences while abstracting to families or packages, depending on MaMaDroid's mode of operation. In family mode, this phase completes in about 1.3s on average (and at most 11.0s) with both benign and malicious samples. Abstracting to packages takes slightly longer, due to the use of 341 packages in MaMaDroid. On average, this extraction takes 1.67s±3.1 for malicious apps and 1.73s±3.2 for benign

samples. As it can be seen, the call sequence extraction in package mode does not take significantly more than in family mode.

MaMaDroid's third step includes Markov chain modeling and feature vector extraction. This phase is fast regardless of the mode of operation and datasets used. Specifically, with malicious samples, it takes on average 0.2s±0.3 and 2.5s±3.2 (and at most 2.4s and 22.1s), respectively, with families and packages, whereas, with benign samples, averages rise to 0.6s±0.3 and 6.7s±3.8 (at most 1.7s and 18.4s).

Finally, the last step involves classification, and performance depends on both the machine learning algorithm employed and the mode of operation. More specifically, running times are affected by the number of features for the app to be classified, and not by the initial dimension of the call graph, or by whether the app is benign or malicious. Regardless, in family mode, Random Forests, 1-NN, and 3-NN all take less than 0.01s. With packages, it takes, respectively, 0.65s, 1.05s, and 0.007s per app with 1-NN, 3-NN, Random Forests.

Overall, when operating in family mode, malware and benign samples take on average, 10.7s and 27.3s respectively to complete the entire process, from call graph extraction to classification. Whereas, in package mode, the average completion times for malware and benign samples are 13.37s and 33.83s respectively. In both modes of operation, time is mostly ($> 80\%$) spent on call graph extraction.

We also evaluate the runtime performance of DroidAPIMiner [4]. Its first step, i.e., extracting API calls, takes 0.7s±1.5 (min 0.01s, max 28.4s) per app in our malware datasets. Whereas, it takes on average 13.2s±22.2 (min 0.01s, max 222s) per benign app. In the second phase, i.e., frequency and data flow analysis, it takes, on average, 4.2s per app. Finally, classification using 3-NN is very fast: 0.002s on average. Therefore, in total, DroidAPIMiner takes respectively, 17.4s and 4.9s for a complete execution on one app from our

benign and malware datasets, which while faster than MaMaDroid, achieves significantly lower accuracy.

In conclusion, our experiments show that our prototype implementation of MaMaDroid is scalable enough to be deployed. Assuming that, everyday, a number of apps in the order of 10,000 are submitted to Google Play, and using the average execution time of benign samples in family (27.3s) and package (33.83s) modes, we estimate that it would take less than an hour and a half to complete execution of all apps submitted daily in both modes, with just 64 cores. Note that we could not find accurate statistics reporting the number of apps submitted everyday, but only the total number of apps on Google Play [187]. On average, this number increases of a couple of thousands per day, and although we do not know how many apps are removed, we believe 10,000 apps submitted every day is likely an upper bound.

5.3.7 Finer-Grained Abstraction

In Section 5.3, we have showed that building models from abstracted API calls allows MaMaDroid to obtain high accuracy, as well as to retain it over the years, which is crucial due to the continuous evolution of the Android ecosystem. Our experiments have focused on operating MaMaDroid in family and package mode (i.e., abstracting calls to family or package).

In this section, we investigate whether a finer-grained abstraction – namely, to classes – performs better in terms of detection accuracy. Recall that our system performs better in package mode than in family mode due to the system using in the former, finer and more features to distinguish between malware and benign samples, so we set to verify whether one can trade-off higher computational and memory complexities for better accuracy. To this end, as discussed in Section 5.1.3, we abstract each API call to its corresponding class name using a whitelist of all classes in the Android API, which consists of 4,855 classes (as of API level 24), and in the Google API, with 1,116 classes, plus self-defined and obfuscated.

Mode / Dataset	[Precision, Recall, **F**-measure]					
	Class			Package		
drebin, oldbenign	0.95	0.97	0.96	0.95	0.97	0.96
2013, oldbenign	0.98	0.95	0.97	0.98	0.95	0.97
2014, oldbenign	0.93	0.97	0.95	0.93	0.97	0.95
2014, newbenign	0.98	1.00	0.99	0.98	1.00	0.99
2015, newbenign	0.93	0.98	0.95	0.93	0.98	0.95
2016, newbenign	0.91	0.92	0.92	0.92	0.92	0.92

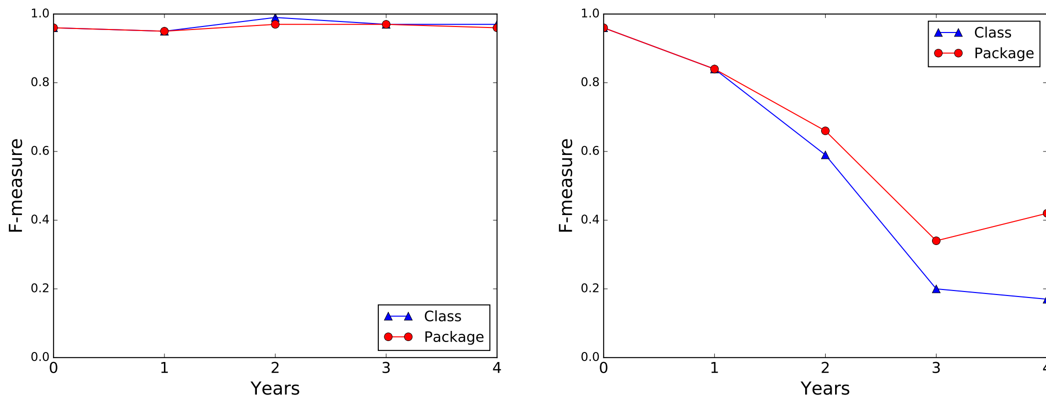
Table 5.4: MaMaDroid's Precision, Recall, and F-measure when trained and tested on dataset from the *same* year in class and package modes.

5.3.8 Reducing the Size of the Problem

Since there are 5,973 classes, processing the Markov chain transitions that results in this mode increases the memory requirements. Therefore, to reduce the complexity, we cluster classes based on their similarity. To this end, we build a co-occurrence matrix that counts the number of times a class is used with other classes in the same sequence in all datasets. More specifically, we build a co-occurrence matrix $\boldsymbol{C}$, of size (5,973·5,973)/2, where $C_{i,j}$ denotes the number of times the i-th and the j-th class appear in the same sequence, for all apps in all datasets. From the co-occurrence matrix, we compute the cosine similarity (i.e., $cos(\boldsymbol{x},\boldsymbol{y}) = \frac{\boldsymbol{x}\cdot\boldsymbol{y}}{||\boldsymbol{x}||\cdot||\boldsymbol{y}||}$), and use k-means to cluster the classes based on their similarity into 400 clusters and use each cluster as the label for all the classes it contains. Since we do not cluster classes abstracted to self-defined and obfuscated, we have a total of 402 labels.

5.3.9 Class Mode Results

In Table 5.4, we report the resulting F-measure in class mode using the above clustering approach when the classifier is trained and tested on samples from the same year. Once again, we also report the corresponding results from package mode for comparison. Overall, we find that class abstraction does not provide significantly higher accuracy. In fact, compared to package mode, abstraction to classes only yields an average increase in F-measure of 0.0012.



(a) F-measure of MaMaDroid classification using newer samples for training and older for testing (class mode).

(b) F-measure of MaMaDroid classification using older samples for training and newer for testing (class mode).

Figure 5.13: F-measure values in the different test settings.

5.3.10 Detection Over Time

We also report in Figures 5.13a and 5.13b (The x-axis shows the difference in years between the training and test data.), the accuracy when MaMaDroid is trained and tested on dataset from different years. We find that, when MaMaDroid operates in class mode, it achieves an F-measure of 0.95 and 0.99, respectively, when trained with datasets one and two years newer than the test sets, as reported in Figure 5.13a). Likewise, when trained on datasets one and two years older than the test set, F-measure reaches 0.84 and 0.59, respectively (see Figure 5.13b).

Overall, comparing results from Figure 5.11b to Figure 5.13b, we find that finer-grained abstraction actually performs worse with time when older samples are used for training and newer for testing. We note that this is due to a possible number of reasons: 1) newer classes or packages in recent API releases cannot be captured in the behavioral model of older tools whereas, families are; and 2) evolution of malware either as a result of changes in the API or patching of vulnerabilities or presence of newer vulnerabilities that allows for stealthier malicious activities.

On the contrary, Figure 5.12b and 5.13a show that finer-grained abstraction performs better when the training samples are more recent than the test samples. This is because from recent samples, we are able to capture the full behavioral model of older samples. However, our results indicate there is a threshold for the level of abstraction which when exceeded, finer-grained abstraction will not yield any significant improvement in detection accuracy. This is because API calls in older releases are subsets of subsequent releases. For instance, when the training samples are two years newer, MaMaDroid achieves an F-measure of 0.99, 0.97, and 0.95 respectively, in class, package, and family modes. Whereas, when they are three years newer, the F-measure is respectively, 0.97, 0.97, and 0.96 in class, package, and family modes.

5.4 Discussion

We now discuss the implications of our results with respect to the feasibility of modeling app behavior using static analysis and Markov chains, discuss possible evasion techniques, and highlight some limitations of our approach.

5.4.1 Lessons Learned

Our work yields important insights around the use of API calls in malicious apps, showing that, by modeling the sequence of API calls made by an app as a Markov chain, we can successfully capture the behavioral model of that app. This allows MaMaDroid to obtain high accuracy overall, as well as to retain it over the years, which is crucial due to the continuous evolution of the Android ecosystem.

As discussed in Section 5.2.1, the use of API calls changes over time, and in different ways across malicious and benign samples. From our newer datasets, which include samples up to Spring 2016 (API level 23), we observe that newer APIs introduce more packages, classes, and methods, while also deprecating some. Figure 5.6, 5.7a, and 5.7b show that benign apps are using more calls than malicious ones developed around the same time. We also notice an interesting trend in the use of Android and Google APIs: malicious

apps follow the same trend as benign apps in the way they adopt certain APIs, but with a delay of some years. This might be a side effect of Android malware authors' tendency to repackage benign apps, adding their malicious functionalities onto them.

Given the frequent changes in the Android framework and the continuous evolution of malware, systems like DroidAPIMiner [4] – being dependent on the presence or the use of certain API calls – become increasingly less effective with time. As shown in Table 5.3, malware that uses API calls released after those used by samples in the training set cannot be identified by these systems. On the contrary, as shown in Figure 5.11a and 5.11b, MaMaDroid detects malware samples that are *1 year* newer than the training set obtaining an 86% F-measure (as opposed to 46% with DroidAPIMiner). After 2 years, the value is still at 75% (42% with DroidAPIMiner), dropping to 51% after 4 years.

We argue that the effectiveness of MaMaDroid's classification remains relatively high "over the years" owing to Markov models capturing app behavior. These models tend to be more robust to malware evolution because abstracting to families or packages makes the system less susceptible to the introduction of new API calls. Abstraction allows MaMaDroid to capture newer classes/methods added to the API, since these are abstracted to already-known families or packages. In case newer packages are added to the API, and these packages start being used by malware, MaMaDroid only requires adding a new state to the Markov chains, and probabilities of a transition from a state to this new state in old apps would be 0. Adding only a few nodes does not likely alter the probabilities of the other 341 nodes, thus, two apps created with the same purpose will not strongly differ in API calls usage if they are developed using almost consecutive API levels.

We also observe that abstracting to packages provides a slightly better tradeoff than families. In family mode, the system is lighter and faster, and actually performs better when there are more than two years between training

and test set samples However, even though both modes of operation effectively detect malware, abstracting to packages yields better results overall. Nonetheless, this does not imply that less abstraction is always better: in fact, a system that is too granular, besides incurring untenable complexity, would likely create Markov models with low-probability transitions, ultimately resulting in less accurate classification. We also highlight that applying PCA is a good strategy to preserve high accuracy and at the same time reducing complexity.

5.4.2 Evasion

Next, we discuss possible evasion techniques and how they can be addressed. One straightforward evasion approach could be to repackage a benign app with small snippets of malicious code added to a few classes. However, it is difficult to embed malicious code in such a way that, at the same time, the resulting Markov chain looks similar to a benign one. For instance, our running example from Section 5.1 (malware posing as a memory booster app and executing unwanted commands as root) is correctly classified by MaMaDroid; although most functionalities in this malware are the same as the original app, injected API calls generate some transitions in the Markov chain that are not typical of benign samples.

The opposite procedure – i.e., embedding portions of benign code into a malicious app – is also likely ineffective against MaMaDroid, since, for each app, we derive the feature vector from the transition probability between calls over the entire app. In other words, a malware developer would have to embed benign code inside the malware in such a way that the overall sequence of calls yields similar transition probabilities as those in a benign app, but this is difficult to achieve because if the sequences of calls have to be different (otherwise there would be no attack), then the models will also be different.

An attacker could also try to create an app from scratch with a similar Markov chain to that of a benign app. Because this is derived from the sequence of abstracted API calls in the app, it is actually very difficult to create sequences resulting in Markov chains similar to benign apps while, at the same

time, actually engaging in malicious behavior. Nonetheless, in future work, we plan to systematically analyze the feasibility of this strategy.

Moreover, attackers could try using reflection, dynamic code loading, or native code [188]. Because MaMaDroid uses static analysis, it fails to detect malicious code when it is loaded or determined at runtime. However, MaMaDroid can detect reflection when a method from the reflection package (`java.lang.reflect`) is executed. Therefore, we obtain the correct sequence of calls up to the invocation of the reflection call, which may be sufficient to distinguish between malware and benign apps. Similarly, MaMaDroid can detect the usage of class loaders and package contexts that can be used to load arbitrary code, but it is not able to model the code loaded; likewise, native code that is part of the app cannot be modeled, as it is not Java and is not processed by Soot. These limitations are not specific of MaMaDroid, but are a problem of static analysis in general, which can be mitigated by using MaMaDroid alongside dynamic analysis techniques.

Malware developers might also attempt to evade MaMaDroid by naming their self-defined packages in such a way that they look similar to that of the `android`, `java`, or `google` APIs, e.g., creating packages like java.lang.reflect.*malware* and java.lang.*malware*, aiming to confuse MaMaDroid into abstracting them to respectively, `java.lang.reflect` and `java.lang`. However, this is easily prevented by whitelisting the list of packages from `android`, `java`, or `google` APIs.

Another approach could be using dynamic dispatch so that a class X in package A is created to extend class Y in package B with static analysis reporting a call to root() defined in Y as X.root(), whereas, at runtime Y.root() is executed. This can be addressed, however, with a small increase in MaMaDroid's computational cost, by keeping track of self-defined classes that extend or implement classes in the recognized APIs, and abstract polymorphic functions of this self-defined class to the corresponding recognized package,

while, at the same time, abstracting as self-defined overridden functions in the class.

Finally, identifier mangling and other forms of obfuscation could be used aiming to obfuscate code and hide malicious actions. However, since classes in the Android framework cannot be obfuscated by obfuscation tools, malware developers can only do so for self-defined classes. MaMaDroid labels obfuscated calls as `obfuscated` so, ultimately, these would be captured in the behavioral model (and the Markov chain) for the app. In our sample, we observe that benign apps use significantly less obfuscation than malicious apps, indicating that obfuscating a significant number of classes is not a good evasion strategy since this would likely make the sample more easily identifiable as malicious.

5.4.3 Limitations

MaMaDroid requires a sizable amount of memory in order to perform classification, when operating in package mode, working on more than 100,000 features per sample. The quantity of features, however, can be further reduced using feature selection algorithms such as PCA. As explained in Section 5.3 when we use 10 components from the PCA the system performs almost as well as the one using all the features; however, using PCA comes with a much lower memory complexity in order to run the machine learning algorithms, because the number of dimensions of the features space where the classifier operates is remarkably reduced.

Soot [177], which we use to extract call graphs, fails to analyze some apks. In fact, we were not able to extract call graphs for a fraction (4.6%) of the apps in the original datasets due to scripts either failing to apply the `jb` phase, which is used to transform Java bytecode to the primary intermediate representation (i.e., jimple) of Soot or not able to open the apk. Even though this does not really affect the results of our evaluation, one could avoid it by using a different/custom intermediate representation for the analysis or use different tools to extract the call graphs.

In general, static analysis methodologies for malware detection on Android could fail to capture the runtime environment context, code that is executed more frequently, or other effects stemming from user input [127]. These limitations can be addressed using dynamic analysis, or by recording function calls on a device. Dynamic analysis observes the live performance of the samples, recording what activity is actually performed at runtime. Through dynamic analysis, it is also possible to provide inputs to the app and then analyze the reaction of the app to these inputs, going beyond static analysis limits. To this end, we plan to integrate dynamic analysis to build the models used by MaMaDroid as part of future work.

Chapter 6

Discussion

This chapter will discuss the different aspects of the technical chapters of this work. Although each of the chapters has its own related discussion sections, the aim of this section is to show how all these contributions have common points and what they highlight.

We presented three phases of study, prediction, and detection of malicious activities on the Internet. In Chapter 3 we presented a framework to study the kind of malware that could be attacking your PC by feeding it simulated user triggers and recording its reactions. Chapter 4 shows TIRESIAS, a system that predicts multi-step attacks phases, in order to prepare system administrators to counter these actions by knowing in advance what is going to happen. Next, we present MAMADROID (Chapter 5), a system for Android malware detection. These three phases are crucial in the fight against cybercriminals and our goal is to tackle adversaries strategies over these different levels. The idea of working on different phases that have the common goal of making cybercriminal operations harder, is based on an holistic approach: it is necessary to improve on several aspects and contemporary operate on different angles to be efficient.

All the techniques of this work have another concept in common: understanding and extracting malicious behavior. We evaluate the fact that malicious behavior has to follow certain paths that can differ from benign ones; for instance, information stealing malware will always have to send these informa-

tion to the C&C servers. Being able to extrapolate the behavior can be the key for more appropriate countering actions. The concept of behavior in this context is seen as the actions defining the malicious events and it can involve some of the actions taken into account that are not necessarily malicious anytime, independently from what operates them (sending personal information on the Internet may be done automatically also by benign software). External actions, such as user triggers, have to be taken into account, as mentioned in Section 2.4 when talking about Zhang et al. [134], context makes the difference in understanding these phenomena.

The techniques, statistical frameworks, and systems created are built on and aiming to reinforce existing techniques and system security research areas. In different terms, we did not reinvent the wheel, we did not invent a new car, but modified the existing ones to make them better. We are not the first ones using a sandbox, but we operate the sandbox to induce the malicious samples to operate different actions, rather than just wait and record. We are not the first ones using deep learning for prediction purposes, but we use it for a specific purpose where, properly tuned, can release its potential. We are not the first ones focusing on API calls and using ML on the information extracted on their usage, but with the use of abstraction and Markov Chains, we relate them into sequences, lose detailed information to gain on the gross idea of what is happening when apps are running.

Causality in malware behavior. In Chapter 3 we extract behavior as the ability to react to certain user triggers. The causality framework based on counterfactual analysis has been proven effective both in an application presenting minor quantities of noise (the extensions leaking browser history in Section 3.3) and in one presenting challenges that caused noise in the results (the malware samples network traffic analysis in Section 3.2). For the second one it has been necessary to apply the whole statistical framework that allowed to rule out the noise and extract the causality relationships. At the beginning, we present the methodology itself and the chapter shows how the methodology

is not efficient because tailored and limited to a specific application, but can be extended as it is general and applicable to different challenges.

As mentioned earlier, this chapter is showing that sandboxes/honeypots can be more than passive tools that record everything done by the pieces of code we analyze. When we passively analyze such code we may have only a few operations made by the analyzed sample, or we may have a stream of actions that do not look coherent. When analyzing this information we might not extract nor analyze what we needed. The simulation of user triggers may help to select the actions we seek, unveil the pieces of information that can lead to the results the defense system is looking for. Our approach to sandboxes is not avoiding the limitations of such tool (Section 3.2.4), it is trying to build on the qualities of the tool. Malware may detect simulated environments, therefore we changed settings that might be checked by malicious samples. However, the simulation of user triggers makes such virtual environments more real to the malicious samples.

The application to network traffic (Section 3.2) is not including any kind of detection systems. When we worked on this project we thought about how to implement such system: given the distribution of how many times samples from each family reacted to the triggers, it is possible to see whether the distribution of triggered experiments for an unknown sample is closer to a family rather than the other ones. When working on this phase, our goal was to investigate whether the counterfactual analysis model and the statistical framework were efficient or not, but, considered how general the model can be, it is possible to add an interesting evolution into a possibly reliable detection system. In such development, it would be important to add triggers and malware families.

Section 3.3 is already showing a possible implementation of a detection system based on unsupervised classification using linear regression. As this application has shown less noise affecting the measurements, we have been able to develop a classification system even though we had a few points available for regression. The causality framework is part of a more complicated system,

it is not stand alone in this case. However, it is the starting point, having the rest of the system building on the results of this step.

Prediction through deep learning. The second phase analyzed in this work is explained in Chapter 4. We successfully aimed to building a classifier based on deep learning and able to predict the exact event that is going to happen next out of 4495 possible options. The results are showing TIRESIAS' ability of efficiently identifying the steps of multi-step attacks to prevent the attacker from doing them.

Behavior is defined as the signatures triggered by the adversary's actions because we applied TIRESIAS to a database of Intrusion Prevention Systems events. In this case, as well as the previous ones, the behavior is given by actions that are essential to the adversary final goal. The key of this phase is foreseeing what an adversary could do, act preemptively to disrupt the adversary's actions. Such approach allows system administrators to prepare the defenses where there weren't and the system could have been hit. The use of an advanced statistical tool as deep learning is carefully evaluated against less complicated structures showing that the potential of this tool is higher than the one of, for instance, Markov Chains as we have to rely on particular mechanisms that are involving long term memory for this specific problem.

Detection of Android malware using Markov Chains. The third phase, following the prediction one, is detection. Prediction is a rather new research trend where we act by anticipating the adversary's moves. However, as this approach cannot be implemented on all the aspects of the malware and malicious behavior world, it is necessary to focus on the detection of malicious samples as well. In the specific case of this work, MAMADROID (Section 5) is a tool focused on the detection of Android malware. In this case MAMADROID models the sequences of API calls of the analyzed apps. The API calls are nothing more than the single actions that the apps can do and the sequences are the apps behavior.

The system is efficient, presenting high F1 scores because there actually is a clear difference between the behavior of benign and malicious apps. Moreover, it fits requirements for the implementation on infrastructures such as markets. MaMaDroid is designed to act as a security check before the malicious apps get on the market. Even though the families abstraction is lightweight and could be implemented on a mobile phone, the packages one is slightly more efficient; the finer resolution of this abstraction results in higher F1 scores at the cost of an heavier model that can be managed more easily by central markets rather than personal devices. MaMaDroid is not immune to evasion techniques, at the same time it is an approach that tackles the Android malware detection field from a different angle with satisfying results. We discuss the possible evasion techniques in Section 5.4.3, but the techniques are not related to how MaMaDroid itself works: they are more related to which techniques have to be used (e.g., static analysis) and the stealthiness of malicious samples.

Chapter 7

Ethical Discussion of this Work

In the previous sections of this work we discussed the technical contributions and how they have been filling open gaps and opening new questions in studies on malicious behavior. In this Section, we are going to analyze what are the ethical implications of these works. We will analyze all the technical sections one by one and then the ethical implications of possible implementations into the wild.

7.1 Research Analysis and Ethics

Doing research in security and cybercrime has several ethical implications. Starting from the most obvious ones, like involving humans, to less obvious ones, like the possible attacks that can be done by malicious samples running in a monitored environment.Every project has ethical implications that have to be evaluated. When doing research, we are interested in the results of every project, but it is important that the work follows guidelines. Moreover, the results we are able to reach are subject to deductions and interpretations. A researcher may come to incorrect conclusions that may affect the decisions of practitioners in the field.

Causality in Malware Activities. When it comes to the research related to the first phase, there are many implications related to the experimental environment to be taken into account: we run malware samples that are active and functioning to study their reactions. It implies that no real information

of any kind should be used during the experiments as malicious samples may otherwise use this information for their own purposes.

Samples may also use the virtual environment for attacks to third machines; these attacks may be different one from the other and it is important to implement security measures: limiting bandwidth to avoid DoS attacks, redirecting SMTP packages to avoid spamming campaigns or targeted blackmailing messages, limited lifetime of the virtual environment to limit the possible advanced exploits that may be run against other machines. All these restrictions are needed to avoid that research becomes an harm to someone else's life.

We already talked about how hard it is to establish causal relationships and how a causal relationship is an extremely strong link between variables, but it is also important to think about what it means to declare such a strong link. Some of the findings related to this phase are intuitive and researchers and practitioners can easily see them by using logic. However, other aspects may be less straightforward; for instance, establishing the linear relationship between amount of history in the browser and aggregate size of the packets sent to leak the history. The rationale behind this result is intuitive, but the reality of telecommunications may affect these results: there can be packets sent more than once or payloads that have been limited in sizes, creating overhead. It is important that findings are valid and that events that may affect them are not undermining the validity of the results.

Predicting Security Alarms due to Malicious Activities Using Deep Learning Algorithms. The second phase is related to the use of advanced tools such as Deep Learning. We often evaluate Deep Learning as a black box from which we take into account only the results. However, in some cases this may lead to having biases being unnoticed. When we do not notice the biases that are affecting the dataset or the decisions we pose important threats of validity of the results. In our field of research this may lead to several issues, (a) the paper is presenting results that are not reliable nor valid, (b) the results

presented raise the bar of acceptance for other research projects in the field without being as good as they are presented, (c) the system may be used for future work that will base itself on invalid constructions or tools.

These reasons may be seen as common to all the research fields that are presenting any kind of result, however, the use of Deep Learning can be even more prone to these issues. Deep Learning (and more in general Machine Learning) is often seen as black magic that solves the problems for the researcher. For this reason it is often used without full knowledge of its rules, protocols, and mechanisms. By applying Deep Learning in this way, researchers cannot notice if something is affecting the validity of their own results.

This work has other ethical implications related to research: TIRESIAS is one of the few works on prediction of events and preemptive behavior. This fairly new trend may become one of the most explored themes in the system security field and the oldest works a reference for the newest ones. The validity of TIRESIAS's methodology, its datasets and evaluation are even more crucial under this light.

Detecting Malware by Using Markov Chains as Behavioral Models. The last phase we worked on is the detection one. The detection phase is one of those that is continuously raising interest over the years. As for our system, MAMADROID is a system applying Machine Learning algorithms in a relatively new field, android malware. While the considerations thought for Deep Learning systems in the previous paragraphs are valid for Machine Learning systems, the works on detection present another important ethical concern: full disclosure and complete reproducibility. This is an ethical concern that affects all the aspects of research; however, TIRESIAS has been created using datasets from a private company and may be implemented by them; the amount of information that can be shared is, therefore, limited. In such cases the work has to have detailed descriptions to allow the community to thoroughly understand it, but cannot be fully disclosed as there is company

related information. MaMaDroid, on the other hand, is using public data on a topic that has been widely explored. When we claim MaMaDroid performances are better than the ones of DroidAPIMiner, we have to be able to prove it. For this reason we preferred using only DroidAPIMiner (of which the authors sent the original source code), rather than other systems (e.g. DREBIN) of which the sourcecode is not available. For the same reason, MaMaDroid source code is publicly available for researchers to try it and compare its efficiency with the one of their systems.

7.2 Systems Ethics and Implementation in the Wild

The area of information security that this work is part of, is system security. Practitioners all over the world are constantly developing systems for their own companies and monitoring the systems produced by researchers in their work. This means that research is influencing the world around us and the implementation of systems that are inspired by researcher's work is not uncommon. In this section we are going to understand what are the implications of the systems developed in this work.

Causality in the wild. In the first part of this thesis we talked about a statistical framework. We already described how the use of triggers may positively affect the sandboxing techniques and how effective this can be, however we also mentioned how important it is to study the validity of the results (Section 7.1). When we foresee this framework implementations into the wild, we think about howits results can be significant. This system could be implemented to analyze new software characteristics in environments where crucial information and operations are, e.g., a bank subnetwork where stock exchange operations are handled. This kind of networks are limited in the amount of operations and the traffic that can be exchanged, therefore it is possible to find out which behaviors are suspicious. Moreover, Ex-Ray is already an effective

application of such framework that could be implemented as security check by those markets that are selling browser extensions.

Prediction of Security Alarms in the wild. The second area of work we talked about is the prediction of alarms. TIRESIAS has shown reliability, however, as expected, it does not have 100% Accuracy. When implemented in the wild, such system has to give indication on whether the prediction is reliable or not and on which could be the risk of an incorrect prediction in case there are two possible events with similar probability. Handling these issues is crucial because a system like TIRESIAS can be a valuable help for system administrators, however, a wrong prediction cannot turn into a disaster for the company. Systems adminitrators rely on the suggestions given by automated systems every day; they rely on wrong suggestions as well, in their decisions. When we talk about prediction of events, a wrong suggestion may be hinting that a specific attack is about to happen instead of another one. We may consider a wrong prediction like not having any prediction at all about the attack, however this is not correct as the reactions of the administrator may affect how the attack develops or, even worse, may delay the application of the correct countermeasures. It is therefore crucial that systems like TIRESIAS give a detailed feedback on which could be the kind of attack but also whether other (completely different) attacks could be misinterpreted and being the ones taking place.

Android Malware Detection in the wild. The evaluation of system errors is crucial also in detection systems like MAMADROID. We have explained in Section 2.2 that systems security is not about having one system able to detect everything. MAMADROID is an extremely effective system, however it may miss malicious apps that without other controls could be injected into the market. As for TIRESIAS, MAMADROID must be carefully evaluated to understand which decision could be borderline or which one could be wrong because the app presents characteristics that are leading to wrong evaluations.

By being able to evaluate all these things it is possible to determine whether it is necessary to further check the specific app or not.

Chapter 8

Conclusions and Final Remarks

This thesis is the final product of three years of research work. We presented the successful research aiming to help in the continuous arms race between cybercriminals and the security community. This work can be an inspirational starting point for new research work as well as being used by practitioners as part of their tools. They have been milestones for the teams working on them and they have been proven to be efficient elements respecting high standards of research.

We worked on tools for measuring, predicting and detecting malicious behavior on the Internet. We decided to operate on these three phases as part of the most crucial ones to apply advanced statistical methods in this field.

In the first phase we have presented a statistical framework to study causality relationships between user and malicious behavior. We have presented and thoroughly evaluated two practical applications of this tool, the study of malicious samples from different malware families in a sandboxing environment and the study of browser extensions leaking browser history.

In the prediction phase we have worked on TIRESIAS, a system able to predict multi-step attacks. We have evaluated the system over different datasets, analyzed it over different timescales, and evaluated case studies to have better insights on the work.

In the detection phase we have created MAMADROID, a system that is applying Markov Chains on sequences of API calls to distinguish between

benign and malicious apps. We analyzed MaMaDroid's performance using different malware datasets, studying the evolution of the malware samples and how the system is able to adapt through the years.

This thesis tackled the issues related to malicious behavior on the Internet from different points of view, focusing on the concept of behavior, how to recognise patterns in Internet behavior (whether benign or malicious) and the use of ML and other advanced statistical methods for this scope. The systems created as part of this work have in common a very important aspect: their versatility. In fact, the causality framework is applicable to several different issues, MaMaDroid can be easily integrated in security systems used by Android markets and the adaptation to different versions of the Android OS (or different OSs using the same kind of API architecture and functionalities) is immediate. Tiresias can be implemented on several devices and we have shown that its detection sub-system is usable by devices with limited power as well.

The versatility of the systems is not the only takeaway from this work. The results are showing, without any doubt, that statistical systems are able to extrapolate and evaluate patterns from different issues in a very efficient way. However, as explained in Section 2.3.2.1for ML systems, the work of the statistical methods would not be that effective without an effective representation. Among the different settings in which we have used statistical evaluation, a particular mention goes to the abstraction invented and implemented in MaMaDroid: this technique is an important contribution to the features extraction for these systems. Abstraction has removed detailed information about API calls by keeping only the general name, representing its very rough behavior (classes and packages) or even just an indication of what is involved (families). The use of abstraction allowed to use a much more lightweight system, but increased the robustness as well. In fact, while the evaluation metrics highlighted great performances even when abstracting at the highest level of granularity, we were able to extract information readable

by users as well on how the classifier was working. Abstraction is a technique that should definitely be evaluated by the community in other problems as it probably is the most promising idea that can be listed as contribution of this thesis.

Bibliography

[1] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern classification*. John Wiley & Sons, 2012.

[2] Michael A. Nielsen. *Neural networks and deep learning*, volume 25. Determination press USA, 2015.

[3] Eric M. Hutchins, Michael J. Cloppert, and Rohan M. Amin. Seven ways to apply the cyber kill chain with a threat intelligence platform. *Leading Issues in Information Warfare & Security Research*, 1(1):80, 2014.

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

[5] Gary Schneider, Jessica Evans, and Katherine Pinard. *The Internet-Illustrated*. Nelson Education, 2009.

[6] Fred Cohen. Computer viruses. *Computers & security*, 6(1):22–35, 1987.

[7] Fred Cohen. Computer viruses, 1985.

[8] Tim Berners-Lee. World wide web, 1992.

[9] McAfee labs threat report december 2017. page 13, 2017.

[10] Evan Perez and Daniella Diaz. White house announces retaliation against russia: Sanctions, ejecting diplomats, 2016.

[11] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection. 2008.

[12] Bum Jun Kwon, Jayanta Mondal, Jiyong Jang, Leyla Bilge, and Tudor Dumitras. The dropper effect: Insights into malware distribution with downloader graph analytics. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1118–1129. ACM, 2015.

[13] Gavin O'Gorman and Geoff McDonald. *Ransomware: A growing menace*. Symantec Corporation, 2012.

[14] Hamad Binsalleeh, Thomas Ormerod, Amine Boukhtouta, Prosenjit Sinha, Amr Youssef, Mourad Debbabi, and Lingyu Wang. On the analysis of the zeus botnet crimeware toolkit. In *Privacy Security and Trust (PST), 2010 Eighth Annual International Conference on*, pages 31–38. IEEE, 2010.

[15] Ralph Langner. To kill a centrifuge: a technical analysis of what stuxnet's creators tried to achieve. 2013.

[16] Jonathan Berr. Wannacry ransomware attack losses could reach $4 billion, 2017$.

[17] Ellen Nakashima. Russian military was behind tpetyayberattack in ukraine, cia concludes, 2018.

[18] Lucky Onwuzurike, Mario Almeida, Enrico Mariconti, Jeremy Blackburn, Gianluca Stringhini, and Emiliano De Cristofaro. A family of droids: Analyzing behavioral model based android malware detection via static and dynamic analysis. *arXiv preprint arXiv:1803.03448*, 2018.

[19] Mario Almeida, Muhammad Bilal, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Matteo Varvello, and Jeremy Blackburn. CHIMP: Crowdsourc-

ing human inputs for mobile phones. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, pages 45–54. International World Wide Web Conferences Steering Committee, 2018.

[20] Jeremiah Onaolapo, Enrico Mariconti, and Gianluca Stringhini. What happens after you are pwnd: Understanding the use of leaked webmail credentials in the wild. In *Proceedings of the 2016 Internet Measurement Conference*, pages 65–79. ACM, 2016.

[21] Jeremiah Onaolapo, Enrico Mariconti, and Gianluca Stringhini. Harvesting wild honey from webmail beehives. In *Engineering Secure Software and Systems (ESSoS)*, 2016.

[22] Enrico Mariconti, Guillermo Suarez-Tangil, Jeremy Blackburn, Emiliano De Cristofaro, Nicolas Kourtellis, Ilias Leontiadis, Jordi Luque Serrano, and Gianluca Stringhini. " you know what to do": Proactive detection of YouTube videos targeted by coordinated hate attacks. *arXiv preprint arXiv:1805.08168*, 2018.

[23] Enrico Mariconti, Jeremiah Onaolapo, Syed Sharique Ahmad, Nicolas Nikiforou, Manuel Egele, Nick Nikiforakis, and Gianluca Stringhini. Why allowing profile name reuse is a bad idea. In *Proceedings of the 9th European Workshop on System Security*, page 3. ACM, 2016.

[24] Enrico Mariconti, Jeremiah Onaolapo, Syed Sharique Ahmad, Nicolas Nikiforou, Manuel Egele, Nick Nikiforakis, and Gianluca Stringhini. What's in a name?: Understanding profile name reuse on twitter. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1161–1170. International World Wide Web Conferences Steering Committee, 2017.

[25] Enrico Mariconti, Jeremiah Onaolapo, Gordon Ross, and Gianluca Stringhini. A methodology to assess malware causality in network activities. In *Engineering Secure Software and Systems (ESSoS)*, 2016.

[26] Enrico Mariconti, Jeremiah Onaolapo, Gordon Ross, and Gianluca Stringhini. The cause of all evils: Assessing causality between user actions and malware activity. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, volume 10. USENIX, 2017.

[27] Enrico Mariconti, Jeremiah Onaolapo, Gordon Ross, and Gianluca Stringhini. What's your major threat? on the differences between the network behavior of targeted and commodity malware. In *Availability, Reliability and Security (ARES), 2016 11th International Conference on*, pages 599–608. IEEE, 2016.

[28] Michael Weissbacher, Enrico Mariconti, Guillermo Suarez-Tangil, Gianluca Stringhini, William Robertson, and Engin Kirda. Ex-ray: Detection of history-leaking browser extensions. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 590–602. ACM, 2017.

[29] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[30] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. MaMaDroid: Detecting android malware by building markov chains of behavioral models (extended version). *arXiv preprint arXiv:1711.07477*, 2017.

[31] Wentao Chang, An Wang, Aziz Mohaisen, and Songqing Chen. Characterizing botnets-as-a-service. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 585–586. ACM, 2014.

[32] Eric M Hutchins, Michael J Cloppert, and Rohan M Amin. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. page 14, 2011.

[33] Sewall Wright. Correlation and causation. *Journal of agricultural research*, 20(7):557–585, 1921.

[34] Tyler Vigen. *Spurious correlations.* Hachette books, 2015.

[35] Karl Pearson. Mathematical contributions to the theory of evolution.—on a form of spurious correlation which may arise when indices are used in the measurement of organs. *Proceedings of the royal society of london*, 60(359-367):489–498, 1897.

[36] David Hume. An enquiry concerning human understanding. In *Seven Masterpieces of Philosophy*, pages 191–284. Routledge, 2016.

[37] David Lewis. Counterfactuals and comparative possibility. In *Ifs*, pages 57–85. Springer, 1973.

[38] Judea Pearl. *Causality.* Cambridge university press, 2009.

[39] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[40] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 1997-12-01.

[41] Tin Kam Ho. Random decision forests. In *Document analysis and recognition, 1995., proceedings of the third international conference on*, volume 1, pages 278–282. IEEE, 1995.

[42] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[43] Leo Breiman. Using adaptive bagging to debias regressions, 1999.

[44] Thomas G. Dietterich. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine learning*, 40(2):139–157, 2000.

[45] Leo Breiman. *Classification and regression trees.* Routledge, 2017.

[46] R. Behi and M. Nolan. Causality and control: key to the experiment. *British Journal of Nursing (Mark Allen Publishing)*, 5(4):252–255, March 1996.

[47] Xu Chen, Ming Zhang, Zhuoqing Morley Mao, and Paramvir Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *OSDI*, volume 8, pages 117–130, 2008.

[48] Ali Zand, Giovanni Vigna, Richard Kemmerer, and Christopher Kruegel. Rippler: Delay injection for service dependency detection. In *INFOCOM, 2014 Proceedings IEEE*, pages 2157–2165. IEEE, 2014.

[49] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. Mast: Triage for market-scale mobile malware analysis. In *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*, pages 13–24. ACM, 2013.

[50] Jiyong Jang, David Brumley, and Shobha Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 309–320. ACM, 2011.

[51] Dhilung Kirat, Lakshmanan Nataraj, Giovanni Vigna, and B. S. Manjunath. Sigmal: A static signal processing based malware triage. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 89–98. ACM, 2013.

[52] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. Springer, 2008.

[53] Samuel T. King, Zhuoqing Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. Enriching intrusion alerts through multi-host causality. In *NDSS*, 2005.

[54] Hao Zhang, Danfeng Daphne Yao, and Naren Ramakrishnan. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 39–50. ACM, 2014.

[55] Jim Freer, Keith Beven, and Bruno Ambroise. Bayesian estimation of uncertainty in runoff prediction and the value of data: An application of the GLUE approach. *Water Resources Research*, 32(7):2161–2173, 1996.

[56] William R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3):285–294, 1933.

[57] Eero P. Simoncelli and Edward H. Adelson. Noise removal via bayesian wavelet coring. In *Proceedings of the 3rd IEEE International Conference on Image Processing*, volume 1, 1996.

[58] Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. In *Advances in neural information processing systems*, pages 2249–2257, 2011.

[59] Stefan Heule, Devon Rifkin, Alejandro Russo, and Deian Stefan. The most dangerous code in the browser. page 7, 2015.

[60] Frans Rosand Linus Sd. Chrome extensions - aka total absence of privacy, 2015.

[61] Oleksii Starov and Nick Nikiforakis. Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1481–1490. International World Wide Web Conferences Steering Committee, 2017.

[62] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. In *2011 IEEE Symposium on Security and Privacy*, pages 115–130. IEEE, 2011-05.

[63] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. *Usenix Annual Technical Conference (ATC)*, 2007.

[64] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting malicious behavior in browser extensions. 2014.

[65] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon Mccoy, Antonio Nappa, Vern Paxson, Paul Pearce, Niels Provos, and Moheeb Abu Rajab. Ad injection at scale: Assessing deceptive advertisement modifications. In *2015 IEEE Symposium on Security and Privacy*, pages 151–167. IEEE, 2015-05.

[66] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security - CCS '12*, page 736. ACM Press, 2012.

[67] Jonathan R. Mayer and John C. Mitchell. Third-party web tracking: Policy and technology. In *2012 IEEE Symposium on Security and Privacy*, pages 413–427. IEEE, 2012-05.

[68] Franziska Roesner, Kohno Tadayoshi, and David Wetherall. Detecting and defending against third-party tracking. *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[69] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. page 18, 2016.

[70] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting privacy leaks in iOS applications. page 15, 2011.

[71] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Trust and Trustworthy Computing*, volume 7344, pages 291–307. Springer Berlin Heidelberg, 2012.

[72] Chaz Lever, Platon Kotzias, Davide Balzarotti, Juan Caballero, and Manos Antonakakis. A lustrum of malware network communication: Evolution and

insights. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 788–804. IEEE, 2017-05.

[73] Yun Shen, Enrico Mariconti, Pierre-Antoine Vervier, and Gianluca Stringhini. Tiresias: Predicting security events through deep learning. *Conference on Computer and Communication Systems (CCS)*, page 14, 2018.

[74] Kyle Soska and Nicolas Christin. Automatically detecting vulnerable websites before they turn malicious. In *USENIX Security Symposium*, pages 625–640, 2014.

[75] Leyla Bilge, Yufei Han, and Matteo Dell'Amico. RiskTeller: Predicting the risk of cyber incidents. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1299–1311. ACM, 2017.

[76] Yang Liu, Armin Sarabi, Jing Zhang, Parinaz Naghizadeh, Manish Karir, Michael Bailey, and Mingyan Liu. Cloudy with a chance of breach: Forecasting cyber security incidents. In *USENIX Security Symposium*, pages 1009–1024, 2015.

[77] Yang Liu, Jing Zhang, Armin Sarabi, Mingyan Liu, Manish Karir, and Michael Bailey. Predicting cyber security incidents using feature-based characterization of network-level malicious activities. In *Proceedings of the 2015 ACM International Workshop on International Workshop on Security and Privacy Analytics*, pages 3–9. ACM, 2015.

[78] Carl Sabottke, Octavian Suciu, and Tudor Dumitras. Vulnerability disclosure in the age of social media: Exploiting twitter for predicting real-world exploits. In *USENIX Security Symposium*, pages 1041–1056, 2015.

[79] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *USENIX Security Symposium*, pages 611–626, 2015.

[80] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *Proceedings of the 26th USENIX Conference on Security Symposium, Security*, volume 17, 2017.

[81] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298. ACM, 2017.

[82] William Melicher, Blase Ur, Sean M. Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Fast, lean, and accurate: Modeling password guessability using neural networks. In *USENIX Security Symposium*, pages 175–191, 2016.

[83] Razvan Pascanu, Jack W. Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas. Malware classification with recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 1916–1920. IEEE, 2015.

[84] Denis Maslennikov. Mobile malware evolution: An overview, part 4, 2011.

[85] Denis Maslennikov. Mobile malware evolution, part 5, 2012.

[86] McAfee. McAfee mobile threat report q1, 2018. page 16, 2018.

[87] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting android applications to java bytecode. In *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*, page 6. ACM, 2012.

[88] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[89] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245. ACM, 2009.

[90] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294. ACM, 2012.

[91] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

[92] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[93] 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*, pages 1–6. ACM, 2014.

[94] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[95] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD Center. ScanDal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 12:110, 2012.

[96] Lok-Kwong Yan and Heng Yin. DroidScope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *USENIX security symposium*, pages 569–584, 2012.

[97] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[98] Vaibhav Rastogi, Yan Chen, and William Enck. AppsPlayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 209–220. ACM, 2013.

[99] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In *NDSS*, volume 25, pages 50–52, 2012.

[100] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015.

[101] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 347–356. ACM, 2010.

[102] Michelle Y. Wong and David Lie. IntelliDroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, volume 16, pages 21–24, 2016.

[103] Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. DyTa: dynamic symbolic execution guided with static verification results. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 992–994. ACM, 2011.

[104] Yajin Zhou Xuxian Jiang and Zhou Xuxian. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[105] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 899–914. IEEE, 2015.

[106] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. Brahmastra: Driving apps to test the security of third-party components. In *USENIX Security Symposium*, pages 1021–1036, 2014.

[107] Shuaifu Dai, Alok Tongaonkar, Xiaoyin Wang, Antonio Nucci, and Dawn Song. NetworkProfiler: Towards automatic fingerprinting of android apps. In *INFOCOM*, volume 13, pages 809–817, 2013.

[108] Gerardo Canfora, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Acquiring and analyzing app metrics for effective mobile malware detection. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, pages 50–57. ACM, 2016.

[109] Sen Chen, Minhui Xue, Zhushou Tang, Lihua Xu, and Haojin Zhu. Stormdroid: A streaminglized machine learning-based system for detecting android malware. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 377–388. ACM, 2016.

[110] Hanlin Zhang, Yevgeniy Cole, Linqiang Ge, Sixiao Wei, Wei Yu, Chao Lu, Genshe Chen, Dan Shen, Erik Blasch, and Khanh D. Pham. ScanMe mobile: a cloud-based android malware analysis service. *ACM SIGAPP Applied Computing Review*, 16(1):36–49, 2016.

[111] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of computer security*, 6(3):151–180, 1998.

[112] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *USENIX security symposium*, volume 4, pages 351–366, 2009.

[113] Madhu K. Shankarapani, Subbu Ramamoorthy, Ram S. Movva, and Srinivas Mukkamala. Malware detection using assembly and API call sequences. *Journal in computer virology*, 7(2):107–119, 2011.

[114] Gerardo Canfora, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Detecting android malware using sequences of system calls. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, pages 13–20. ACM, 2015.

[115] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[116] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, page 68. ACM, 2013.

[117] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 59. ACM, 2012.

[118] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[119] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458. ACM, 2014.

[120] Justin Sahs and Latifur Khan. A machine learning approach to android malware detection. In *Intelligence and security informatics conference (eisic), 2012 european*, pages 141–147. IEEE, 2012.

[121] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69. IEEE, 2012.

[122] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013.

[123] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *European symposium on research in computer security*, pages 163–182. Springer, 2014.

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

[125] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 303–313. IEEE Press, 2015.

[126] Joshua Garcia, Mahmoud Hammad, Bahman Pedrood, Ali Bagheri-Khaligh, and Sam Malek. Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. *Department of Computer Science, George Mason University, Tech. Rep*, 2015.

[127] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. DREBIN: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[128] Yang Chen, Mo Ghorbanzadeh, Kevin Ma, Charles Clancy, and Robert McGwier. A hidden markov model detection of malicious android applications at runtime. In *Wireless and Optical Communication Conference (WOCC), 2014 23rd*, pages 1–6. IEEE, 2014.

[129] Gerardo Canfora, Francesco Mercaldo, and Corrado Aaron Visaggio. An hmm and structural entropy based detector for android malware: An empirical study. *Computers & Security*, 61:1–18, 2016.

[130] John P. John, Alexander Moshchuk, Steven D. Gribble, and Arvind Krishnamurthy. Studying spamming botnets using botlab. In *NSDI*, volume 9, pages 291–306, 2009.

[131] Christian Rossow, Christian J. Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 65–79. IEEE, 2012.

[132] Alberto Ortega. Pafish, 2013.

[133] Steven L. Scott. A modern bayesian look at the multi-armed bandit. *Applied Stochastic Models in Business and Industry*, 26(6):639–658, 2010.

[134] Jing Zhang, Zakir Durumeric, Michael Bailey, Mingyan Liu, and Manish Karir. On the mismanagement and maliciousness of networks. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[135] David A. Harville. Bayesian inference for variance components using only error contrasts. *Biometrika*, 61(2):383–385, 1974.

[136] Roberto Trotta. Bayes in the sky: Bayesian inference and model selection in cosmology. *Contemporary Physics*, 49(2):71–104, 2008.

[137] Hamad Binsalleeh, Thomas Ormerod, Amine Boukhtouta, Prosenjit Sinha, Amr Youssef, Mourad Debbabi, and Lingyu Wang. On the analysis of the zeus botnet crimeware toolkit. In *Privacy Security and Trust (PST), 2010 Eighth Annual International Conference on*, pages 31–38. IEEE, 2010.

[138] John Aycock. *Spyware and Adware*, volume 50. Springer Science & Business Media, 2010.

[139] Xinyu Xing, Wei Meng, Byoungyoung Lee, Udi Weinsberg, Anmol Sheth, Roberto Perdisci, and Wenke Lee. Understanding malvertising through ad-injecting browser extensions. In *Proceedings of the 24th international conference on world wide web*, pages 1286–1295. International World Wide Web Conferences Steering Committee, 2015.

[140] Sajjad Arshad, Amin Kharraz, and William Robertson. Identifying extension-based ad injection via fine-grained web content provenance. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 415–436. Springer, 2016.

[141] George AF Seber and Alan J Lee. *Linear Regression Analysis*. John Wiley & Sons, 2012.

[142] S. I. V. Sousa, F. G. Martins, M. C. M. Alvim-Ferraz, and M. C. Pereira. Multiple linear regression and artificial neural networks based on principal components to predict ozone concentrations. *Environmental Modelling & Software*, 22(1):97–103, 2007.

[143] Alex J Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, pages 199–222, 2004.

[144] Lorrie Faith Cranor, Joseph Reagle, and Mark S. Ackerman. Beyond concern: Understanding net users' attitudes about online privacy. *The Internet*

upheaval: raising questions, seeking answers in communications policy, pages 47–70, 2000.

[145] Naresh K. Malhotra, Sung S. Kim, and James Agarwal. Internet users' information privacy concerns (IUIPC): The construct, the scale, and a causal model. *Information systems research*, 15(4):336–355, 2004.

[146] Ping Chen, Lieven Desmet, and Christophe Huygens. A study on advanced persistent threats. In *IFIP International Conference on Communications and Multimedia Security*, pages 63–72. Springer, 2014.

[147] Gianluca Stringhini and Olivier Thonnard. That ain't you: Blocking spearphishing through behavioral modelling. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 78–97. Springer, 2015.

[148] Stevens Le Blond, Adina Uritesc, Cédric Gilbert, Zheng Leong Chua, Prateek Saxena, and Engin Kirda. A look at targeted attacks through the lense of an NGO. In *USENIX Security Symposium*, pages 543–558, 2014.

[149] 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*, pages 281–290. ACM, 2010.

[150] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. The ghost in the browser: Analysis of web-based malware. *HotBots*, 7:4–4, 2007.

[151] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *USENIX Security Symposium*, 2003.

[152] Brown Farinholt, Mohammad Rezaeirad, Paul Pearce, Hitesh Dharmdasani, Haikuo Yin, Stevens Le Blond, Damon McCoy, and Kirill Levchenko. To catch a ratter: Monitoring the behavior of amateur darkcomet rat operators in the

wild. In *2017 38th IEEE Symposium on Security and Privacy (SP)*, pages 770–787. Ieee, 2017.

[153] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. Your botnet is my botnet: analysis of a botnet takeover. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 635–647. ACM, 2009.

[154] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. Cutting the gordian knot: A look under the hood of ransomware attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2015.

[155] Eugene Kolodenker, William Koch, Gianluca Stringhini, and Manuel Egele. PayBreak: defense against cryptographic ransomware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 599–611. ACM, 2017.

[156] Grant Ho, Aashish Sharma Mobin Javed, Vern Paxson, and David Wagner. Detecting credential spearphishing attacks in enterprise settings. In *Proceedings of the 26rd USENIX Security Symposium (USENIX Security'17)*, pages 469–485, 2017.

[157] Guofei Gu, Phillip A. Porras, Vinod Yegneswaran, Martin W. Fong, and Wenke Lee. Bothunter: Detecting malware infection through ids-driven dialog correlation. In *USENIX Security Symposium*, volume 7, pages 1–16, 2007.

[158] Christopher Kruegel and Giovanni Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 251–261. ACM, 2003.

[159] Gianluca Stringhini, Yun Shen, Yufei Han, and Xiangliang Zhang. Marmite: spreading malicious file reputation through download graphs. In *Proceedings*

of the 33rd Annual Computer Security Applications Conference, pages 91–102. ACM, 2017.

[160] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 1999 IEEE symposium on security and privacy (Cat. No. 99CB36344)*, pages 133–145. IEEE, 1999.

[161] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M. Rush. Character-aware neural language models. In *AAAI*, pages 2741–2749, 2016.

[162] Michiel Hermans and Benjamin Schrauwen. Training and analysing deep recurrent neural networks. In *Advances in neural information processing systems*, pages 190–198, 2013.

[163] Saizheng Zhang, Yuhuai Wu, Tong Che, Zhouhan Lin, Roland Memisevic, Ruslan R. Salakhutdinov, and Yoshua Bengio. Architectural complexity measures of recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 1822–1830, 2016.

[164] Kamil Rocki. Recurrent memory array structures. *arXiv preprint arXiv:1607.03085*, 2016.

[165] James R. Norris. *Markov chains*. Number 2. Cambridge university press, 1998.

[166] Peter F. Brown, Peter V. Desouza, Robert L. Mercer, Vincent J. Della Pietra, and Jenifer C. Lai. Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–479, 1992.

[167] Denis Arrivault, Dominique Benielli, Francois Denis, and Remi Eyraud. Sp2learn: A toolbox for the spectral learning of weighted automata. In *International Conference on Grammatical Inference*, pages 105–119, 2017.

[168] Kamvar Kamvar, Sepandar Sepandar, Klein Klein, Dan Dan, Manning Manning, and Christopher Christopher. Spectral learning. In *International Joint Conference of Artificial Intelligence*. Stanford InfoLab, 2003.

[169] Gary M. Weiss and Haym Hirsh. Learning to predict rare events in event sequences. In *KDD*, pages 359–363, 1998.

[170] Jianxin Wu, James M. Rehg, and Matthew D. Mullin. Learning a rare event detection cascade by direct feature selection. In *Advances in Neural Information Processing Systems*, pages 1523–1530, 2004.

[171] Junseok Kwon and Kyoung Mu Lee. A unified framework for event summarization and rare event detection. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 1266–1273. IEEE, 2012.

[172] Lukasz Kaiser, Ofir Nachum, Aurko Roy, and Samy Bengio. Learning to remember rare events. *arXiv preprint arXiv:1703.03129*, 2017.

[173] Iasonas Polakis, Michalis Diamantaris, Thanasis Petsas, Federico Maggi, and Sotiris Ioannidis. Powerslave: analyzing the energy consumption of mobile antivirus software. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 165–184. Springer, 2015.

[174] Jon Oberheide and Charlie Miller. Dissecting the android bouncer. *SummerCon2012, New York*, 2012.

[175] Google Play has hundreds of Android apps that contain malware. `http://www.trustedreviews.com/news/malware-apps-downloaded-google-play`, 2016.

[176] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Android permissions: a perspective combining risks and benefits. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pages 13–22. ACM, 2012.

[177] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[178] Ryan De Souza. Ghost push android malware responsible for infecting 600k new users daily, 2015.

[179] Patrick Schulz. Code protection in android. *Insititute of Computer Science, Rheinische Friedrich-Wilhelms-Universitgt Bonn, Germany*, 110, 2012.

[180] Ian Jolliffe. Principal component analysis. In *International encyclopedia of statistical science*, pages 1094–1096. Springer, 2011.

[181] Marti A. Hearst, Susan T. Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. Support vector machines. *IEEE Intelligent Systems and their applications*, 13(4):18–28, 1998.

[182] Simon Bernard, Sébastien Adam, and Laurent Heutte. Using random forests for handwritten digit recognition. In *Document Analysis and Recognition, 2007. ICDAR 2007. Ninth International Conference on*, volume 2, pages 1043–1047. IEEE, 2007.

[183] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *ACM SIGMETRICS Performance Evaluation Review*, volume 42, pages 221–233. ACM, 2014.

[184] Anthony Desnos. Androguard-reverse engineering, malware and goodware analysis of android applications, 2013.

[185] Panagiotis Andriotis, Martina Angela Sasse, and Gianluca Stringhini. Permissions snapshots: Assessing users' adaptation to the android runtime permission model. In *Information Forensics and Security (WIFS), 2016 IEEE International Workshop on*, pages 1–6. IEEE, 2016.

[186] Xuxian Jiang and Yajin Zhou. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109. IEEE, 2012.

[187] AppBrain Stats. Number of android applications. 2014.

[188] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS*, volume 14, pages 23–26, 2014.