FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Malware Through the Looking Glass:

Malware Analysis in an Evolving Threat Landscape

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doctor of Technical Sciences

by

Dipl.-Ing. Martina Lindorfer BSc
Matriculation Number 0626770

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Priv.-Doz. Dipl.-Ing. Mag.rer.soc.oec. Dr. techn. Edgar Weippl

The dissertation has been reviewed by:

Thorsten Holz

Engin Kirda

Vienna, November 26, 2015

Martina Lindorfer

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Malware Through the Looking Glass:

Malware Analysis in an Evolving Threat Landscape

DISSERTATION

zur Erlangung des akademischen Grades

Doktorin der Technischen Wissenschaften

eingereicht von

Dipl.-Ing. Martina Lindorfer BSc
Matrikelnummer 0626770

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Priv.-Doz. Dipl.-Ing. Mag.rer.soc.oec. Dr. techn. Edgar Weippl

Diese Dissertation haben begutachtet:

Thorsten Holz

Engin Kirda

Wien, 26. November 2015

Martina Lindorfer

Declaration of Authorship

Dipl.-Ing. Martina Lindorfer BSc
Margaretenstrasse 145, 1050 Vienna

I hereby declare that I have written this thesis independently, that I have fully specified all external sources and resources, and that I have properly marked all parts that are not my own—including tables, maps and figures—, which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, November 26, 2015

Martina Lindorfer

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Martina Lindorfer BSc
Margaretenstrasse 145, 1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit—einschließlich Tabellen, Karten und Abbildungen—, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 26. November 2015

Martina Lindorfer

Abstract

Malware has become a multi-million dollar industry and is the basis of many forms of cybercrime, including information theft, spam emails, denial of service attacks, fake antivirus scams, and ransomware. Motivated by financial gains, malware authors are constantly evolving their code to increase their profit by evading security defenses and exploiting new monetization techniques.

Developing effective and efficient analysis methods is an arms race against malware authors. Large-scale dynamic analysis sandboxes, which execute samples in a controlled environment and monitor their behavior, are the de facto standard for the automated analysis of malware. However, one current challenge is that malware authors extensively use polymorphism and overwhelm analysis systems with an increasing number of malware samples, which are mostly repacked versions of already known malware. We develop novel techniques to compare multiple versions of self-updating malware and quantify their code differences. By associating code changes of functional components with the high-level behavior that they are implementing, we can observe the evolution of a malware family and highlight interesting components for further analysis.

With the emergence of mobile platforms, like smartphones and tablets, malware has spread to these devices as well. Mobile devices provide new ways for monetization, such as premium SMS services, advertisement fraud, and circumventing SMS-based security features for mobile banking. In addition, the possibility to surveil and track users through a wealth of sensor data and personal information stored on this devices makes them attractive targets. However, mobile devices also pose unique challenges for building malware defenses, for example because of limiting the capabilities of on-device defenses in terms of resources and permissions. We build a large-scale public analysis sandbox for Android apps, called Andrubis, as a cloud-based service, to analyze and understand Android malware. During its first two years of operation, Andrubis collected a large and diverse dataset of over one million Android apps, which we leverage to gain insights into the behavior and evolution of Android malware. Furthermore, we use machine learning techniques to build a robust classifier that can automatically distinguish benign from malicious apps with high accuracy based on features extracted during static and dynamic analysis with Andrubis.

Finally, mobile platforms led to the emergence of application markets as the main app distribution channel. We study how these markets are used to distribute malware and present AndRadar, an Android market radar for the fast discovery of Android malware in alternative application markets. As a result, we gain important insights into publishing strategies of malicious app authors and can facilitate fast remediation procedures in markets.

Kurzfassung

Malware hat sich zu einer millionenschweren Industrie entwickelt und ist die Grundlage für viele Formen der Internetkriminalität, wie zum Beispiel Datendiebstahl, Spam E-Mails, Denial-of-Service Angriffe, gefälschte Anti-Viren Programme und Ransomware. Motiviert durch finanzielle Profite entwickeln Autoren von Schadsoftware ihren Code ständig weiter um Sicherheitsmaßnahmen zu umgehen und um neue gewinnbringende Funktionalität zu implementieren.

Die Entwicklung von effektiven und effizienten Analyseverfahren ist jedoch ein ständiges Wettrüsten zwischen Sicherheitsforschern und den Autoren von Malware. Dynamische Analyse-Sandboxen, die Programme in einer kontrollierten Umgebung ausführen um ihr Verhalten zu beobachten, sind der De-facto-Standard für die automatisierte Analyse von Malware. Eine aktuelle Herausforderung ist jedoch, dass es sich beim Großteil der zu analysierenden Programme um polymorphe und neu gepackte Versionen von schon bekannter Schadsoftware handelt. Daher entwickeln wir neue Techniken, um mehrere Versionen von sich selbst aktualisierender Malware zu vergleichen und Unterschiede im Code zu quantifizieren. Zusätzlich weisen wir das während der dynamischen Analyse aufgezeichnete Verhalten den Änderungen im Code auf Ebene von funktionalen Komponenten, die dieses implementieren, zu. Hiermit können wir die Entwicklung einer Malware-Familie beobachten, und interessante Komponenten, die signifikante Codeänderungen aufweisen oder neues Verhalten implementieren, automatisch erkennen und für die weitere manuelle Analyse priorisieren.

Mit der steigenden Beliebtheit von mobilen Plattformen wie Smartphones und Tablets werden auch diese immer mehr zum Ziel von Malware. Für Angreifer bieten mobile Geräte attraktive neuartige Angriffsmöglichkeiten, wie zum Beispiel Premium SMS-Dienste, Betrug mit Werbeanzeigen und die Aushebelung von SMS-basierte Sicherheitsfunktionen für mobiles Banking. Darüber hinaus können Benutzer durch eine Vielzahl von Sensordaten und auf den mobilen Geräten gespeicherten persönlichen Daten überwacht werden. Entwickler von Sicherheitsmaßnahmen stehen vor neuen Herausforderungen, da die Ressourcen und Berechtigungen von Apps auf mobilen Geräten üblicherweise stark eingeschränkt sind. Wir entwickeln eine Analyse-Sandbox für Android Apps, genannt Andrubis, als Cloud-basierten Dienst um Android Malware zu analysieren und zu verstehen. Andrubis erlaubt uns anhand eines im Laufe von zwei Jahren gesammelten vielfältigen Datensatzes von über einer Million Android Apps Einblicke in das Verhalten und die Entwicklung von Android Malware zu gewinnen. Darüber hinaus verwenden wir Machine Learning, um basierend auf den während der statischen und dynamischen Analyse mit Andrubis extrahierten Features, automatisch gutartige von bösartigen Apps mit hoher Genauigkeit zu unterscheiden.

Eine weitere Eigenheit von mobilen Plattformen ist die Verbreitung von Apps über sogenannte App Stores. Wir untersuchen, inwieweit diese Märkte für die Verbreitung von Malware verwendet werden und entwickeln AndRadar, ein Android-“Marktradar” mit dem wir schnell und effizient Android Malware in alternativen App Stores aufspüren können. Als Resultat gewinnen wir wichtige Einblicke in das Publikationsverhalten von Malware-Autoren und erleichtern die Entfernung von Malware aus den Märkten.

Acknowledgements

I am deeply grateful to my advisor Edgar Weippl, who made this thesis possible and allowed me the freedom to pursue my own research ideas. I owe my gratitude to everybody at the Secure Systems Lab and the Automation Systems Group for an awesome work environment despite the occasional administrative challenges of being a security lab without a security professor. It was an incredible learning experience and an honor working with many great researchers. A special thanks goes to Matthias Neugschwandtner, Gilbert Wondracek, Christian Platzer, Clemens Kolbitsch, Paolo Milani Comparetti, and Lukas Weichselbaum for their collaboration, valuable feedback, and many insightful discussions. I also would like to thank SBA Research for all the opportunities I received and my colleagues for providing a great research environment during the last stages of this thesis. In particular, Adrian Dabrowski and Cathy Krombholz, arigatou gozaimasu.

I also had the pleasure to work with great researchers from other institutions. My sincere thanks go to Engin Kirda and William Robertson, who provided me an opportunity to join the Systems Security Lab at Northeastern University for a semester. I would further like to thank Alessandro di Federico, Victor van der Veen, Yanick Fratantonio, Alessandro Sisto, Stamatis Volanis, Elias Athanasopoulos, Federico Maggi, Stefano Zanero, and Sotiris Ioannidis for their support and contributions to this thesis.

This thesis further would not have been possible without my parents Rita and Johann and their constant support throughout my life. My thanks go to my friends, in particular Karin Maier, Florian Meyer, Clemens Peither, and Patrick Wolowicz, for their patience and moral support throughout this thesis—and for the occasional zombie movie night and bacon experiments. I also would like to thank Kevin Borgolte for his love, encouragement, and all the extra nit-picky comments during the creation of this thesis.

Lastly, I wish to thank coffee, coffee, coffee, more coffee, and coffee.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Malware, short for malicious software, has been a serious threat to Internet user's security and privacy for more than a decade. Today, malware is the basis for many forms of cybercrime and it has evolved into a fully-fledged *malicious software industry.* On the one hand, a flourishing underground economy allows miscreants to deal with goods and services such as malware creation kits and components, stolen credentials and credit card information, or the rental of bot networks for sending spam e-mails and performing denial of service (DoS) attacks. On the other hand, specialized services streamline the malware distribution process by selling malware installations, for example through drive-by downloads or malware droppers [40, 95]. Consequently, malware has been growing increasingly more sophisticated as malware authors have become strongly motivated by financial gains [84,230]. In order to maximize their profits, in what now is a multi-million dollar industry [83,189], malware authors are driven by two main goals: (1) staying under the radar of security companies and researchers by evading and defeating their defenses, and (2) exploring new avenues for monetization, such as, scaring victims into buying fake antivirus software, or extorting money from infected victims with ransomware.

A large body of related work has focused on developing detection techniques for Windows malware. Due to the vast amount of new samples being released in the wild—PandaLabs estimated 225,000 new Windows malware samples *per day* in the first quarter of 2015 [151]—, researchers and antivirus (AV) vendors have to rely on automated analysis tools to distinguish between malware and goodware. Large-scale dynamic malware analysis sandboxes, such as Anubis [32, 34] and CWSandbox [211], which automatically execute samples in a controlled environment and monitor their behavior, have become the de facto standard for the automated analysis of malware. However, developing effective and efficient analysis methods is an arms race against malware developers: First, malware authors try to detect and evade analysis sandboxes with environment-sensitive malware samples [51,120] to stay undetected for as long as possible. Second, malware authors overwhelm analysis systems with an increasing number of malware samples. In an effort to evade detection by signature-based AV scanners, malware authors extensively use polymorphism and frequently repack their malware samples, in the worst case on demand for every download [40, 171]. As a consequence, finding interesting samples that implement new functionally and that are not just repacked versions of existing malware is like finding a needle in a haystack.

In recent years, mobile computing platforms, i.e., smartphones, and Android as the dominant operating system, have been rising in popularity dramatically. Unsurprisingly, malware authors are trying to cash in on this trend. Compared to desktop platforms, smartphones provide many new and unique ways of monetization, such as premium SMS services, SMS payment systems, advertisement fraud, and the circumvention of two-factor authentication by intercepting tokens and mobile TANs. In addition, the possibility to surveil and track users through a wealth of sensor data, e.g., from GPS, camera, and microphone, and the personally identifiable information (PII) stored on them, makes smartphones attractive targets. Especially advertisement libraries—the primary source of revenue for the majority of smartphone app developers—extensively collect PII, in turn causing serious privacy and security concerns [92]. Furthermore, smartphones introduced a new software distribution model, with the vast majority of apps being downloaded from official and alternative application markets, providing new ways for malware to be distributed.

Even though several AV companies offer solutions for mobile devices, traditional AV techniques that perform behavior-based malware tracking on-device are impossible to implement, as the limited resources and privileges on mobile devices restrict their functionality [53, 149]. As a consequence, users largely have to rely on the trustworthiness of the source when installing apps and depend on application markets to perform verification of apps at the market level. While the details of how the official Google Play Store verifies apps are unknown, related work [148, 154] and numerous instances of malware [24, 56, 100, 161, 188] have shown that its vetting process can be bypassed. In the case of hundreds of alternative application markets it is even more unclear if and how they check submitted apps for malicious behavior before they are published. While estimations of new Android samples are far behind those of their Windows counterparts, with only several thousand new samples released each day [192], the lack of efficient on-device malware defenses and the sheer size and number of application markets makes the need for automated analysis methods for Android apps apparent.

1.2 Aim of this Work

Looking at the constant advancement of malware techniques as a form of *malware evolution*, we can view the arms race against security researchers as what is referred to in biology as the *Red Queen Problem* [99]: "Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!" [42], or in other words "for an evolutionary system, continuing development is needed just in order to maintain its fitness relative to the systems it is co-evolving with." [202]. Hence, the aim of our work is to extend existing malware analysis solutions and to find new malware analysis techniques to co-evolve with the next steps in the malware's evolution.

With Beagle (Chapter 2) we address the first challenge: shedding light on the general malware development lifecycle. Motivated by the observation that the majority of new malware samples are polymorphic and repacked versions of previous samples, we propose novel methods to quantify code and behavior changes between updated versions of malware samples. Analysis resources, human resources specifically, for the in-depth manual analysis of malware samples are scarce. In order to reduce manual analysis efforts we can guide human analysts towards samples of a malware family that implement significant updates or new functionality.

In order to deal with the emerging threat of Android malware, we build the Android app analysis sandbox Andrubis (Chapter 3). By integrating our solution into the public Windows binary analysis sandbox Anubis, we provide access to our analysis system to other researchers as well as end users, and allow it to be integrated into other tools and services. We can use Andrubis to collect a large-scale dataset of Android malware in the wild, which contrary to existing, static Android malware datasets reflects changes in the Android malware landscape. Furthermore, we enhance the behavioral analysis reports with an automated malware classification. For Marvin (Chapter 4) we experiment with machine learning classifiers to investigate whether such an approach is robust enough in practice to automatically distinguish between benign and malicious apps in the long run, i.e., over the evolution of malware.

Finally, an integral part of the malware lifecycle is its distribution. Given the dominant role of application markets in the Android ecosystem, and the wealth of metadata they provide about published apps, we investigate the distribution process of Android malware. Based on our findings, we devise methods for the fast discovery of malicious Android apps in markets with AndRadar (Chapter 5) to understand malware publishing patterns and facilitate fast remediation procedures.

1.3 Methodology

We demonstrate the practical feasibility of our approaches by presenting the design and prototype implementation of each approach. We further perform thorough evaluations using large-scale and real-world datasets, which include both malware currently found in the wild, as well as benign apps distributed in application markets. Furthermore, we provide our results to the research community through several means: (a) We provide all our datasets to other researchers (upon request), (b) we provide Andrubis and its malware classification extension Marvin through a public web interface, and (c) we provide statistics and insights from AndRadar's app discovery on a public website.

For Beagle we combine static and dynamic analysis techniques to highlight significant code and behavior changes between updated versions of the same malware family. We further propose a novel approach to continuously monitor specific variants of a malware family and taking advantage of their auto-update functionality to obtain updated versions of an original sample. We then use a binary similarity measure to quantify the difference between subsequent versions and to estimate the overall development effort for a sample. We further combine these results with observations of high-level behavior during dynamic analysis to augment code changes with information about the functional components that they are implementing. As a result we gain insights into the malicious software development process and can triage the most interesting versions of a sample for further analysis.

With Andrubis we built an Android app analysis sandbox, mainly based on dynamic analysis in an emulated environment, which allows a thorough investigation of Android apps' behavior on both the Dalvik VM and system level. In contrast to other approaches, we provide Andrubis as a public analysis sandbox, which allows us to collect a large and diverse dataset of both Android malware and benign apps. We use this dataset to characterize changes in Android malware behavior spanning four years, invalidating common held beliefs about conspicuous characteristic of Android malware and goodware.

We further leverage machine learning techniques for Marvin to effectively classify Android malware based on features extracted during static and dynamic analysis. We evaluate Marvin on a large-scale dataset of benign and malicious Android apps, and we show that it can be efficiently retrained and that our approach is applicable in practice in the long-term, over a period of more than a one and a half years from September 2012 to May 2014, without requiring any changes to the feature extraction process. Given the black-box nature of machine learning approaches, we also investigate the highest-ranked features used in the classification process to validate their meaningfulness and to understand how robust our system is to evasion through for example mimicry attacks.

Finally, for AndRadar we start with a thorough market study by crawling 8 alternative markets in their entirety, showing that they are indeed used to distribute malware. Motivated by our findings we then leverage the market-based distribution of Android apps to automatically collect malicious Android apps that are distributed in the wild through the Google Play Store and alternative markets. We use a novel approach to query markets based on a seed of known malicious apps, and we use lightweight identifiers to match apps in the seed against apps found in the markets with varying levels of confidence. Once we discover a malicious app in a market we further track updates to the app and its metadata to gain insights into the malware's lifecycle and its authors publishing strategy.

1.4 Structure of this Work

This thesis is organized as follows: Chapter 2 presents Beagle, a tool that identifies semantically interesting changes in functionality between related versions of malware samples to gain insights into the malicious software industry. This work was published under the title "*Lines of Malicious Code: Insights Into the Malicious Software Industry*" at the Annual Computer Security Applications Conference (ACSAC) in 2012 [119] and was carried out in collaboration with Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. In Chapter 3 we present Andrubis, an Android app analysis sandbox and our experiences with providing it as a public service since 2012, as well as our insights into the behavior of malicious Android apps, gained from a large-scale dataset of over one million Android apps that we collected during this time period. An early version describing the implementation of Andrubis was published as a technical report [210], the extended version including the large-scale evaluation was published as "*Andrubis – 1,000,000 Apps Later: A View on Current Android Malware Behaviors*" at the International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS) in 2014 [123]. Andrubis was implemented in collaboration with Lukas Weichselbaum, Matthias Neugschwandtner, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. As described in Chapter 4, we further extended Andrubis with Marvin, an automatic classification of malicious Android apps based on machine learning techniques, which we published under the title "*Marvin: Efficient and Comprehensive Mobile App Classification Through Static and Dynamic Analysis*" at the Annual International Computers, Software & Applications Conference (COMPSAC) in 2015 [122]. Marvin was developed with support from Matthias Neugschwandtner and Christian Platzer. Since the majority of Android apps are distributed through application markets, we present our study of the prevalence of malware in these markets and the design of AndRadar, an Android market radar for the efficient discovery of malware in the Google Play Store as well as alternative application markets, in Chapter 5. This work was published as "*AndRadar: Fast Discovery of Android Applications in Alternative Markets*" at the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) in 2014 [125] and is a result of joint work with Stamatis Volanis, Alessandro Sisto, Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, and Sotiris Ioannidis. In Chapter 6 we discuss related work and compare our work to other solutions. Finally, in Chapter 7, we conclude with an outlook on future work.

CHAPTER 2

Beagle: Insights into the Malicious Software Industry

In parallel with the development of cybercrime into a large underground economy driven by financial gain, malicious software has changed tremendously. Originally, malicious software was mostly simple self-propagating code crafted primarily in low-level languages and with limited code reuse. Today, malware has turned into a fully-fledged industry that provides the tools that cybercriminals use to run their business [169]. Like legitimate software, malware is equipped with auto-update functionality that allows malware operators to deploy arbitrary code to the infected hosts. New malware versions are frequently developed and deployed: Panda Labs observed 73,000 new malware samples per day in 2011 [150], with this number steadily rising to up to 225,000 new samples per day in the first quarter of 2015 [151]. Clearly, the majority of these samples are not new software but rather repacked versions or incremental updates of existing malware. Malware authors update their code in an endless arms race against security countermeasures such as antivirus engines and spam filters. Furthermore, to succeed in a crowded, competitive market they "innovate," refining the malware to better support the cybercriminals' modus operandi or to find new ways to profit at the expense of their victims.

Understanding how malware authors update their code over time is thus an interesting and challenging research problem with practical applications. Previous work has focused on constructing the *phylogeny* of malware, i.e., modeling the evolutionary relationship between malware samples and inferring their lineage, similar to the "tree of life" in biology [98, 113]. Instead, our goal is to observe subsequent versions of malware and automatically characterize their evolution. As a first step towards this goal, quantifying the differences between versions can provide an indication of the overall development effort behind this industry, during a specific observation window.

To provide deeper insight into malicious software and its development, we need to go one step further and identify how the observed changes between malware versions relate to the functionality of the malware. This is the main challenge that this work addresses. We propose techniques that combine dynamic and static code analysis to (1) identify the component of a malware binary that is responsible for each behavior observed in a malware execution trace, and to (2) measure the evolution of each component across malware versions. This allows us to attribute the measured development effort to behavior changes across malware families. In addition, our insights also allow a malware analyst to triage malware samples: By pinpointing versions of a family that introduce significant code changes and novel functionality she can focus reverse engineering efforts on interesting new versions instead of repacked versions or insignificant updates.

In order to demonstrate the viability of our approach we selected 16 malware samples from 11 families[1] that included auto-update functionality, and we repeatedly executed them in an instrumented sandbox over a period of several months, allowing them to update themselves and to download additional components. As a result of dynamic analysis, we obtain the unpacked malware code and a log of its system-level activity. We then compare subsequent malware versions to identify code that is shared with previous versions, and code that was added or removed. From the system-level activity we infer high-level behavior such as "downloading and executing a binary" or "harvesting email addresses." Then, we identify the binary code that implements this observed functionality, and we track how it changes over time. As a result, we are able to observe not only the overall evolution of each malware sample, but also the evolution of its individual functional components.

In summary, we make the following contributions:

- We propose techniques for binary code comparison that are effective on malware and that can also be used to contrast a single binary against multiple others, such as against all previous versions of a specific malware sample, samples from other malware families, or against a dataset of benign code.
- We propose techniques that use behavior observed from dynamic analysis to assign semantics to binary code. With these techniques, we can identify functional components in a malware sample and track their evolution across malware versions.
- We implement these techniques in a tool called Beagle,[2] and use it to automatically monitor 16 malware samples from 11 families for up to three months and track their evolution in terms of code and expressed behavior.
- Our results, based on over one thousand executions of 381 distinct malware binaries, provide insight into how malware evolves over time and demonstrate Beagle's ability to identify and track changes to malware components.

[1] We include multiple samples of the same family (*ZeuS*) to monitor the evolution of variants that are deployed by different botmasters.

[2] HMS Beagle was the ship that Charles Darwin sailed on in the voyage that would provide the opportunity for many of the observations leading to his development of the theory of evolution [61].

2.1 Evolution of Malicious Code

Malware is the underlying platform for online crime: From spam to identity theft, to denial of service attacks and to fake-antivirus scams [193], malware surreptitiously installed on a victim's computer plays an essential role in criminals' online operations. One key reason for the enduring success of malware is the way it has adapted to remain one step ahead of defenses and to support new, innovative criminal modus operandi. Cybercriminals are under constant pressure, both from the security industry, and—as in any market with low barriers to entry—from competing with other criminals. As a result, malware authors are required to constantly update their malware to adapt to the changing environment to survive.

The most obvious form of adversarial pressure that the security industry puts on malware authors comes from antivirus (AV) engines: Being detected by widely-deployed AVs greatly reduces the pool of potential victims of a malware. Thus, malware authors strive to defeat AV detection by using packers [155]—also known as "crypters." AV companies respond by detecting the unpacking code of known crypters. The result is that a part of the malicious software industry has specialized in developing crypters that are not yet detected by AVs. Interestingly, Russian underground forums advertise job openings for crypter developers with monthly paychecks of 2,000 to 5,000 US Dollars [114].

This is however only one aspect of the ongoing arms race. Spam bots try to defeat spam filters using sophisticated template-based spam engines [190]. Meanwhile, botnets' command and control (C&C) infrastructure are threatened by takedowns, so malware authors experiment with different strategies for reliably communicating with their bots [194]. Similarly, the security measures deployed at banking websites and other online services, such as two-factor authentication, require additional malicious development effort. For this, malware authors embed code into the victim's browser that is targeted at a specific website, for instance to mislead her into sending money to a different account than the one she intended to. A plugin implementing such a "web inject" against a specific website for a popular bot toolkit can be worth 2,000 US Dollars [114].

The need to remain one step ahead of defenses is only one reason for the constant evolution of malware. Cybercriminals further strive to increase their profits, which are threatened by trends such as the declining prices of stolen credentials [115], by developing new business models. These often require new or improved malware features. As an example, one sample in our dataset implements functionality to simulate a system malfunction on the infected host, presumably to try to sell the victim fake AV or utility software that will "solve" the problem, and another one is able to steal Bitcoin wallets.

From an economic perspective, we would like to know how expensive it is for a malware author to develop a new feature, as well as how much effort she needs to invest into defeating security countermeasures. In the absence of direct intelligence on a malware authors' practices, the malware code itself is the richest available source of information on their malware development process. Human analysts routinely analyze malware binaries to understand not only what they can do against them, but also to get a glimpse into the underground economy of cybercrime. In this work, we develop techniques to automate one aspect of this analysis: Measuring how malware changes over time and how these changes relate to the functionality it implements.

2.2 Approach

We implement the approach described in this section in a system we call Beagle. Beagle first dynamically analyzes a malware sample by running it in an instrumented sandbox, as detailed in Section 2.3.1. We allow the sample to connect to its C&C infrastructure to obtain an updated binary. This way, we capture new versions of the analyzed malware. Later, we analyze these updated samples by also executing them in our analysis sandbox. In addition to recording the sample's system- and network-level behavior, the sandbox also acts as a simple generic unpacker [173], and it records the unpacked version of the malware binary and of any other executables that are run on the system, including malicious code that is injected into benign processes. This allows us to analyze the deobfuscated binary code and compare it across versions with static code analysis techniques.

Then, Beagle compares subsequent versions of a malware's code to quantify the overall evolution of the malware code. Our approach to binary code comparison is based on the control-flow graph's (CFG) structural features, as described in Section 2.3.2. This first comparison provides us with an overall measure of how much the code has changed and how much new code has been added, but it offers no insight into the *meaning* of the observed changes.

To attach semantics to the observed changes, we take advantage of the behavioral information provided by our analysis sandbox. The sandbox provides us with a trace of low-level operations performed by the malware. In the behavior extraction phase, discussed in Section 2.3.3, we analyze this system-level activity to detect higher-level behaviors such as "sending spam," "downloading and executing a binary from the Internet." These behaviors are detected based on a set of high-level rules. In addition to these behaviors that have well-defined semantics (as determined by the human analyst who wrote the detection rule), we also detect behaviors with a-priori unknown semantics, by grouping related system-level events into *unlabeled behaviors*. Once a behavior has been detected, we employ a lightweight technique to map the observed behavior to the code that is responsible for it, and tag individual functions with the set of behaviors they are involved in.

Last, we perform semantic-aware binary comparison as described in Section 2.3.4. For this, we combine the results of the behavior-detection step with our binary-comparison techniques to analyze how the code's evolution relates to the observed behaviors. Thus, we are able to monitor when each behavior appears in a malware's code-base and to quantify the frequency and extent of changes to its implementation. The goal is to provide insight into the semantics of the code changes and help the analyst to answer questions such as "What is the overall amount of development effort behind a malware family?", "Within a family, which components are most actively developed?" or "How frequently is a specific component tweaked?".

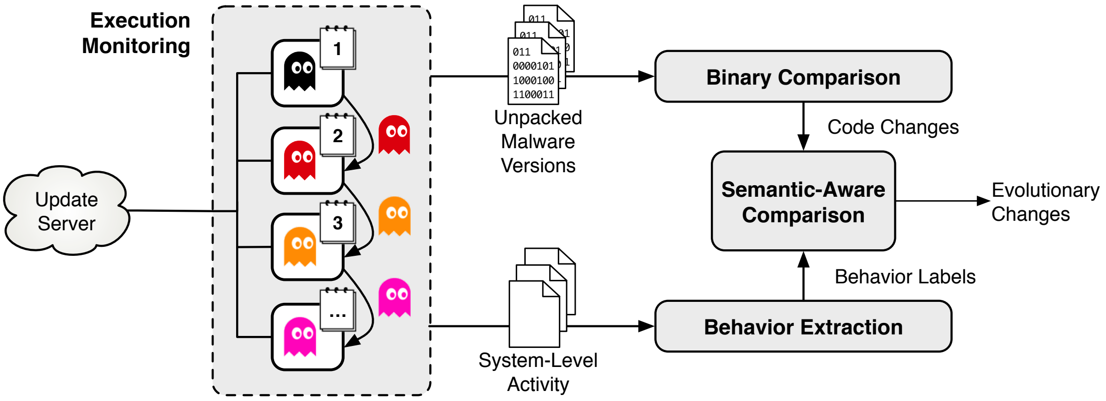


Figure 2.1: *System overview of* Beagle. We monitor the execution of samples in a sandbox and allow them to update themselves. We extract the unpacked binary code, which we decompile and analyze for static code changes to previous versions. In parallel, we extract the dynamic behaviors from the system-level logs collected during execution. These behaviors form "labels" that we then match to the corresponding portion of static code—attaching meaning (i.e., a behavioral label) to code changes.

2.3 System Description

Beagle has four modules (see Figure 2.1): The Execution Monitoring module is a dynamic analysis sandbox that logs the relevant actions that each sample performs during runtime (e.g., system calls, sockets opened, registry and file modifications). The Behavior Extraction module analyzes these logs for each malware sample and extracts high-level behaviors using a set of rules and heuristics. In parallel, the Binary Comparison module disassembles the unpacked binary code. Then, it analyzes the code of each malware sample to find added, removed and shared portions of the CFG across samples, and "labels" these portions with the high-level behavior extracted at runtime. Finally, the Semantic-Aware Comparison module monitors how the labeled code evolves over time.

2.3.1 Execution Monitoring

The first step is to obtain subsequent versions of a specific malware family. One approach would be to rely on the labeling of malware samples by AV engines, and select samples from a specific family as they become available over time. However, this approach has two problems. First, AV engines are not very good at classifying malware largely because their focus is on detection rather than classification [27, 132]. Second, different samples of a malware family may be independent forks of a common code base, which are developed and operated by different actors. The approach we use in Beagle is to take advantage of the auto-update functionality included in most modern malware. Specifically, by running malware in a controlled environment and letting it update itself, we can be confident that the intermediate samples that we obtain represent the evolution over time of a malware family—as deployed by a specific botmaster.

Stateful sandbox. A malware analysis sandbox typically runs each binary it analyzes in a "clean" environment: Before each execution, the state of the system in the sandbox is reset to a snapshot. This ensures that the sample's behavior and other analysis artifacts are not affected by previously analyzed samples. To allow samples to install and update themselves, however, we need a different approach. A simple yet extremely inefficient solution would be to let the malware run in the sandbox for the entire duration of our experiments. Running a malware sample for months also increases the risk that, despite containment measures, it may cause harm. Instead, we rely on malware's ability to persist on an infected system. Each sample is allowed to run for only a few minutes. At the end of each analysis run, Beagle captures the state of the system, in the form of a patch that contains file-system and registry changes to the "clean" system state. Additionally, we detect a malware sample's persistence mechanism by monitoring known auto-start locations [209]. When we want to re-analyze the sample, we start by applying the patch to the "clean" snapshot. Then, we trigger the detected persistence mechanism, so that we effectively simulate a reboot, thus continuing the analysis with updated versions. Hence, in the first execution we observe the installation and update functionality of the original malware sample. In the following executions we observe the updated versions and additional updates. If the original binary is a dropper, we monitor the installation of additional malware in the first execution. During consecutive executions we then monitor updates of the dropped payload as well as—if the dropper itself stays persistent—updates to the dropper and changes in the dropped payload.

Generic unpacking. Beagle's analysis sandbox also captures the deobfuscated binary code by acting as a simple, generic unpacker. In its current implementation, Beagle simply captures and dumps all code found in memory at process exit or at the end of the analysis run. We do not restrict this to the initial malware process, but also include all code from binaries started by the malware or of processes in which the malware has injected code. For instance, one of the *ZeuS* samples in our dataset downloads code from a C&C server and injects it into the `explorer.exe` process. In such a case, Beagle also dumps an image of the `explorer.exe` process that includes the injected code.

This simple approach to unpacking, although sufficient for the purpose of our experiments, could be defeated by more advanced packing approaches where unpacked code is re-packed or deleted immediately after use. To address this limitation, Beagle could be extended to incorporate more advanced generic unpacking techniques [135, 173].

Sandbox implementation. Beagle's sandbox is an extension of Anubis [32, 34], a dynamic malware analysis sandbox based on the whole-system emulator QEMU [35]. Anubis captures a malware's behavior at the API level, producing a log of the invoked system and API calls, including parameters and return values. Furthermore, Anubis uses dynamic taint tracking to capture data flow dependencies between these calls [33]. In order to facilitate attribution of behavior observed in the sandbox to the code responsible for it, we extended the sandbox to log the call stack corresponding to each call. Here, we need to make sure to not only log return addresses inside libraries for nested API calls. We achieve this by walking the call stack until we find a return address inside the malware binary and dumping the remaining call stack for each API call.

2.3.2 Binary Comparison

The next component of Beagle compares binaries and identifies the code that is shared between versions, the code that was added, and the code that was removed. This allows us to quantify the evolution of malicious code by measuring the size of code changes and computing a similarity score between malware versions. More importantly, as discussed in Section 2.3.4, we combine this information with code semantics inferred from dynamic behavior to gain an understanding of how evolution relates to malware functionality.

Code fingerprints. Our technique for binary comparison relies on the structure of the intra-procedural CFG, extending work from Kruegel et al. [116]. The CFG is a directed graph where nodes are basic blocks and an edge between two nodes indicates a possible control flow transfer (e.g., a conditional jump) between those basic blocks. Nodes in the CFG are colored based on the classes of instructions that are present in each node, and the problem of finding shared code between two binaries is reduced to searching for isomorphic k-node subgraphs (we use $k = 10$ following Kruegel et al.). As this problem is intractable, Kruegel et al. proposed an efficient approximation that relies on extracting a subset of a CFG's k-node connected subgraphs and normalizing them. Each normalized subgraph is a concise representation of a code fragment. In practice, we hash the normalized subgraph to generate a succinct fingerprint. By matching fingerprints generated from different code samples, we are able to efficiently detect similar code.

As shown by Kruegel et al., these fingerprints are to some extent resistant to code metamorphism, and are effective for detecting code reuse in malware binaries [139]. Here, we take this a step forward and use them to locate the shared code between successive malware versions. For this, given an unpacked malware binary M, we disassemble it, extract the colored CFG and compute the corresponding set of CFG fingerprints $\mathcal{F}_M$. For each fingerprint $f \in \mathcal{F}_M$, we also keep track of the addresses $b_M(f)$ of the set of basic blocks it was generated from. For a set of fingerprints $\mathcal{F}$, we indicate the corresponding basic blocks in sample M with $\mathcal{B}_M(\mathcal{F}) = \bigcup_{f \in \mathcal{F}} b_M(f)$. To compare two samples M and N, we compute the intersection of the their fingerprints $\mathcal{I}_{M,N} = \mathcal{F}_M \cap \mathcal{F}_N$. Finally, when mapping back the fingerprints in this set to the corresponding basic blocks in either sample we get the code that is common to M and N. We call them *shared* basic blocks. In sample M, the set of basic blocks that is shared with N is thus

$$\mathcal{S}_M(M,N) = \mathcal{B}_M(\mathcal{I}_{M,N}) \tag{2.1}$$

If M and N are two successive versions of a malware

$$\mathcal{A}_N(M,N) = \mathcal{B}_N \setminus \mathcal{S}_N(M,N) \tag{2.2}$$

(short, $\mathcal{A}(M,N)$) is the set of basic blocks *added* in the new sample N, whereas

$$\mathcal{R}_M(M,N) = \mathcal{B}_M \setminus \mathcal{S}_M(M,N) \tag{2.3}$$

(short, $\mathcal{R}(M,N)$) is the set of basic blocks *removed* from the old sample M.

Code whitelisting. The unpacked code produced by our sandbox may include code unrelated to the malware. One reason is that malware may inject code into legitimate processes, which will then be included into our analysis. Furthermore, a malware process sometimes loads system dynamically linked system libraries (DLLs) directly into its address space, without using the standard API functions for loading DLLs. In both cases, the unpacked binaries produced by our analysis sandbox will include code that is not part of the malware.

To exclude this code from analysis, we rely on a whitelisting approach: We identify all code (executables and DLLs) in the "clean" sandbox, and compute the set $\mathcal{W}$ of all CFG fingerprints found in this code. In our analysis of each malware sample M, we then identify the basic blocks $\mathcal{B}(\mathcal{W})$ that match these fingerprints. We call them *whitelisted* basic blocks, and do not take them into account for further analysis. An additional benefit of whitelisting code from a clean system is that this can also eliminate code from standard system libraries that has been statically linked into a malware binary.

Similarity and evolution. We compute the similarity between two malware samples with a variant of the Jaccard set similarity [103]:

$$J(M, N) := \frac{|\mathcal{S}^{\star}(M, N)|}{|\mathcal{S}^{\star}(M, N)| + |\mathcal{A}(M, N)| + |\mathcal{R}(M, N)|} \tag{2.4}$$

This similarity measure is the number of shared basic blocks over the total of shared, added and removed basic blocks. The number of shared basic blocks in the two samples $|\mathcal{S}_N(M, N)|$ and $|\mathcal{S}_M(M, N)|$ may differ slightly, because multiple identical k-node subgraphs may exist in a single sample. We mitigate this by picking $|\mathcal{S}^{\star}(M, N)|$, which is the maximum between the two.

In addition to comparing pairs of samples, we would like to contrast a new malware sample against all previously observed versions, and identify the code that is new in the latest variant. Measuring this code provides the most direct measure of the malware authors' development effort for this variant. For this, we compute the set $\mathcal{F}_{M_1,..,M_{t-1}}$ of fingerprints found in the first $t-1$ samples, and identify the *new basic blocks* $\mathcal{N}_{M_t} = \mathcal{B}_{M_t} \setminus \mathcal{B}_{M_t}(\mathcal{F}_{M_1,..,M_{t-1}})$.

2.3.3 Behavior Extraction

Automatically understanding the purpose and semantics of binary code is a challenging task. The system-level behavior of a malware sample in an analysis sandbox, however, can be more readily interpreted. To detect specific patterns of malicious behavior previous work [104, 136] has started from system-level events, enriched with data flow information. Martignoni et al. [136] frame this as a "semantic gap" problem, and propose a technique to detect high-level behavior from system-level events using a hierarchy of manually crafted rules. In the absence of prior knowledge of a pattern of malicious behavior, on the other hand, no such rules are available. As an alternative to leveraging prior knowledge, researchers have used data flow to link individual system-level events into graphs, and applied data mining techniques to a corpus of such graphs to learn to detect malware [55] or to identify its C&C communication [105].

We aim to make our observation of the evolution of malware functionality as complete and insightful as possible. Therefore, we assign semantics to observed behavior that matches known patterns, but also take into account behaviors for which no high-level meaning can be automatically established. In this work, we call the former *labeled*, and the latter *unlabeled* behavior.

Unlabeled behavior. An unlabeled behavior is a connected graph of system-level events observed during the execution of a sample. Nodes in this graph represent system or API calls, whereas the edges represent data flow dependencies between them. As data flow dependencies are lost when files are written to disk, we also connect nodes that operate on the same file system resources.

Our purpose for taking unlabeled behavior into account is to identify components of the malware and measure their evolution, even though we do not (yet) know what functionality they implement.

Labeled behavior. A labeled behavior consists of one or more unlabeled behaviors—or, in other words, connected subgraphs of system-level events—such that they match a manually crafted specification of a known malware behavior. These specifications can take into account the API calls involved, their arguments, as well as the data flow between them. For instance, we define the `DOWNLOAD_EXECUTE` behavior as any data flow from the network to a file that is later executed; another example is the `AUTO_START` behavior, which we define as write access to any known auto-start location.

Behaviors that remain unlabeled can be examined by an analyst who can assign semantics to them and define behavior specifications for detecting them. Furthermore, as follow-up work by Polino et al. [160] demonstrates, web knowledge bases can be used to automatically annotate behaviors. Thus, Beagle's knowledge base of behavior specifications grows over time to cover a broader spectrum of malware functionality. As we show in Section 2.4.6, a relatively small number of manually written behavior specifications was sufficient to cover a significant fraction of the malware code that was executed during our experiments.

2.3.4 Semantic-Aware Comparison

The goal of the semantic-aware comparison is to attach meaning to the overall changes in the binary code. Essentially, we divide the malware program into a number of functional components. To this end, Beagle starts from the behaviors observed at runtime, as discussed in Section 2.3.3, and identifies the binary code that is responsible for each observed behavior. By tracking the evolution of this code across malware versions, we are able to measure the evolution of malware's functional components.

Mapping behavior to code. The output of the behavior extraction phase is, for each behavior observed in a sample's execution, a sequence of system or API calls with the corresponding call stack. Beagle next tries to identify the code responsible for this behavior. Beagle works at function granularity: It identifies the set of functions in the unpacked binary that are involved in that behavior, and "tags" them with the behavior. A single function may be tagged with multiple behaviors (e.g., a utility function that is re-used across different functional components). To identify the functions involved in a behavior, we use static analysis to identify a code path that could have been responsible for the observed sequence of calls (taking into account the corresponding call stacks). To do so, we resolve the path between any two consecutive calls by recursively looking up code references to the target function until we find the source function. We use the addresses in the call stack as landmarks this path should traverse and in case the dynamic path cannot be resolved statically. We tag all functions in the identified path as part of the behavior.

Working at function granularity is a design decision that trades off some precision in delimiting functional components, in order to achieve performance compatible with a large-scale experiment. As discussed in Section 2.3.1, our modified sandbox logs events at the system API level. Previous work that performed a similar mapping of behavior to code at instruction granularity [139], on the other hand, relied on a sandbox logging each executed basic block.

Behavior evolution. The set of functions that implement a behavior is a functional component of a malware instance. By comparing the components that implement a behavior in successive versions of a malware, we can observe the evolution of that functionality over time. This allows us to get an idea of the development effort involved in updating this functionality by measuring the amount of new code, as well as quickly identifying significant updates to the malicious functionality that may warrant further inspection. For this, we apply the techniques discussed in Section 2.3.2 to successive versions of a component, instead of considering entire unpacked binaries. Among the versions of a component observed throughout our experiments, we select the largest implementation by number of basic blocks and call it the *reference behavior*.

Dormant functionality. Like any dynamic code analysis approach, a limitation of Beagle is incomplete code coverage. In a typical execution, a malware sample will reveal only a fraction of the functionality it is capable of: For instance, a bot will send spam only if instructed to do so by the botnet's C&C infrastructure. Thus, the techniques described above will identify the presence of each functional component only in some of a sample's executions, even though the code implementing the functionality is present throughout our experiments. This limits our visibility of the component's evolution: If a behavior is observed only once, we do not see any evolution. In order to address this limitation and to track evolution in a more complete way, we use the CFG fingerprints described in Section 2.3.2 to identify a component even in executions where it is *dormant* and the corresponding behavior cannot be observed. For this, we identify the dormant components by locating the functions in a sample that match fingerprints from an *active* (non-dormant) component in another execution.

2.4 Evaluation

We now describe our experiments and the dataset that we processed with Beagle to measure and analyze the evolution of malicious code over several months.

2.4.1 Setup

We run Beagle on a desktop-class, dual-core machine with 4GB of RAM, and execute each sample for 15 minutes approximately once a day, depending on the workload of the sandbox. Each analysis continues from the state of the previous day's execution in order to analyze only the updated versions. Since we want to observe malware updating itself, we cannot run it in a completely isolated environment, but need to allow it to access the C&C infrastructure from which to obtain updates. To prevent malware from causing harm, we employ containment measures such as redirecting "dangerous" protocols to a local honeypot and limiting bandwidth and connections. These measures cannot guarantee that the malware we run will never cause harm (0-day attacks are especially hard to recognize and block), but we believe that they are sufficient in practice if combined with a prompt response to any abuse complaints (we did not receive any complaints during the course of our experiments).

2.4.2 Dataset

We selected samples from three different sources: (1) Recent submissions to Anubis for which the data flow detection of Jackstraws [105] indicated download & execute behavior, (2) malware variants from the top threats according to the Microsoft Malware Protection Center,[3] and (3) *ZeuS* samples from ZeuS Tracker.[4] We then discarded samples that showed no update activity in our environment.

As summarized in Table 2.1, we analyzed the evolution of 16 samples from 11 families between September 2011 and April 2012. We stopped the analysis and discarded a sample after it failed to contact its C&C server for more than two weeks. Overall, we analyzed a total of $1,023$ executions of 381 distinct malware binaries.

2.4.3 Validation

Beagle automatically finds differences between hundreds of malware samples in a few hours on a desktop-class computer. Clearly, an in-depth understanding of the code differences would still require a reverse engineer, which Beagle cannot possibly substitute. However, Beagle provides valuable guidance to quickly decide what are the interesting changes between malware releases to focus manual inspection on. That said, it is hard to assess the correctness of Beagle, mainly because we do not have the source code of successive malware versions, and we lack ground truth on the semantics of malware components and their comparison.

[3] `https://www.microsoft.com/security/portal/threat/threats.aspx`

[4] `http://zeustracker.abuse.ch`

Family Name, Label & MD5 of 1st Sample	Source	1st Day	Days	Executions	MD5s	Lifespan in Days
Banload TrojanDownloader:Win32/Banload.ADE `09af6de40ab414f41ba48b447345e75d`	(1)	2012-01-31	87	78	3	2.00/83.00/29.33/37.95
Cycbot Backdoor:Win32/Cycbot.G `b5f1ac49b4112965cb98d19f0a2817c4`	(1)	2011-09-15	73	73	69	1.00/73.00/ 2.04/ 8.60
Dapato Worm:Win32/Cridex.B `fd1ca22b357c60fda74a262e1f9aa050`	(2)	2012-02-24	65	62	25	1.00/43.00/ 4.60/ 8.31
Gamarue Worm:Win32/Gamarue.B `df655a37e7da76c5382901a39ffe55ef`	(2)	2012-02-10	78	77	19	1.00/76.00/ 8.47/16.44
GenericDownloader TrojanDownloader:Win32/Banload.AHC `e4cb4a134cbca527c0ea858417451950`	(1)	2012-01-31	82	79	5	2.00/69.00/16.80/26.16
GenericTrojan Worm:Win32/Vobfus.gen!S `08465623c844031fb4f9fd13c262404d`	(1)	2012-02-07	76	73	55	1.00/44.00/ 2.71/ 6.32
Graftor TrojanDownloader:Win32/Grobim.C `50aafcaaab1db2619bfbe4bf5a8154d5`	(1)	2012-02-17	37	39	22	1.00/17.00/ 6.00/ 5.53
Kelihos TrojanDownloader:Win32/Waledac.C `b21322b221738ba0906e79b07e51314b`	(2)	2012-03-03	56	38	8	1.00/54.00/21.00/22.88
Llac Worm:Win32/Vobfus.gen!N `a02d77cd8e7b7b08fc7f61ef43fcd817`	(1)	2012-02-07	32	33	82	1.00/10.00/ 1.49/ 1.71
OnlineGames Worm:Win32/Taterf.D `4d6abb30d945663c822a061de6f095b7`	(1)	2011-09-02	87	80	47	1.00/38.00/ 3.94/ 7.28
ZeuS PWS:Win32/Zbot.gen!AF `1be8884c7210e94fe43edb7edebaf15f`	(3)	2012-02-09	79	78	6	1.00/78.00/26.67/28.70
ZeuS PWS:Win32/Zbot `9926d2c0c44cf0a54b5312638c28dd37`	(3)	2012-02-15	74	73	4	1.00/50.00/18.50/19.63
ZeuS PWS:Win32/Zbot.gen!AF `c9667edbbcf2c1d23a710bb097cddbcc`	(3)	2012-02-23	66	63	6	1.00/36.00/11.00/13.43
ZeuS PWS:Win32/Zbot.gen!AF `dbedfd28de176cbd95e1cacdc1287ea8`	(3)	2012-02-09	79	78	4	1.00/78.00/20.25/33.34
ZeuS PWS:Win32/Zbot.gen!AF `e77797372fbe92aa727cca5df414fc27`	(3)	2012-02-10	79	77	5	1.00/77.00/16.20/30.40
ZeuS PWS:Win32/Zbot.gen!AF `f579baf33f1c5a09db5b7e3244f3d96f`	(3)	2012-03-03	57	55	11	1.00/30.00/ 5.64/ 9.75

Table 2.1: *Malware families in our dataset.* The labels in the first columns are based on Microsoft AV naming convention. Sources are (1) Anubis, (2) Microsoft Malware Protection Center and (3) Zeus Tracker. The MD5 column is the number of distinct binaries encountered. Lifespan is the duration in days of the interval in which an MD5 was observed (min/max/mean/stdev).

The binary comparison component of BEAGLE described in Section 2.3.2, however, can be validated by comparing its results against established tools for binary code comparison: We use BinDiff [2, 81], the leading commercial tool for binary comparison and patch analysis, and compare the similarity scores it produces when comparing pairs of samples to those from BEAGLE. Note that BinDiff is based on a completely different approach and uses a number of proprietary heuristics for comparison. Furthermore, it relies on program's call graph, which is not taken into account by our tool. Although the absolute value of BinDiff's similarity score and BEAGLE's similarity score differ, we were able to find a linear transformation[5] from one value to another with a low residual mean square error (6.3% on average, with a peak of 17% for *GenericTrojan*). However, BinDiff cannot efficiently be used to contrast a binary against multiple others. BEAGLE's binary comparison component, on the other hand, can be used to contrast a sample against all previous versions (to detect new code), or against a library of benign software (for whitelisting), as discussed in Section 2.3.2.

2.4.4 Overall Changes

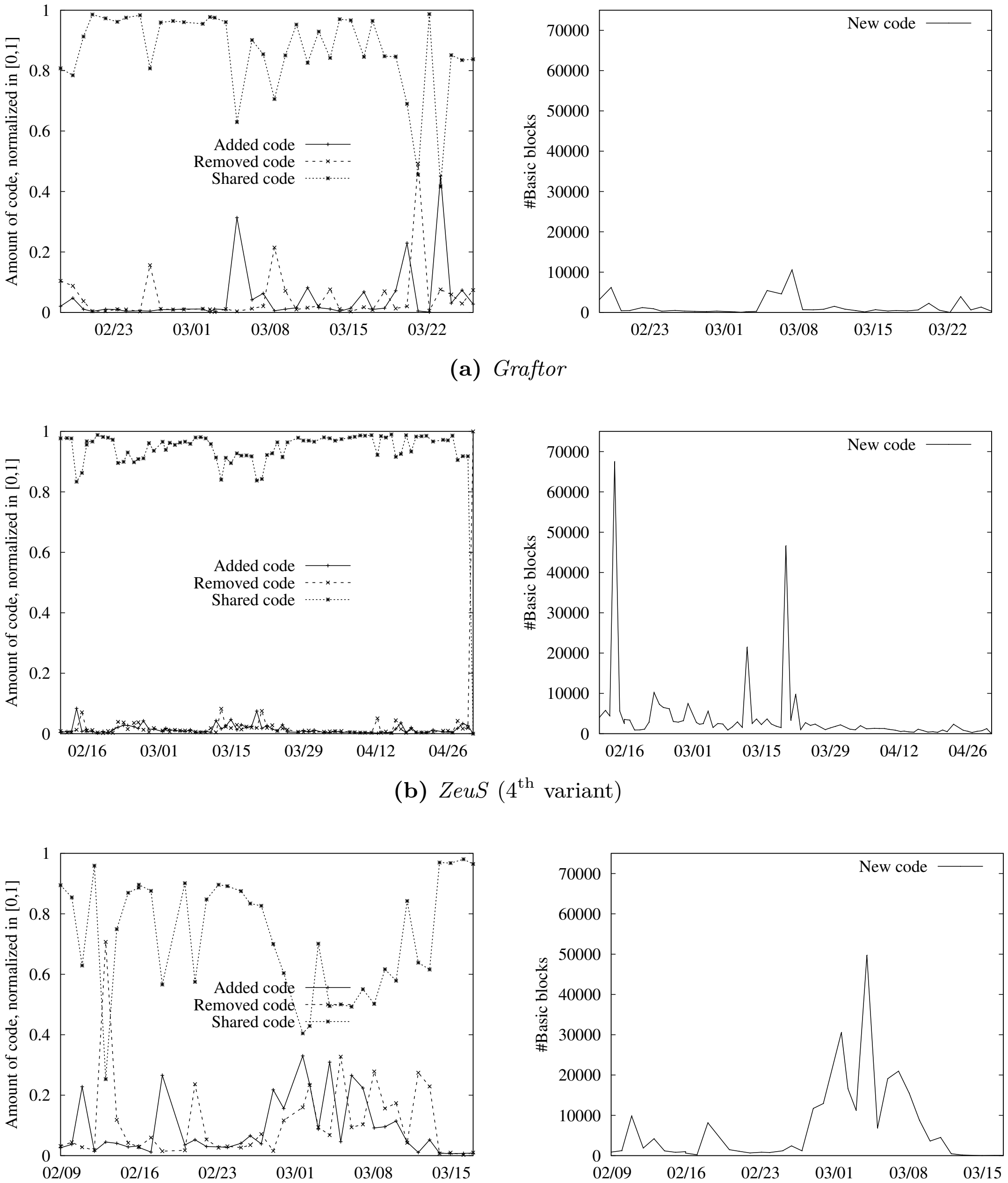
In Figure 2.2 we show the timelines resulting from comparing samples within six families: *Graftor*, *ZeuS* (4th variant), *Llac*, *Dapato*, *Gamarue*, and *GenericDownloader*. For the first column of the figure, we consider each consecutive pair of samples of a malware family, that is at time t and $t-1$. Then, as described in Section 2.3.2, we calculate the amount of added, removed and shared code (see Equation (2.4)), expressed in distinct basic blocks, between the two samples, and normalized to the total number of distinct basic blocks. For the second column, we calculate the absolute amount of code that was never found in any of the previous samples.

These timelines show that overall, day-to-day changes in malware code are relatively small: As would be expected, most new malware versions are incremental updates that reuse most of the code. New code is largely concentrated in a smaller number of peaks that indicate significant updates to the malware code base. For some families, the amount of brand new code in some of these peaks is significant, up to for instance $50,000$ new basic blocks for *Llac*, and more than $70,000$ new basic blocks for *GenericDownloader*.

Figure 2.3 shows a CDF of day-to-day differences between successive malware versions (2.3a) and between each version and the first analyzed version (2.3b). Comparing the two graphs clearly shows that while daily updates mostly consist of small changes, for some families the cumulative effect of these small changes eventually result in binaries that are very different from the original sample. This long-term evolution varies a lot across the families in our dataset. In Figure 2.3b we highlight *ZeuS* (2nd variant) and *Gamarue* that show particularly large cumulative changes.

Table 2.2 summarizes our results, which confirm that, in the majority of cases, malware authors reuse a significant amount of code when they release day-to-day updates and also in the long run to some extent. Remarkably, for a few families the new code added in day-to-day changes accounts for ∼10% of the entire malware code on *average*.

[5] To this end, we used `R`'s linear model fitting functions (`lm()` and `poly()`).

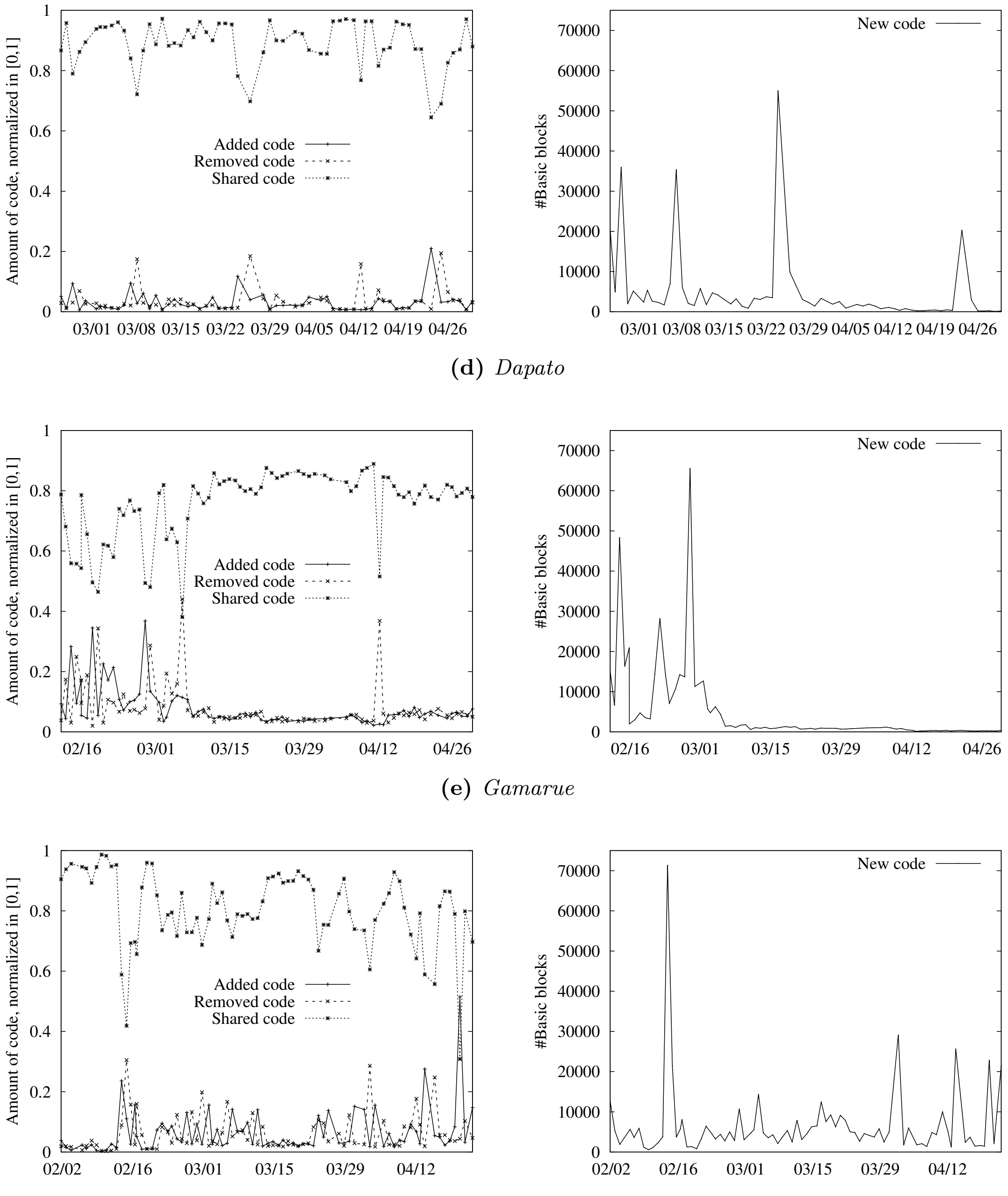


(a) *Graftor*

(b) *ZeuS* (4th variant)

(c) *Llac*

Figure 2.2: *Added, removed and shared code over time.* The first column shows the day-to-day changes in the code (i.e., we compare subsequent pairs of samples in the same family), whereas the second column shows the new code added by each sample compared to all the previous version. The development effort varies between families, but in general incremental update reuse most of the code, while new code is largely concentrated in a smaller number of peaks.



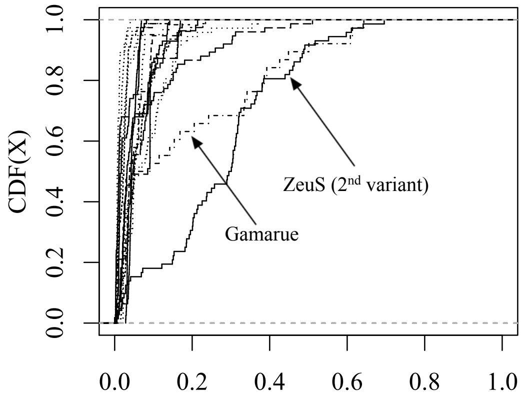
(d) *Dapato*

(e) *Gamarue*

(f) *GenericDownloader*

Family Name	%Tagged	%Labeled	%Ratio	%Added	%Removed	%Shared	New	#Labels
Banload	7.31 ± 1.70	6.68 ± 0.75	91.43	2.48 ± 2.96	2.83 ± 3.10	94.69 ± 3.75	176.2 ± 409.2	5
Cycbot	32.36 ± 2.40	31.23 ± 2.95	96.50	10.59 ± 10.36	10.30 ± 10.42	79.11 ± 12.80	1361.4 ± 3937.2	11
Dapato	2.81 ± 1.22	1.15 ± 0.55	40.90	5.15 ± 5.14	5.57 ± 5.63	89.28 ± 7.48	2402.9 ± 7165.3	4
Gamarue	15.90 ± 14.06	14.06 ± 13.40	88.42	12.08 ± 8.16	12.50 ± 9.32	75.41 ± 11.57	2500.1 ± 7747.2	12
GenericDownloader	9.10 ± 1.93	8.58 ± 1.59	94.30	9.80 ± 9.85	9.58 ± 8.81	80.62 ± 12.48	3330.6 ± 7367.8	6
GenericTrojan	22.94 ± 11.05	20.18 ± 10.69	87.97	16.66 ± 16.15	17.03 ± 15.15	66.31 ± 18.76	4974.1 ± 14339.6	11
Graftor	12.66 ± 6.20	9.58 ± 4.70	75.70	6.47 ± 10.40	6.84 ± 9.96	86.69 ± 13.48	682.0 ± 1662.8	4
Kelihos	24.20 ± 2.24	24.09 ± 2.26	99.53	5.18 ± 8.69	5.60 ± 10.10	89.23 ± 12.64	2145.3 ± 4065.3	12
Llac	19.13 ± 14.25	19.11 ± 14.26	99.91	12.82 ± 12.53	14.45 ± 14.70	72.73 ± 19.05	3323.3 ± 7899.1	10
OnlineGames	2.18 ± 0.30	1.96 ± 0.21	89.97	3.35 ± 3.12	3.37 ± 3.12	93.28 ± 5.44	420.0 ± 718.0	9
ZeuS	8.37 ± 2.59	6.15 ± 1.32	73.44	2.10 ± 2.24	3.59 ± 11.27	94.31 ± 11.28	1910.8 ± 6148.0	11
ZeuS	8.26 ± 1.56	6.44 ± 1.14	78.00	3.65 ± 3.07	5.25 ± 11.85	91.09 ± 12.41	4086.0 ± 11936.3	12
ZeuS	10.45 ± 2.67	7.91 ± 2.49	75.73	2.61 ± 2.20	4.51 ± 12.64	92.88 ± 12.47	2234.5 ± 7117.9	11
ZeuS	8.55 ± 2.15	6.53 ± 1.19	76.41	2.55 ± 2.51	3.93 ± 11.26	93.52 ± 11.35	2013.6 ± 6874.5	12
ZeuS	8.82 ± 1.79	7.73 ± 1.36	87.65	3.12 ± 2.78	4.57 ± 11.33	92.32 ± 11.46	3245.9 ± 7456.3	12
ZeuS	7.44 ± 1.31	6.41 ± 0.88	86.06	2.24 ± 2.51	4.53 ± 13.46	93.23 ± 13.46	2523.9 ± 6834.9	13

Table 2.2: *Summary of code changes.* We list the percentage of overall tagged and labeled code (in each version), added, removed, shared code (between consecutive versions), and new code (with respect to all previous versions) for each family (mean±variance, measured in basic blocks). #Labels is the number of distinct behavior labels detected throughout the versions.



(a) CDF of day-to-day changes ($t-1$ vs. t) X=fraction of added basic blocks.

(b) CDF of changes to first version (t_0 vs. t) X=fraction of added basic blocks.

Figure 2.3: *CDF of added basic blocks per family.* (a) Day-to-day changes are concentrated around low values for all the families, whereas (b) in the long run each family evolves distinctively, showing different development efforts. *ZeuS* (2^{nd} variant) and *Gamarue* show particularly large cumulative changes.

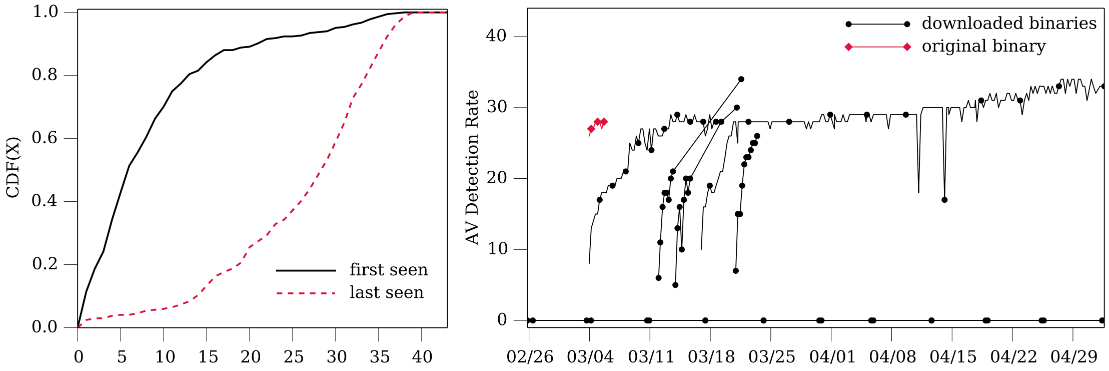
2.4.5 AV Detection

In Section 2.1 we suggested that AV engines are one of the main reasons malware authors frequently update their code. During the course of our experiments, we scanned all observed binaries with 43 AV engines by using the VirusTotal service. Figure 2.4a shows that, as expected, the detection rate of AV engines on a set of binaries—in this case, all binaries executed in our experiments—is generally lower at the beginning of their life cycle than towards the end. More interestingly, we found that in some families, such as *Kelihos*, as shown in Figure 2.4b, the malware authors seem to release a new binary as soon as previous binaries get detected by AV engines.[6] Indeed, whenever a larger number of AV engines start detecting a sample, the malware authors unleash a new, unknown binary, causing a sudden drop in the detection rate.

2.4.6 Code Behavior

As described in Section 2.3.3, we specify behavior as graphs of API calls, connected by data flow, and also take into account API call parameters. Depending on the granularity at which Beagle should track changes, an analysts can label behavior by rules that comprise only one function such as `RegSetValue(HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run,*)` for (one variant of) the `AUTO_START` behavior, or more complex rules such as `InternetOpenUrl|connect` → `InternetReadFile|recv` → `WriteFile` → `WinExec|CreateProcess` for the `DOWNLOAD_EXECUTE` behavior.

[6] In addition to malicious binaries *Kelihos* also installs the legitimate WinPcap library, which is consequently not flagged by any AV scanner (flat line at the bottom of Figure 2.4b).



(a) AV detection rate on all binaries (X out of 43 AV scanners)

(b) AV detection rate on *Kelihos* binaries (Y out of 43 AV scanners)

Figure 2.4: *AV detection rate of 43 AV engines on our dataset.* (a) CDF of the detection rate on all malware samples, at the time each sample is observed for the first and last time in our experiments. (b) Detection rate on *Kelihos* binaries over time—one line per distinct binary. The majority of binaries were detected by none or very few scanners at beginning of their life cycle—black, solid line in (a). At the end of their respective life cycles, when these binaries were possibly replaced with updated versions, the majority of binaries were detected by (almost) all scanners—red, dashed line in (a).

Taking into account a-priori knowledge about the behavior of the samples in our dataset and observations during the analysis, we defined rules that label a total of 33 distinct behaviors in the following categories:

Install. We detect modifications of known auto-start locations (AUTO_START) to recognize when a sample installs itself and gains persistence on the system.

Networking. We label network behavior based on protocols and port numbers. We label any data flow to the network over port 25 as SPAM. Furthermore, we distinguish between HTTP_REQUEST, HTTPS_REQUEST, DNS_QUERY, general TCP_CONNECTION, TCP_TRAFFIC and UDP_TRAFFIC as well LOCAL_CONNECTION from and to localhost. Finally, OPEN_PORT indicates that the malware listens on a local port.

Download & execute. This describes the updating functionality of a malware sample by labeling any data flow from the network to files that are then executed (DOWNLOAD_EXECUTE) or to memory sections that are injected into foreign processes (DOWNLOAD_INJECT).

Information stealing. We currently detect five information-stealing behaviors. FTP_CREDENTIALS and EMAIL_HARVESTING indicate that the malware harvests FTP credentials and email addresses, respectively, from the file system and registry. BITCOIN_WALLET indicates that the malware searches for a Bitcoin wallet to steal digital currency. `REMOVE_FLASHPLAYER_FILES` is detected when the malware tries to steal and delete Flash Player cookies and settings. Finally, `INTERNET_HISTORY` is detected when the malware accesses the user's browsing history.

Fake AV. Fake AV software modifies system settings to simulate system instability and persuade the victim to pay for additional software that fixes these "problems." We detect this as modifications to the registry that disable the task manager (`DISABLE_TASKMGR`) or hide items in the start menu (`HIDE_STARTMENU`), as well as enumeration of the file system and setting all file attributes to hidden (`HIDE_FILES`).

Browser hijacking. We detect this type of behavior when a malware changes Internet Explorer's or Firefox's proxy settings, (`IE_PROXY_SETTINGS` and `FIREFOX_SETTINGS`).

Anti AV. We detect behavior that disables system call hooks commonly used by AV software by restoring the System Service Dispatch Table (SSDT) from the disk image of the Windows kernel (`RESTORE_SSDT`).

AutoRun. We detect the use of the `AUTO_RUN` feature as changes to the AutoRun settings in the registry, modifications of `autorun.inf`, and the enumeration and spreading to external drives.

Lowering security settings. We currently detect three types of behaviors that lower a system's security settings: the creation of new Windows firewall's rules or attempts to disable it completely (`FIREWALL_SETTINGS`), registry modifications that disable Internet Explorer's phishing filter (`IE_SECURITY_SETTINGS`), and changes to a system security policy that classifies executables as low risk file types when downloading them from the Internet or when opening email attachments (`CHANGE_SECURITY_POLICIES`).

Miscellaneous. We also detect a number of simple behaviors that have self-explanatory labels: `INJECT_CODE`, `START_SERVICE`, `EXECUTE_TEMP_FILE`, `ENUMERATE_PROCESSES` and `DOWNLOAD_FILE`.

Unpacking. To identify a malware's unpacking code (`UNPACKER`), we do not rely on behavior rules as we do for other labels. Instead, we assume that all code found in the original malware binary *before unpacking* is part of the unpacking behavior. The reason is that malware authors use packing to hide as much as possible of their software from analysis and detection: Thus, all other functionality is typically found inside the packing layer.

These behaviors can be observed on their own, or as part of other behaviors, e.g., `DOWNLOAD_FILE` and `EXECUTE_TEMP_FILE` both are components of the `DOWNLOAD_EXECUTE` behavior when they are connected by data flow.

We summarize the behavior coverage we achieve with this rule set in Table 2.2. The first column of Table 2.2 shows the percentage of a sample's overall code that is tagged with any behavior (labeled or unlabeled). This is all the code that is responsible for *any* observed behavior, and is a measure of the code coverage of our dynamic analysis. Overall coverage is relatively low, which confirms the difficulty of performing a complete dynamic analysis. The second column shows the percentage that is tagged with a labeled behavior; that is, code to which we were able to attribute a high-level purpose. Except for one outlier (*Dapato*, at 40.9%) the labeled code is on average 73.4% – 99.91% of the total tagged code. This shows that Beagle was able to assign most executed code to a functional component based on the 33 behaviors we defined rules for.

2.4.7 Behavior Evolution

With the techniques discussed in Section 2.3.4, Beagle is able to monitor the evolution of each of the detected behaviors across successive malware versions. For each functional component that implements a behavior, Beagle can produce results similar to those presented for the overall malware code in Table 2.2 and in Figures 2.2 and 2.3. To present the evolution of behaviors, we focus on the similarity (as defined in Equation (2.4)) between each version of the behavior and the *reference behavior*. As discussed in Section 2.3.4, the reference behavior is the largest implementation of a behavior by number of basic blocks, across a malware family's versions. While this does not provide a complete picture of the code's evolution, it gives an idea of how each behavior grows towards its largest implementation (i.e., the reference behavior).

Figure 2.5 shows the similarity over time of each behavior found in *ZeuS* (3^{rd} variant) against the respective reference behavior. This shows the contribution of each behavior to the overall changes. Interestingly, in this family as well as in other families, we notice very limited code change overall (first timeline). However, the breakdown in individual behaviors reveals some significant changes towards the end of the observation window, where behaviors such as `TCP_CONNECTION`, `DOWNLOAD_INJECT` and `HTTP_REQUEST` change their similarity with respect to the reference behavior. In some way, these changes "compensate" against each other, resulting in a quite constant overall similarity.

A more compact representation of the behavior "variability" is exemplified in Figure 2.6, which shows the boxplot distribution of the similarity of each behavior against the respective reference behavior. As we observed before when looking at the timeline of the similarity for *ZeuS* (3^{rd} variant) in Figure 2.5, in this family all behaviors except `AUTO_START` changed throughout our observation period, with `EXECUTE_TEMP_FILE`, `ENUMERATE_PROCESSES`, `REMOVE_FLASHPLAYER_FILES`, and `INJECT_CODE` showing the most code changes and thus development effort (Figure 2.6a). In the second family under examination, which is *Gamarue* (Figure 2.6b), behaviors such as `DOWNLOAD_EXECUTE`, `UDP_TRAFFIC`, and `DOWNLOAD_FILE` almost never change, except for some outliers (empty circles). Other behaviors, such as `CHANGE_SECURITY_POLICIES` and `DISABLE_TASKMGR`, exhibit more variance, which means that their code is changed often, corresponding to a proportionally larger development effort.

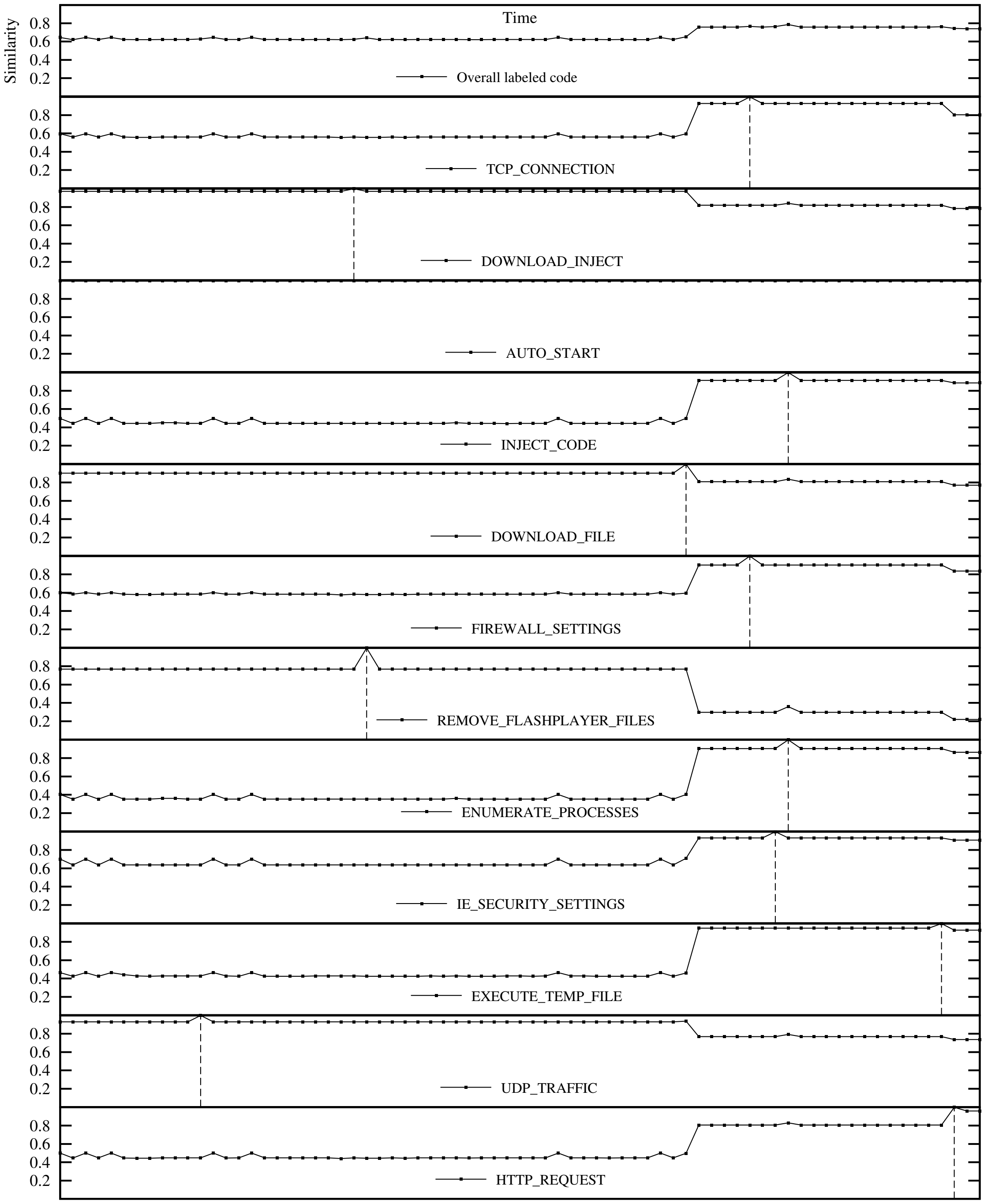
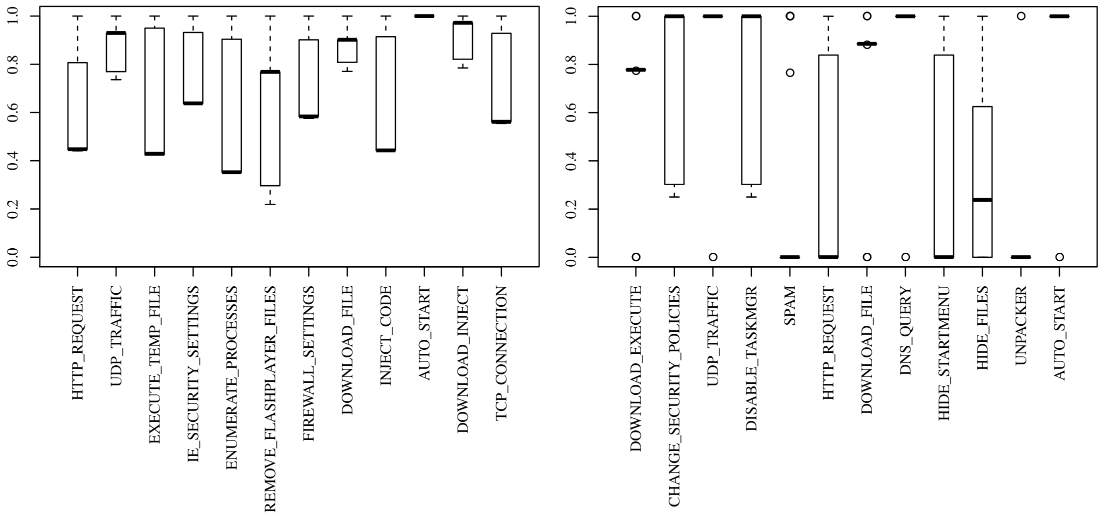


Figure 2.5: *Similarity over time of each behavior against the respective reference behavior for ZeuS (3rd variant).* The first timeline is the overall code similarity with respect to the first sample of that family. The remaining timelines show how the overall changes are broken down into changes in the single behaviors of this family. Despite a very limited code change overall we observed significant changes in the individual behaviors.



(a) *ZeuS* (3^{rd} variant) **(b)** *Gamarue*

Figure 2.6: *Distribution of the similarity of each behavior against the respective reference behavior (behavior "variability").* Each box marks the 0%-, 25%- and 75%-, 100%-quantiles, and the median. Circles indicate outliers.

2.4.8 Lines of Malicious Code

Throughout our evaluation, we have used basic blocks as the unit of measurement for code, whereas it would be more useful to quantify the malware development effort in terms of lines of malicious code. Unfortunately, directly measuring the Lines of Code (LoC) would require the malware's source code, which is typically unavailable. Nonetheless, we would like to provide a rough estimate of the amount of source code that may correspond to the observed changes. For this, we attempt to estimate a range of possible values for the ratio of basic blocks to LoC in malware samples. Clearly, factors such as the compiler, compiler options and programming paradigms can significantly influence such ratio, thus our estimate is not generalizable outside the scope of our dataset.

We obtained the source code of 150 malware samples of various kinds (e.g., bots, worms, spyware), including the leaked *ZeuS* source code, and a corresponding executable binary. Apart from *ZeuS*, none of the families in our dataset are represented in these 150 samples. We then calculated the number of lines of C/C++ source code (LoC), using `cloc`, and the number of basic blocks, using Beagle's binary comparison submodule (excluding the blocks that belong to whitelisted fingerprints). Samples in this dataset contain an average of 11,678.12 LoC, with *ZeuS* being the most complex sample with 48,196 LoC in total. We further found that the ratio between basic blocks and LoC ranges between 50 and 150 blocks per LoC, and around 14.64 for *ZeuS*. However, one third of the 150 samples that we analyze exhibit a ratio very close to 50 basic blocks per LoC, which we take as a conservative lower bound.

Given these estimates, in the case of *ZeuS*, the average amount of new code is around 140–280 LoC, with peaks up to 9,000 LoC. Since the *ZeuS* samples in our dataset are closely related to the source code used to estimate this ratio, we consider this a relatively reliable estimate. For the remaining families, with a rough estimate using a ratio of 50, we notice an average amount of *new* code around 100–300 LoC per update cycle, with peaks up to 4,600–9,000 LoC. Although these are just estimates, they give an overall idea of the significant development effort behind the evolution of malware.

2.4.9 Discussion

Beagle revealed that, within our observation window, some families are much more actively developed than others. For instance, *GenericTrojan*, *Llac* and *ZeuS* (3^{rd} variant) have a remarkable amount of new code added. A wider observation window may obviously unveil other "spikes" of development efforts (e.g., new code and functionality being added). As discussed in Section 2.4.8, for some families such as *ZeuS*, we can also estimate these quantities in lines of source code.

Beagle is also able to tell whether and how such changes target each individual behavior. For instance, some behaviors in certain families almost never change (e.g., `UDP_TRAFFIC` or `DOWNLOAD_FILE` in *Gamarue*), whereas other behaviors change over time (e.g., `HIDE_STARTMENU` or `HTTP_REQUEST` in *Gamarue*). In some cases, such as *ZeuS* (3^{rd} variant), the overall development effort appears constant and relatively low. Beagle's more focused analysis of the evolution of individual components of this malware family, however, reveals that some behaviors undergo significant changes.

2.5 Limitations and Future Work

Our results demonstrate that Beagle is able to provide insight on the real-world evolution of malware samples. However, malware authors could take steps to defeat our system. First of all, the simple unpacking techniques used by Beagle could be defeated by using more advanced approaches such as multi-layer or emulation-based packing. To counter this limitation, Beagle could be extended with advanced unpacking approaches [135, 181]. A further limitation is that malware analyzed by Beagle may be able to detect that it is running in an analysis sandbox and refuse to update. In recent years, a number of techniques have been proposed to attempt to detect [28, 120] or analyze [63, 111] evasive malware.

Even in the absence of evasion, Beagle's dynamic analysis component suffers from limited code coverage. This problem is to some extent alleviated by the fact that we combine information on a malware sample from a large number of executions over a period of months. Nonetheless, behavior that is never observed in our sandbox cannot be analyzed. For instance, in our current experiments our observation of the `BITCOIN_WALLET` behavior is incomplete because the analysis environment does not include a Bitcoin wallet for the malware to steal.

Beagle can identify and measure the evolution of a malware's functional components. However, it cannot tell us anything more about the semantics of the code changes it detects: This task is currently left to a human analyst. The next logical step is to develop patch analysis techniques to attempt to automatically understand how the update of a component changes its functionality.

2.6 Conclusion

Understanding the mechanics and economics of malware evolution over time is an interesting and challenging research problem with practical applications. We proposed an automated approach to associate observed behaviors of a malware binary with the components that implement them, and to measure the evolution of each component across malware versions. To the best of our knowledge, no prior research exists that can automatically monitor how malware components change over time.

Our system can observe the overall evolution of a malware sample and of its individual functional components. This led us to interesting insights on the development efforts of malicious code. Our measurements confirmed commonly held beliefs (e.g., that the malware industry is partly driven by AV advances), but gave also novel and interesting insights: We observed that most malware authors reuse significant portions of code, but that this varies wildly by family, with significant "spikes" of software development in short timespans. We were also able to distinguish between behaviors that never change in a certain family, and others that are constantly being updated. In some cases, spikes of development of a malware component are not visible in the overall evolution of the malware sample, but are revealed by the analysis of individual behaviors.

Beagle proved to be useful both to build the "big picture" of how and when self-updating malware changes, and to guide malware analysts to the most interesting portions of code, i.e., parts that have changed significantly between two successive versions.

CHAPTER 3

Andrubis: A View on Current Android Malware Behaviors

Android is undoubtedly the most popular operating system for smartphones and tablets with a market share of over 80% [101]. Its widespread distribution and wealth of application (app) distribution channels besides the official Google Play Store, however, also make it the undisputed market leader when it comes to mobile malware: According to a recent estimate, as many as 97% of mobile malware families target Android [76]. Estimates by antivirus (AV) vendors regarding the number of Android malware in the wild vary widely. In 2013 McAfee reported about 68,000 distinct malicious Android apps [138] and Sophos collected a total of 650,000 unique Android malware samples until this date, with 2,000 new samples being discovered every day [192].

Google reacted to the growing interest of miscreants in Android by introducing *Bouncer* [128], a service that transparently checks apps submitted to the Google Play Store for malware. Google reported that this service led to a decrease of the share of malware in the Play Store by nearly 40% since its deployment in February 2012. However, Android users are not limited to the official Google Play Store when it comes to installing apps. Apps are available from arbitrary sources—including a wealth of alternative markets. A common practice among malware authors is *repackaging* popular apps with malicious code and publishing them in alternative markets that do not employ effective security measures. In fact, as we will discuss in Chapter 5, in line with findings from F-Secure [76], we found alternative markets hosting up to 5-8% malicious apps [125].

Consequently, a significant amount of research has focused on analyzing and detecting Android malware, with numerous tools and services being proposed and operated by researchers [37,68,168,186,214] and security companies [5,6,12]. Automated and reliable solutions are required to deal with the growing number of mobile malware samples. Analysis capabilities and availability of proposed research prototypes, however, remain limited. A recent study on state-of-the-art Android malware analysis techniques showed that among the 18 analysis tools surveyed, many systems were not available online or were no longer being maintained [145]. In an evaluation on the susceptibility of Android dynamic analysis sandboxes against evasion, Vidas et al. [204] only found three publicly accessibly systems (including the one presented here).

In order to provide a large-scale analysis solution to the research community we propose Andrubis, a hybrid Android malware analysis sandbox that generates detailed analysis reports of unknown Android apps based on features extracted during static analysis and behavior observed through dynamic analysis during runtime. Similar to the spirit of AndroTotal [133], a service that allows researchers to scan Android apps with a number of AV scanners, we operate Andrubis as a publicly available service and data collection tool that allows us to collect and share a comprehensive and diverse dataset of both Android malware and benign apps.

We built Andrubis as an extension to the dynamic Windows malware analysis sandbox Anubis [32, 34]. Anubis has collected a dataset of Windows malware samples that represent a comprehensive and diverse mix of malware found in the wild since 2007 [31]. Andrubis itself has been online since June 2012 and has analyzed over 1,000,000 unique Android apps in its first two years of operation. Based on AV labels collected from VirusTotal [11], we estimate that at least 40% of those apps are malware (not including adware). We further assess the age of apps in our dataset and categorize them by year starting in 2010 allowing us to identify trends in Android malware behavior. Similar to the dataset of Anubis, our dataset represents apps from a variety of sources, with apps collected from crawls of the Google Play Store and alternative markets, sample exchanges with other researchers, torrents and direct downloads, and anonymous user submissions.

The tight integration of our analysis with the existing Anubis infrastructure for analyzing Windows malware provides two main benefits: (a) We can take advantage of existing sample exchange agreements as malware feeds often contain both Windows and mobile samples, and (b) adapt existing analysis techniques for the use with Android apps. For example, experiments applying clustering [33] to Android apps yielded promising results and showed that the feature set produced by Andrubis is rich enough to allow researchers to build various post-processing methods upon [210]. This last aspect is of particular importance as we envision Andrubis to be integrated with other analysis tools to foster sample exchange and provide deeper insights into Android malware behavior. Andrubis has already been integrated with different tools, such as AndroTotal to provide an additional analysis report to AV scanner results. Similarly, Andrubis provides a seed of malicious apps to AndRadar [125] (presented in Chapter 5), which it uses to scan the Google Play Store and 15 alternative markets and that in turn allows us to collect valuable meta information for our dataset. Besides shedding light on publishing habits of malicious app authors we can gain insights on an app's distribution across markets and popularity according to user ratings and download numbers. In the future, we also hope to gain insights into the infection rates of user's devices by analyzing which apps are submitted directly from user's phones. Thereby we might be able to verify reports of the small infection rates of less than 0.3% reported in related work [118, 131, 198].

In summary, we make the following contributions:

- We introduce ANDRUBIS, a fully automated analysis system that combines static and multi-layered dynamic approaches to analyze unknown Android apps.
- We provide ANDRUBIS as a large-scale analysis service to the research community, accepting public submissions at `https://anubis.iseclab.org` and through a mobile app that is available in the Google Play Store.[1]
- By collecting apps from a variety of sources we build a comprehensive and diverse dataset of over 1,000,000 Android apps, including over 400,000 malicious apps.
- We present insights gained from providing our service over a observation period of two years and we discuss trends in malware behavior observed from apps dating back as far as 2010.

3.1 Andrubis System Overview

In this section we detail the building blocks of ANDRUBIS and how they contribute to forming a complete picture of an app's characteristics.[2] Analyzing Android malware follows the same basic principle that analyzing Windows malware relies on. On the one hand, *static analysis* yields information immediately by examining a sample's application package and code, while *dynamic analysis* executes the sample in a sandbox and provides details on its behavior during runtime. ANDRUBIS follows the *hybrid analysis* approach and is based on both static and dynamic analysis complementing and guiding each other: Results of the static analysis are used to perform more efficient dynamic analysis. Figure 3.1 shows an overview of the individual components of ANDRUBIS and how they relate to one another. Users can submit apps either through our web interface, automated batch submission scripts, or directly from their phone through a dedicated mobile app. We then subject each app to the following three analysis stages:

1. **Static analysis.** During this stage we extract information from an app's manifest and its bytecode.
2. **Dynamic analysis.** This core stage executes the app in a complete Android environment, and its actions are monitored at both the Dalvik VM and the system level. We also implement various stimulation techniques to improve the code coverage of dynamic analysis.
3. **Auxiliary analysis.** We capture the network traffic from outside the Android OS and perform a detailed network protocol analysis during post-processing.

[1] `https://play.google.com/store/apps/details?id=org.iseclab.andrubis`

[2] We previously published a version of this system description as a technical report [210].

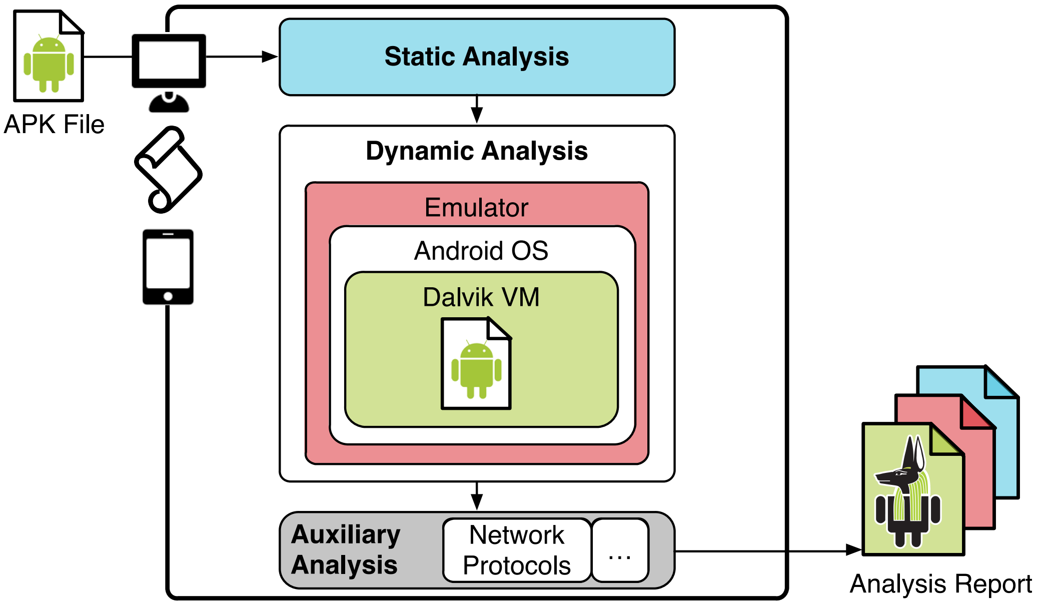


Figure 3.1: *System overview of* Andrubis. In the first step Andrubis performs *static analysis* on each submitted app. Results of this step are then used to guide the *dynamic analysis*, which performs monitoring at Java-level through a modified Dalvik VM as well as at system level through VMI in the emulator. Lastly, the *auxiliary analysis* post-processes the analysis results and enhances them by a detailed network analysis.

3.1.1 Static Analysis

Android apps are packaged in *Android Application Package* (APK) files, a ZIP archive based on the JAR file format. An APK file contains an app's bytecode stored in Dalvik Executable (DEX) format, resources, such as UI layouts, as well a manifest file (`AndroidManifest.xml`). The manifest is mandatory and without its information an app cannot be installed or executed. Thus, as a first step, we unpack the archive and parse meta information from the manifest, such as requested permissions, services, broadcast receivers, activities, package name, and SDK version. In addition we examine the actual bytecode to extract a complete list of available Java objects and methods.

We use the information gathered during static analysis to assist in automating the dynamic analysis, mainly during the stimulation of an app's components. Furthermore, an app requesting dangerous permissions can be indicative of malicious behavior. Therefore, we extract the permissions that are requested as well as the permissions that are actually used in the app's bytecode to later compare them to permissions used during runtime. Finally, we flag apps that make use of functionality that can make static analysis more difficult: We look for the usage of reflection, which allows apps to inspect classes and invoke methods at runtime, essentially hiding them from most static analysis tools, and the invocation of the cryptographic API as well as APIs that allow the dynamic loading of code, both DEX classes and native code.

3.1.2 Dynamic Analysis

Being designed for smartphones and tablets, Android is predominantly deployed on ARM-based devices. Since the underlying architecture should be of no difference to the apps, we decided to build our sandbox for the ARM platform, the typical environment for Android, and chose a QEMU-based emulation environment capable of running arbitrary Android OS versions. Since Android apps are based on Java, we instrument the underlying virtual machine (VM), called the Dalvik VM, and record activities happening within this environment. This allows us to monitor the file system and network, as well as phone events, such as outgoing SMS messages and phone calls, and the loading of additional DEX or native code during runtime. For a comprehensive analysis, however, these capabilities are not sufficient. Therefore, we implemented the following additional analysis facilities:

- **Stimulation.** Due to the event-driven nature of Android, comprehensive input stimulation is invaluable for triggering interesting behavior from the app under analysis.

- **Taint tracking.** To track privacy sensitive information Andrubis uses taint tracking at the Dalvik level [71], which enables us to detect the leakage of sensitive information.

- **Method tracing.** We record invoked Java methods, their parameters, and their return values. Combined with our static analysis, we can use method traces to measure the code covered during an analysis run, e.g., for evaluating and improving our stimulation engine.

- **System-level analysis.** To provide means for analysis beyond the scope of the Dalvik VM, we implemented an introspection-based solution at the emulator level. This enables us to monitor the system from outside the Android OS and to track system calls of native libraries and root exploits.

The output produced by the method tracer and the system-level analysis is not displayed in the public Andrubis analysis report. As these tasks are quite resource-intensive and the log files are quite large, we only perform them on a subset of samples and provide them on an on-demand basis for researchers and analysts rather than ordinary users.

The remainder of the sandboxing system (network setup and traffic capturing, host environment, database, etc.) is comparable to conservative analysis systems. To mitigate potentially harmful effects of our analysis environment to the outside world while allowing apps under analysis to use the network, we took precautions to prevent apps from executing DoS attacks, sending spam e-mails or propagating themselves over the network. This part is based on our experience with Windows malware analysis and proved to be effective with Anubis in the past [32, 34].

Stimulation Event	Target
Activities	Activities declared in the manifest
Services	Services declared in the manifest
Broadcast Receivers	Receivers declared in the manifest and registered dynamically during runtime
Common Events	Boot completion, SMS, phone calls, WiFi+3G connectivity, GPS lock
Random Input	Random input stream from the Application Exerciser Monkey, e.g., clicking buttons

Table 3.1: *Stimulation events performed by* Andrubis.

Stimulation

The purpose of stimulation is to exhaustively explore the functionality of an app. One major drawback of dynamic analysis in general is the fact that only a few of all possible execution paths are traversed within one analysis run. Furthermore, Android apps can have multiple entry points besides the main activity, which is displayed to the user when an app is launched, so that apps can react to system events or interact with each other. Luckily, since the app's manifest lists the various components (activities, services, and broadcast receivers), we can stimulate them individually. Additionally, we can initiate common events that malicious apps are likely to react to.

Our stimulation approach includes the following sequence of events: After the initialization of the emulator, Andrubis installs the app under analysis and starts the main activity. At this point, all predefined entry points are known from static analysis. During runtime Andrubis keeps track of dynamically registered entry points, enabling it to perform the following stimulation events (summarized in Table 3.1):

Activities. An activity provides a screen to interact with and defines the interaction sequences and UI layout presented to the user. Activities have to be registered in the manifest and cannot be added programmatically. Therefore, by parsing the manifest, Andrubis has full knowledge about an app's activities and invokes each activity separately, effectively iterating all existing dialogs within an app.

Services. Background processes on the Android platform are usually implemented as services. In contrast to activities, they come without a graphical component and are designed to provide background functionality for an app. Naturally, they are also of interest to malware authors, as they can be used to implement communication with command and control (C&C) infrastructures of botnets, leak personal information, or forward intercepted text messages to an adversary. Again, all services used by an app must be listed in the manifest. Their existence, however, does not automatically mean the service is started: To save battery life and preserve memory, services have to be started on demand, with a lifetime defined by the programmer. For Andrubis we utilize a customized *Activity Manager* to iterate and start all listed services of an app automatically after it has been installed.

Broadcast receivers. Other possible entry points for Android apps are broadcast receivers. Broadcast receivers are basically event handlers used to receive events from the system or other apps on the Android platform. For example, a broadcast receiver for the `BOOT_COMPLETED` event can be registered to start an app after the phone has finished its boot sequence or a broadcast receiver for the `SMS_RECEIVED` event can be registered to intercept incoming SMS messages.

Just like services and activities, broadcast receivers can be registered in the manifest. However, for broadcast receivers this is not mandatory. In order to provide the possibility to react to certain events, or to provide communication with other apps dynamically, they can also be registered and deregistered at runtime. Therefore, we intercept the calls to `registerReceiver()` to obtain a list of dynamically registered event handlers that we can stimulate. Similar to the previous stimuli, Andrubis uses the *Activity Manager* to invoke all statically registered broadcast receivers found in the manifest as well as the ones that have been dynamically registered.

Common events. A far superior method compared to directly stimulating broadcast receivers with a targeted event is to emulate the events that apps might react to and especially malicious apps are likely to be interested in. Thus, we broadcast events such as boot completion, incoming SMS and phone calls, changes in the GPS lock, and changes in the WiFi and cellular connectivity. In contrast to directed stimuli, these events occur at the system level and thus also trigger receivers of the Android OS itself. That, in turn, avoids causing inconsistent states the OS would have to recover from when only invoking the event handler registered by an app.

Random input. The remaining elements that need to be stimulated are actions based on user input (button clicks, file upload, text input, etc.). For this purpose, we use the *Application Exerciser Monkey* [91], which is part of the Android SDK and generates semi-random user input. Originally designed for stress-testing apps, it randomly creates a stream of user interaction sequences that can be restricted to a single package name. While the triggered interaction sequences include any number of clicks, touches, and gestures, the monkey specifically tries to hit buttons. As some use cases might require repeatable analysis runs without any random behavior introduced by the monkey, we optionally provide a fixed seed in order to always trigger the same interaction sequences.

Taint Tracking

Data tainting is a double-edged sword when it comes to malware analysis. On one hand, it is the perfect tool to keep track of interesting data; on the other hand, it can be tricked quite easily if a malware author is aware of this mechanism within an analysis environment [44]. By leaking data through implicit flows, for instance, it would be possible to circumvent tainting. Furthermore, enabling data tainting always comes at the price of additional overhead to produce and track taint labels. Still, the possibility to track explicit flows of sensitive data sources, such as contacts, phone-specific identifiers, and the location, to the network is a valuable property of a dynamic analysis system. Andrubis leverages *TaintDroid* [71] to track such sensitive information across application borders in the Android system. The introduced overhead in processing time of approximately 15% [71] is also acceptable for our purposes. As a result, Andrubis can log tainted information as it leaves the system through three sinks: network, SMS, and files on disk.

Method Tracing

For an extensive analysis of Java-based operations, we extended the existing Dalvik VM profiler capabilities to incorporate a detailed method tracer as proposed by TraceDroid [201]. For a given app we log the executed Java methods on a per-thread basis. The method trace contains method names and their corresponding classes, the object's `this` value (if any), all provided parameters and their types, return values, constructors, exceptions and the current call depth. For non-primitive types, the tracer looks up and executes the object's `toString()` method, which is then used to represent the object.

Together with the output gained from system-level analysis (described in the next section), the fine-grained method traces can assist reverse engineering efforts, serve as input to machine learning algorithms, or they can be used to create behavioral signatures. Furthermore, by mapping the method trace to permissions utilizing a permission mapping, such as the ones provided by PScout [25] or Stowaway [78] we can determine the permissions an app actually used during runtime.

Our main incentive to integrate method tracing, however, is to measure the code covered during the individual phases of the stimulation engine. To this end, we first compile a list of executed method signatures. We then map this list against the list of functions extracted during static analysis based on their Java method signature excluding parameter types and modifiers, i.e., on their `<package>.<subpackage>.<class>.-<method>` representation. Finally, we compute the code covered as the overall percentage of functions that were called during the dynamic analysis. Using this methodology we calculated an average code coverage of about 28% on a dataset of 500 benign and malicious apps in previous experiments [210]. However, apps may contain numerous functions that, during a normal execution, will never be invoked, such as localization and in-app settings or large portions of unused code from third-party libraries. Thus, for a less conservative and more realistic code coverage computation we can whitelist known third-party APIs or limit the computation to the main app package's code.

System-Level Analysis

In addition to monitoring the Dalvik VM, and in contrast to most related work on Android malware, Andrubis also tracks native code execution. By default, Android apps are Java programs, being distributed as a DEX file within an APK file. Hence, the default way of programming for the Android platform and executing Android apps is by running Dalvik bytecode within the Dalvik VM. However, Android apps are not limited to Dalvik bytecode and can also execute system-level code by loading native libraries via the Java Native Interface (JNI). While this functionality is mainly intended for performance-critical use cases, such as displaying 3D graphics, apps are not restricted to loading the native libraries shipped with the Android OS; instead, they can also ship and load their own native libraries and, in turn, execute arbitrary system-level code. Naturally, the execution of this code takes place outside of the Dalvik VM and, thus, the behavior of this code is invisible to analysis at Dalvik VM level. For malicious apps the use of native code is attractive as the possibilities to perform malicious activities, such as the usage of exploits to gain root privileges, are far greater than within the Dalvik VM—making system-level analysis indispensable for drawing a complete picture of an apps's behavior. In addition, Google recently introduced the new Android Runtime (ART) [88] that compiles Dalvik bytecode to native code at installation time. With the replacement of Dalvik with ART as the default runtime in the latest Android OS releases [196], the capability to perform system-level analysis will gain further importance.

Being based on Linux, there are a couple of ways to implement system-level instrumentation in Android, such as using `LD_PRELOAD`, `ptrace` or a loadable kernel module. We decided to use the most transparent and non-intrusive way—virtual machine introspection (VMI). With VMI our analysis code is placed outside of the scope of the running Android OS, right in the emulator's codebase, and tracks the complete list of system calls performed by the emulator as a whole, including the OS. To capture the system-level behavior of the app under analysis, we ultimately need to extract the system calls executed by the library code that was loaded via JNI. To this end, we intercept the Android dynamic linker's actions to track shared object function invocations. System call tracking bundled with this information enables us to associate system calls with invocations of certain functions of loaded libraries. Android assigns a unique user ID (UID) to every app and runs the app as that user in a separate process—allowing us to associate system calls with apps based on the process UID. The result is a list of native code events caused by just the specific app under analysis.

3.1.3 Auxiliary Analysis

Network traffic is one of the most essential parts when establishing malware-detection metrics, with C&C communication being of particular importance. According to studies performed in production environments [87], more than 98% of Windows malware samples established a TCP/IP connection. Thus, in addition to tracking sensitive information to network sinks via taint tracking, we also capture all the network activity during analysis regardless of the performed action or the app causing it. This is necessary since apps not requesting and using the `INTERNET` permission themselves, can still use other installed apps like the browser, to send data over the network. Another way to transmit network data without requesting the appropriate permissions is by exploiting the Android OS and circumventing the permission system as a whole.

During post-processing we perform a detailed *Network Protocol Analysis* with customized *Bro* [152] scripts that extract high-level network protocol features from the captured network traffic suitable for identifying interesting samples. Currently, we focus on the well-known and often used protocols DNS, HTTP, FTP, SMTP, and IRC.

3.2 Andrubis as a Service

In this section we present insights gained from offering Andrubis as a publicly available service for two years and the dataset of over 1,000,000 apps we collected along the way.

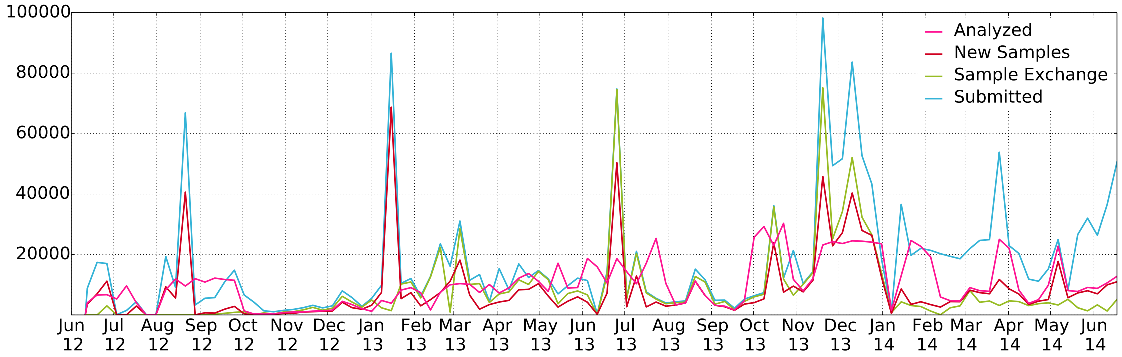


Figure 3.2: Andrubis *submission statistics.* We show the weekly number of total submissions, submissions through sample exchanges (i.e., semi-regular feeds of samples), new and analyzed samples.

3.2.1 Submission Statistics

We base our analysis on a dataset collected over the span of exactly two years, between June 12, 2012 and June 12, 2014. We distinguish between *submissions* (all analysis requests Andrubis received), *tasks* (submissions for which the analysis was performed), and *samples* (unique apps based on their MD5 file hash). Overall, Andrubis received 1,778,997 unique submissions. Since Andrubis usually returns cached analysis reports in case an app is submitted multiple times (unless a user requests a re-analysis of a previous task), it performed analysis tasks for 1,073,078 (around 60%) of submissions. In total Andrubis received and analyzed 1,034,999 unique samples (58.18% of submissions).

To put the number of Android samples into perspective we compare them to overall submissions to Anubis. During our observation period Anubis received a total of over 22 million samples, Android apps thus amount to close to 5% of overall samples. However, since the submission interface only assigns submissions of ZIP archives containing `classes.dex` and `AndroidManifest.xml` to Andrubis, we only report numbers on APK files and not submissions of related files such as stand-alone DEX classes. A large number of submissions to Anubis come from malware feeds as part of exchange agreements. We receive feeds with Android apps from nine sources, most of them submitting both Windows executables and Android apps—with the exception of AndroTotal almost exclusively submitting APK files. Other malware feeds from security researchers and AV vendors contain from as little as 1% to up to 37% Android apps. The largest sample feed contributing more than five million samples to Anubis in the observation period contains around 10% Android submissions.

Figure 3.2 shows the timeline of submissions to Andrubis on a weekly basis. Submissions peaked in August 2012 and January 2013, when we received bulk submissions from Google Play crawls and in July 2013 when one feed submitted a higher than usual amount of samples. In November and December 2013 AndRadar started submitting a backlog of apps before switching to a regular feed of apps. Besides a power outage in January 2014 Andrubis has been operating reliably and analyzed up to the current maximum capacity of 3,500 new apps per day.

Category	# of Submissions	# of Users	% of All Submissions
Bulk	10,000+	15	95.82%
Large	1,000-10,000	13	2.47%
Medium	100-1,000	34	0.59%
Small	10-100	247	0.41%
Single	1-10	7,966	0.72%

Table 3.2: *Characterization of* ANDRUBIS *users.* We categorize users by the number of their submissions and their share of all submissions.

In order to estimate the number of different users using our analysis service, we distinguish them either by their username, or in case of anonymous submissions, by their IP address. Users can register for an account in order to gain special privileges, such as a higher priority for their tasks or the ability to force the re-analysis of an app. The account management is shared with ANUBIS, but 152 registered users submitted at least one Android app. With anonymous submissions coming from 8,123 unique IP addresses we estimate that 8,275 unique users from 130 different countries are using ANDRUBIS. The majority of submissions come from registered users, with 15 individual users amounting to over 95% of total submissions, and only 38,905 (3.76%) of submissions coming from anonymous sources. Table 3.2 categorizes users by their number of submissions from single submitters with less than 10 submission to “power users” with more than 10,000 submissions. The maximum amount of 557,559 submissions for a single user stems from one of the aforementioned malware feeds. In total 4 users submitted more than 100,000 files each.

Figure 3.3 shows the number of different sources, i.e., the number of distinct users that submitted a particular app: Around 70% of apps were submitted by only one user and only 1.5% of apps were submitted by more than three distinct users. In general, malicious samples were submitted more frequently than goodware apps (with the exception of the top two apps): Over 80% of apps submitted by more than five different users belong to the malware category. The popular game Flappy Bird was also the “most popular” app submitted to ANDRUBIS by 88 different users. The second most submitted app with 74 unique submissions is the alpha version of our mobile interface to ANDRUBIS, with which users can submit apps directly from their phones (it was available for download on the front page of the ANDRUBIS’ web interface during our experiments before we published it in the Google Play Store). However, the remaining most popular apps (and all other apps submitted 26 times and more) are part of malware corpora, such as Contagio [3] and the Android Malware Genome Project [227].

3.2.2 Analysis Results and Limitations

Overall, ANDRUBIS successfully analyzed 91.67% of all apps. For the remaining samples, 0.34% failed due to bugs in our analysis environment and 7.99% of samples failed to install in our sandbox due to various reasons, such as the APK file being corrupt or the app exceeding the API level of the Android OS version installed in our sandbox.

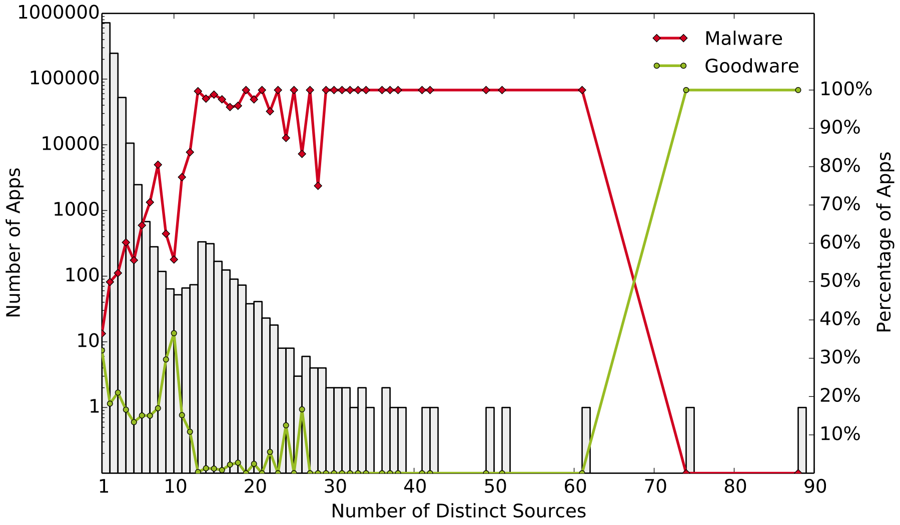
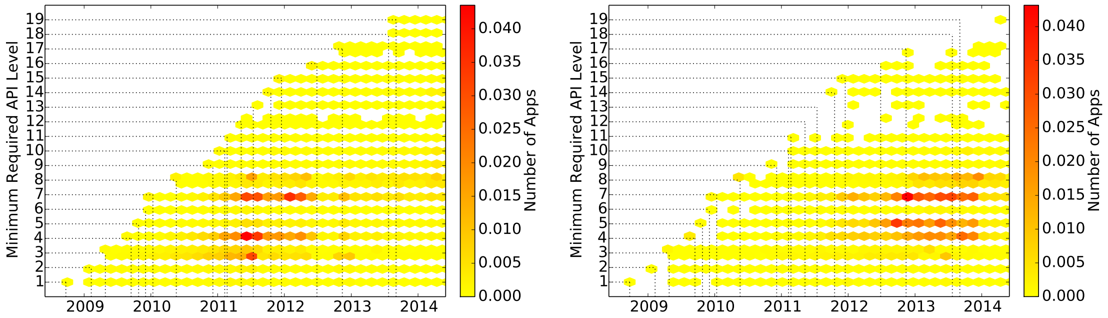


Figure 3.3: *Frequency of submissions per app.* Bars show the number of apps submitted by N unique users, lines show the percentage of goodware and malware submitted by N users. With two exceptions malicious samples are submitted more frequently than goodware apps, especially samples from public malware corpora.



(a) Goodware

(b) Malware

Figure 3.4: *Heat map of API level adoption after Android SDK releases for goodware and malware.* Goodware authors adopt new API levels much faster than malware authors, who try to maximize the potential user base for their apps.

Andrubis currently runs Android 2.3.4 Gingerbread and thus only supports apps with a minimum required API level ≤ 10. We know that 0.78% of apps require a newer OS version, and for 6.66% of samples we could either not parse the manifest or they did not specify an API level. However, this has no significant impact on malware analysis as of now. Instead, it is mainly a concern for goodware, of which 2.11% (6,099) require a higher API level, while only 0.10% (439) of malicious apps fail for this reason. Such a behavior by malware authors is expected: Their malicious apps require a lower API level to maximize the potential user base for their apps, and, in turn, their profit. This is also confirmed by Figure 3.4, which shows that malware authors are much slower in adopting new API levels than goodware authors.

Category	Sample Exchange	Google Play	Alternative Markets	VirusTotal	Malware Corpora	Torrents	Direct Downloads	Unknown
All	683,842	125,602	60,951	37,499	5,997	17,916	1,704	159,040
Goodware	5.20%	88.73%	18.15%	0.20%	0.04%	88.50%	96.36%	78.65%
Malware	55.30%	1.60%	27.51%	98.65%	97.87%	1.60%	1.59%	7.56%
Other	39.50%	9.67%	54.34%	1.15%	2.09%	9.90%	2.05%	13.79%

Table 3.3: *Sources for apps in our dataset.* Sample exchange feeds have a high proportion of malware, while Google Play, torrents, and direct downloads have low infection rates. Interestingly, AV scanners detect not all samples from malware corpora.

3.2.3 Scalability

Currently, Andrubis is capable of processing around 3,500 new apps per day, i.e., apps that have never been analyzed before and for which no cached report is available. The analysis of an app takes around 10 minutes, with 240 seconds analysis runtime in the sandbox plus an additional 387.27 seconds on average for pre- and post-processing. Pre-processing includes setting up the emulator and loading the Android OS snapshot, installing the app, parsing the manifest and performing static analysis on the APK. Post-processing includes extracting protocol information from the network traffic and preparing the final analysis report.

Judging from our experience running the Windows malware analysis service Anubis, and similar to Andlantis [36], Andrubis scales well by simply adding new workers to handle the analysis of new samples should submissions increase. However, already with the current throughput of over 100,000 apps per month, Andrubis is capable of analyzing samples at market scale. For example, Google Play, the largest app store (by far), added, on average, 37,500 new apps per month from June 2013 to June 2014, with peaks of up to 85,000 new apps in December.[3] When it comes to malware, Android still falls far behind the plethora of Windows samples circulating in the wild: Sophos estimates 2,000 new Android malware samples are being discovered each day [192], a number Andrubis can handle in the current configuration and setup comfortably.

3.2.4 Sample Sources

One limitation of a public web interface allowing anonymous submissions is the lack of meta information associated with submitted apps. Since the majority of apps are submitted by registered users, however, we can associate them to sample exchanges, part of our own crawling efforts, or the integration of tools, such as AndRadar. Table 3.3 summarizes the number of apps from each source, as well as the proportion of benign, malicious, and other apps (see the next section on how we separated goodware from malware). The apps in our dataset originate from the following eight sources:

[3] http://www.appbrain.com/stats/number-of-android-apps (retrieved July 1, 2014)

Sample exchange. These apps make up the majority of our dataset and come from sample sharing with other researchers. Most feeds are part of long-standing sample exchanges that started with Windows samples, but now also include Android apps.

Google Play. We initially crawled 100,000 apps from the Google Play US Store during May and June 2012. Additionally, since December 2013, we receive apps crawled from AndRadar that match a seed of malicious apps and are located in the Google Play Store. In April 2014, we started fetching the top apps overall, top new apps and top apps per category (limited to 500 entries each by Google Play) from the Google Play US and AT Store on a daily basis.

Alternative markets. These apps are crawled by AndRadar from 15 alternative markets, including seven Chinese and one Russian market. This dataset is biased towards malware since AndRadar aims at locating malicious apps in alternative markets.

VirusTotal. We regularly download samples from VirusTotal. However, this dataset not only contains malware, but also a small percentage of samples labeled as adware as well as some samples not detected by any AV scanner.

Malware corpora. This is a collection of manually gathered malware samples we encountered over time as well as samples from vetted malware corpora, such as the Contagio Mobile Malware Dump [3], the Android Malware Genome Project [227], and Drebin [22]. However, besides the relatively small Contagio set (470 samples) that is regularly updated, which, in turn, makes comparison hard, available malware corpora are already quite dated: The 1,200 samples (49 different families) from the Android Malware Genome Project were collected from August 2010 to October 2011, the 5,560 apps (179 families, including the Genome Project) from the Drebin dataset were collected in the period of August 2010 to October 2012. Therefore, they are not representative for current malware behavior.

Torrents. We downloaded apps through torrents we crawled from `thepiratebay.se`, `isohunt.com`, and `torrentz.eu`, and which had at least ten seeders. To avoid distribution of copyright-protected content, our torrent client did not upload any data.

Direct downloads. We downloaded a set of apps through direct downloads from various one-click hosters, including `filestube.com` and `iload.to`.

Unknown. These apps stem from anonymous user submissions and thus we do not have any information where they originate from.

3.2.5 Collected Dataset

The dataset gathered from samples submitted to Andrubis allows us to perform a longitudinal analysis of Android app features in general and of features specific to benign apps and malicious apps. First, however, we need to separate the dataset into subsets. Since the primary goal of Andrubis is to provide researchers with a comprehensive static and dynamic analysis report of an app, not to automatically identify apps as goodware or malware, we have to rely on AV signatures as our ground truth:

Goodware. We classify apps as goodware if they do not match any AV signature from VirusTotal's AV scanners. Goodware apps make up 27.90% of our dataset.

Malware. We classify apps as malware if they match at least t AV signatures. We experimented with different settings for the threshold t and settled on at least 5 AV labels, ignoring all AV labels indicating adware. With thresholds $t > 5$ a large portion of apps exhibiting malicious behavior, such as exploiting the Master Key vulnerability (discussed in Section 3.3.1), would have been missed. Malware apps make up 41.15% of our dataset.

All. In addition to goodware and malware our complete dataset contains 30.95% other apps that are detected by 1 to 5 AVs or that are classified as adware, i.e., apps that are grayware.

Estimation of release date. In order to perform any kind of longitudinal analysis on our dataset and to categorize apps by the year of their release, we need to estimate the age of each sample. Besides this yearly division of our dataset we also would like to have a more precise estimate to allow for a fine-grained evaluation, such as the time it takes for us to receive and analyze samples after they have been released. We estimate the age of an app from four data points: (1) the last modification date of the APK file (`zip_modification_date`), (2) the release date of the SDK indicated by the minimum required API level (`sdk_release`), (3) the date a sample was first published in any of the markets monitored by AndRadar (`market_release`), and (4) the date a sample was first submitted to Andrubis (`first_seen`).

For (1), the last modification date of the APK file, we parse the timestamp for the archive member that was modified last, usually the app's certificate, from the ZIP central directory file header. Naturally, this date can be tampered with, as evidenced by 273 apps feigning a modification date in 1980, the first year the ZIP file format supports for timestamps, further 9,703 before the first Android version was released in 2008, as well as 86 apps dated in the future, up to the year 2107. For (2), we parse the minimum required API level from an app's manifest and map it against the Android version history.[4] For (3), we have information from AndRadar for 68,197 apps in our dataset, since not all markets specify an app's upload date and we do not want the overall release date of an app but the date when a specific version (based on the MD5 file hash) was released.

In general, we trust the modification dates extracted from the ZIP header as we only encountered relatively few outliers exhibiting unrealistic modification dates. However, we sanitize the `zip_modification_date` by checking the `sdk_release` as a lower bound for when the app could have been released in case the ZIP timestamp was predated, and the `market_release` as an upper bound when the app was first seen in the wild in case the app was postdated. In the normal case the app requests a specific API level after the corresponding SDK was released and the app is built before it is released to the public, e.g., an application market. In this case we estimate the release date (`apk_date`) as the date the ZIP was created:

[4] http://en.wikipedia.org/wiki/Android_version_history#Version_history_by_API_level

$$\begin{aligned} \texttt{sdk\_release} < \texttt{zip\_modified} < \texttt{market\_release} \\ \Longrightarrow \texttt{apk\_date} = \texttt{zip\_modified} \end{aligned} \tag{3.1}$$

For 10,000 apps (1.04%) the ZIP file was created before the corresponding SDK was released. This could be either due to the ZIP file header being tampered with or the app being part of an alpha/beta test of an unreleased SDK. Since an app cannot be installed on devices if it requires a higher API level than the currently available Android OS version, we assign the date of SDK release to the release date:

$$\begin{aligned} \texttt{zip\_modified} < \texttt{sdk\_release} \\ \Longrightarrow \texttt{apk\_date} = \texttt{sdk\_release} \end{aligned} \tag{3.2}$$

In only six cases the market release date indicates that the app was published before the requested SDK level was released. This could be due to an error on the developers' side, unintentionally requesting a higher API level than required. In this case we choose the maximum of the SDK release and the ZIP creation date:

$$\begin{aligned} \texttt{market\_release} < \texttt{sdk\_release} \\ \Longrightarrow \texttt{apk\_date} = \max(\texttt{sdk\_release}, \texttt{zip\_modified}) \end{aligned} \tag{3.3}$$

Around 5,000 apps (0.50%) were published in a market before the ZIP was last modified. Since this means that the ZIP header obviously has been tampered with, we set the release date to the market release as the first date we saw the app in the wild:

$$\begin{aligned} \texttt{market\_release} < \texttt{zip\_modified} \\ \Longrightarrow \texttt{apk\_date} = \texttt{market\_release} \end{aligned} \tag{3.4}$$

We now can use the apk_date to estimate the *analysis delay*: the time it takes between an app being released and the app being submitted to Andrubis. Figure 3.5 shows the CDF of the analysis delay for the first and second year of operation. Within the first year we only saw 15% of samples within one week of their release for both malware and goodware. In the second year this number significantly increased to over 40% for goodware apps, in part due to our crawling of the popular new apps from the Play Store on a daily basis. In the first year Andrubis analyzed 60%-70% of all samples within the first three months. This number increased for apps of all categories to 80% in the second year. Finally, the number of apps analyzed within six months of their release increased from 2012 to 2013 by 10 percentage points for apps of all categories, to close to 90% of goodware and 95% of malware samples.

Finally, for categorizing our dataset by release year, we also include the date Andrubis first saw an app in the estimation and assign $\min(\texttt{apk\_date}, \texttt{first\_seen})$ as the final app release date. This results in the dataset depicted in Figure 3.6, separated by app release year and category (*All*, *Malware*, *Goodware*). Android was released in September 2008, however, malware first surfaced in 2010 [102] and apps released before 2010 amount to less than 0.76% of all apps in our dataset, thus, we focus in our following evaluation on apps released between 2010 and 2014.

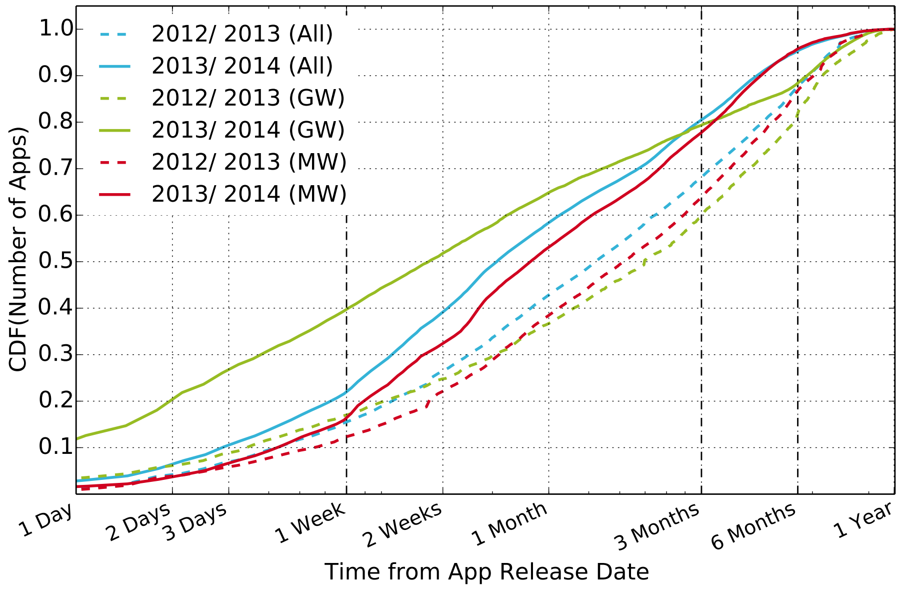


Figure 3.5: *CDF of the analysis delay for the whole dataset (All), malware (MW) and goodware (GW).* The analysis delay is the time between the release date of an app and its first submission to ANDRUBIS. This time decreased significantly between the first (June 2012 - June 2013, dashed line) and second year (June 2013 - June 2014, solid line) of operation. ANDRUBIS now sees 80% of samples within three months of their release.

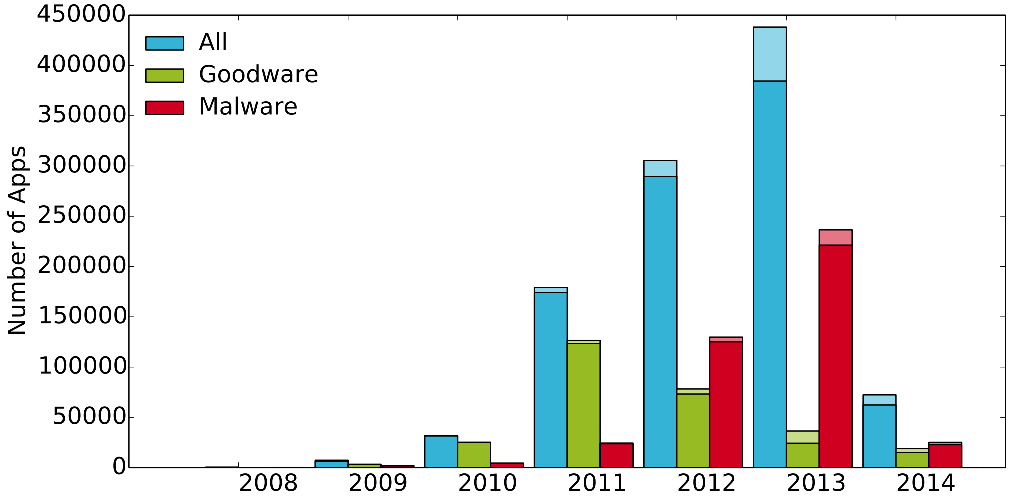


Figure 3.6: *Dataset overview.* Number of all, goodware and malware apps in our dataset categorized by the year of their release. The share of successfully analyzed apps is highlighted in a darker shade. Overall, our dataset contains 27.90% goodware, 41.15% malware, and 30.95% apps we could not reliable assign to either class (i.e., grayware).

3.3 Android Malware Landscape

We already used the dataset described in the previous section in prior work for exploring WebView-related vulnerabilities [144]. Based on apps collected between July 2012 and March 2013 we determined that 30% of apps were vulnerable to web-based attacks by exposing native Java objects via JavaScript.

In the following, we give a summary of apps' static analysis features and behavior during dynamic analysis and identify trends for *All*, *Goodware* and *Malware* samples over the past four years from 2010 to 2014. As our observations show, dynamic analysis is increasingly able to capture behavior otherwise missed by static analysis. This is in part due to the increasing use of dynamic code loading amongst malicious and benign apps and their use of obfuscation techniques and/or DRM protection. Additionally, while 57.08% of malware samples employ reflection with no significant change over the years, use of reflection amongst all apps has increased significantly from 43.87% in 2010 to 78.00% in 2014, and even more in goodware (from 39.55% to 93.00%). Therefore, it is essential for large-scale evaluations to include dynamic analysis systems, such as Andrubis, in order to obtain comprehensive analysis results.

3.3.1 Observations from Static Analysis

While dynamic analysis is gaining importance for getting a more complete picture about an app's functionality, for some features the evaluation of static features already provides valuable insights. In this section we take a look at permission requests and their usage according to static analysis, application names, developer certificates, resources sharing between apps, registered broadcast receivers, the use of third-party libraries, and the exploitation of the Master Key vulnerability.

Requested Permissions

Android apps can define and request arbitrary permissions: In fact, we observed almost 30,000 unique permissions being requested overall. Here, we focus on permissions defined and safeguarded by the Android OS. In addition to parsing all requested permissions from the manifest, we statically extract the usage of permissions from the app's source. While this approach ignores permissions that are requested, but only used in code dynamically loaded at runtime, we could use Andrubis' method tracer (Section 3.1.2), to determine the permission usage during dynamic analysis in future experiments. For now the dynamic extraction of used permissions is in an experimental state and results are only available for a subset of samples.

We statically extract the usage of 143 permissions, covering the most interesting and commonly requested permissions as shown in Table 3.4. In line with previous findings on permission usage amongst malware, malicious samples generally request more permissions than goodware, but use less of them: Malicious apps request 12.99 (11.57 when only looking at the subset of permissions we can statically extract) permissions on average, but use only 5.31 of them, goodware apps on the other hand request 5.85 (5.56)

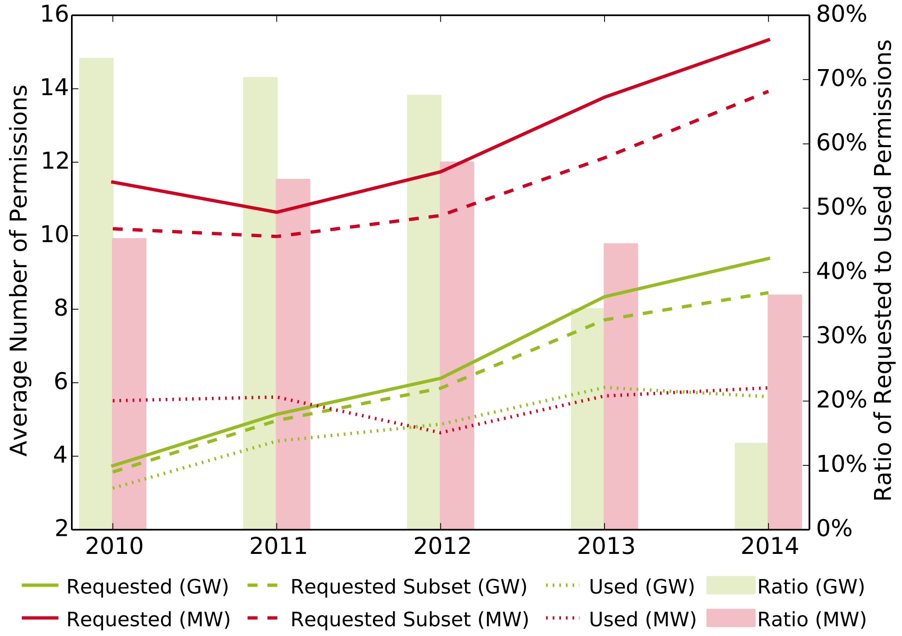


Figure 3.7: *Permission request and usage trends.* Goodware (GW) and malware (MW) apps request an increasing number of permissions (overall as well as from the subset of permissions we statically extract), but permission usage stays constant—a side effect of the increasing use of dynamic code loading and obfuscation. Consequently, the permission usage ratio (shown on the secondary y-axis) decreased significantly.

permissions on average and use 4.50 of them. One explanation for this behavior is that malware samples request more permissions during installation than needed so that they have the possibility to load other code parts that use these permissions later on. Permission requests by malware have also increased from an average of 11.46 (10.19) in 2010 to 15.33 (13.93) in 2014, with the average number of used permissions increasing only from 5.51 to 5.86. The number of requested permissions for goodware has increased from 3.74 (3.58) in 2010 to 9.38 (8.45) in 2014, while the number of used permissions also increased from 3.13 to 5.62. For individual samples, the permission usage ratio has declined for both goodware and malware, however, more significantly for goodware: Samples in this category from 2014 only use 13.38% of requested permissions in their code—a possible side effect of the increased use of dynamic code loading (see Section 3.3.2). Figure 3.7 illustrates this development.

Table 3.4 shows an overview of the most frequently requested permissions for malware and goodware. While the most commonly requested permissions for both malware and goodware are related to accessing the Internet, checking the network connectivity and reading device specific identifiers from the phone state, the majority of malware samples also requests SMS-related permissions. Furthermore, the possibility to manipulate shortcuts on the home screen can be used for phishing attacks and is frequently requested by malware as well. Another critical permission requested by malware is

Goodware		Malware	
83.97%	INTERNET	95.37%	INTERNET
61.54%	ACCESS_NETWORK_STATE	91.42%	READ_PHONE_STATE
43.65%	WRITE_EXTERNAL_STORAGE	82.79%	WRITE_EXTERNAL_STORAGE
38.09%	READ_PHONE_STATE	71.99%	ACCESS_NETWORK_STATE
23.59%	ACCESS_COARSE_LOCATION	69.91%	SEND_SMS
22.51%	VIBRATE	60.67%	RECEIVE_SMS
21.56%	ACCESS_FINE_LOCATION	55.66%	INSTALL_SHORTCUT
19.32%	WAKE_LOCK	51.40%	WAKE_LOCK
18.05%	ACCESS_WIFI_STATE	48.73%	READ_SMS
12.11%	READ_CONTACTS	45.62%	RECEIVE_BOOT_COMPLETED
11.83%	RECEIVE_BOOT_COMPLETED	40.15%	ACCESS_WIFI_STATE
8.30%	CALL_PHONE	32.92%	WRITE_SETTINGS
8.15%	CAMERA	30.05%	READ_CONTACTS
7.66%	GET_TASKS	25.74%	CALL_PHONE
7.45%	SEND_SMS	24.70%	ACCESS_COARSE_LOCATION
6.72%	GET_ACCOUNTS	24.30%	ACCESS_FINE_LOCATION
6.31%	WRITE_SETTINGS	23.83%	VIBRATE
6.11%	WRITE_CONTACTS	23.04%	GET_TASKS
5.02%	SET_WALLPAPER	20.15%	WRITE_SMS
4.96%	CHANGE_WIFI_STATE	20.12%	CHANGE_WIFI_STATE
4.57%	INSTALL_SHORTCUT	19.21%	SYSTEM_ALERT_WINDOW
4.47%	RECEIVE_SMS	19.11%	CHANGE_NETWORK_STATE
4.03%	RECORD_AUDIO	17.81%	GET_ACCOUNTS
4.00%	READ_CALENDAR	13.93%	INSTALL_PACKAGES
3.79%	READ_LOGS	13.23%	UNINSTALL_SHORTCUT

Table 3.4: *Requested permissions statistics.* Most frequently requested permissions in goodware and malware by the percentage of apps in each set.

SYSTEM_ALERT_WINDOW, which allows an app to show windows on top of all other apps, overlapping them completely. It is used to display aggressive ads and by ransomware that draws a window over all other apps to keep the user from accessing any other phone functionality.

It is also important to note that not only individual but also combinations of permissions can be security-critical: While the INSTALL_SHORTCUT permission, which is requested by more than half of the malicious apps, is classified as dangerous, the same functionality can be achieved through the combination of the normal READ_SETTINGS and WRITE_SETTINGS permissions [219]. This is common practice amongst malware, with 10.88% of malware samples requesting both permissions, while only 0.20% of goodware samples do so.

During our evaluation of dynamic analysis features (discussed in Section 3.3.2), we observed samples attempting to send SMS, connect to the Internet or accessing the SD card, without having the appropriate permissions—actions that will be prohibited by the Android OS. One explanation, besides a simple oversight, is developers mistyping the intended permission in some cases, for example as andorid.permission.* or android.permmission.*.

Application Names

The package name is the official identifier of an app, i.e., no two apps on a given device can share the same package name. Some markets, such as Google Play, also use it as a unique reference, but developers are not restricted from creating an app with an already existing package name. For malware authors reusing the package name of a legitimate app is also a way to masquerade as a benign app. Consequently, malware samples are far more likely to reuse package names than goodware samples: While 73.78% of goodware package names are unique, the same holds true for only 25.72% of malware's package names. Note, this number is likely to be slightly biased by submissions from AndRadar that explicitly locates apps in markets based on their package name to model and analyze how they spread [125].

Overall 8.50% of malware samples share their package name with legitimate apps from our goodware set—4,059 distinct package names in total, half of which are currently available in the Google Play Store. Among the most frequently repackaged apps are Armor for Android Antivirus (`com.armorforandroid.security`, 387 samples), Steamy Window (`com.appspot.swisscodemonkeys.steam`, 93 samples), Opera (`com.opera.mini.android`, 68 samples), Bluetooth File Transfer (`it.medieval.-blueftp`, 28 files), and Flappy Bird (`com.dotgears.flappybird`, 23 samples). Besides the paid Armor Antivirus all apps exceed 5 million downloads on the Google Play Store.

By far the most often shared package name, shared by 1,735 malicious apps with a single legitimate Google Play app, is `com.app.android`, however, more likely due to careless naming on the legitimate app's developers side. In general, authors of malicious apps tend to favor generic names and reuse them between samples, `com.software.app` and `com.software.application` being the most popular ones with 9,256 and 8,321 unique samples respectively. Starting in 2012, we observed malware authors adopting random looking package names, such as `ouepxayhr.efutel`, `ovbknnfm.xwscmnoi` and `rpyhwytfysl.uikbvktgwp`. F-Secure observed those package names being particularly popular amongst the *Android.Fakeinst* family [76]. However, contrary to the first impression, package names are not randomized on a per-app basis, as evidenced by up to 3,234 unique samples per name.

Certificates

Certificates are a corner-stone in Android security: Each and every Android app has to be shipped with its developer's certificate and signed with her private key so that it can be installed. Android uses the certificate to enforce update integrity, i.e., it only allows updates signed with the same key, and it uses it to allow resource sharing and permission inheritance between apps from the same author [30].

Google does not impose any restrictions on the certificates used to sign Android apps and over 99% of all certificates are self-signed. Google Play's release guidelines merely recommend that apps in official market should be signed with a key valid until at least October 2033. While this policy is not enforced, far more goodware than malware apps comply with it: Only 1.05% of certificates used to sign goodware apps ignore this policy, but 16.93% of certificates used to sign malicious apps do.

We collected increasingly more apps signed by the same key, for goodware and malware alike. While, in 2010, 19.21% of all keys were used to sign more than one goodware app and 28.57% of the keys were used to sign more than one malicious one, this increased to 40% for both goodware and malware in 2014. This not only means that we are collecting more apps by the same developers, but also that blacklisting certificates used to sign malware is a viable option to keep malware from spreading. Especially widely used are four test keys distributed as part of the Android Open Source Project (AOSP): 8.92% of malicious samples are signed with one of these test key, however, the ratio significantly decreased from 65.29% of malicious apps in 2010 to 7.29% in 2014. Although those keys should not be used by legitimate apps, 2.26% of goodware apps are signed with a publicly available test key—making them vulnerable to attack: If a user has such an app installed, malware signed with the same test key can potentially share permissions with the vulnerable app, as we will show in the next section.

To our surprise we also found four samples, each labeled by more than 11 AV scanners as part of the *Android.Bgserv* malware family, that are signed with a valid Google certificate. These apps with the package name `com.android.vending.sectool.v1` are a malware removal tool by Google, mistakenly flagged as malware by numerous AV vendors [198].

Application Interdependencies

The Android system assigns, by default, a unique user ID (UID) to each app and runs it as that user in a separate process. Apps, however, can share their UID with other apps by specifying a `sharedUserId` in the manifest. This allows apps to share data, run in the same process, and even inherit each other's permissions [30], all under the prerequisite that apps are signed with the same key. Clearly, this feature also allows collusion amongst apps [134]: A malicious payload could be spread across multiple innocent looking apps. We saw this feature more commonly implemented in goodware than in malware: 1.14% of apps share their UID while only 0.29% of malicious apps do. This functionality becomes especially security critical when combined with an exploit for the powerful Master Key vulnerability (detailed in Section 3.3.1). In theory, attackers could inject their code into apps not requesting any permissions at all but inheriting permissions from more privileged apps through a shared UID. Apps can even try to gain system privileges by exploiting an app signed with a platform certificate and sharing the UID with `android.uid.system`. Furthermore, with numerous apps being signed with the test key from the AOSP, crafting a malicious app inheriting the permissions from other apps is possible even without having to utilize an exploit to circumvent the app signing process. In fact, 6.79% of benign and 17.57% of malicious samples that share a UID are signed with a public test key. This becomes especially critical when the Android OS itself is signed with a public key: According to DroidRay [224], a recent security evaluation of custom Android firmware, out of 250 firmware images, 56.80% were signed with a key pair from the AOSP. In our dataset, we identified 84 apps (4 of which were not

Goodware		Malware	
11.29%	BOOT_COMPLETED	56.32%	BOOT_COMPLETED
8.93%	APPWIDGET_UPDATE	41.73%	SMS_RECEIVED
8.74%	INSTALL_REFERRER	14.56%	CONNECTIVITY_CHANGE
6.74%	SCREEN_OFF	13.49%	DATA_SMS_RECEIVED
6.69%	USER_PRESENT	11.95%	AIRPLANE_MODE
4.17%	CONNECTIVITY_CHANGE	10.18%	PACKAGE_ADDED
2.40%	PACKAGE_ADDED	4.24%	NEW_OUTGOING_CALL
2.38%	IN_APP_NOTIFY	2.66%	USER_PRESENT
2.25%	SMS_RECEIVED	2.14%	BATTERY_CHANGED
1.43%	PHONE_STATE	1.72%	DEVICE_ADMIN_ENABLED
0.91%	MEDIA_BUTTON	1.60%	INSTALL_REFERRER
0.78%	PACKAGE_REMOVED	1.50%	APPWIDGET_UPDATE
0.70%	SERVICE_STATE	1.43%	PHONE_STATE
0.65%	SCREEN_ON	1.40%	BATTERY_CHANGED_ACTION
0.64%	MEDIA_MOUNTED	1.03%	PACKAGE_REMOVED
0.61%	NEW_OUTGOING_CALL	0.90%	UNINSTALL_SHORTCUT
0.60%	BATTERY_CHANGED	0.90%	INSTALL_SHORTCUT
0.47%	PACKAGE_REPLACED	0.90%	SIG_STR
0.40%	DEVICE_ADMIN_ENABLED	0.88%	ACTION_POWER_CONNECTED
0.36%	DEVICE_STORAGE_LOW	0.70%	SCREEN_OFF
0.32%	STATE_CHANGE	0.61%	PICK_WIFI_WORK
0.27%	TIME_SET	0.51%	TIME_SET
0.25%	WAP_PUSH_RECEIVED	0.41%	WAP_PUSH_RECEIVED
0.25%	ACTION_POWER_CONNECTED	0.39%	SCREEN_ON
0.24%	MEDIA_UNMOUNTED	0.36%	BATTERY_LOW

Table 3.5: *Broadcast receiver statistics.* Most frequently registered broadcast receivers in goodware and malware by the percentage of apps in each set.

detected by any AV scanners, the remainder was labeled as *Android.Fjcon*) that were capable of gaining system privileges through UID sharing with `android.uid.system`. All samples were signed with the same AOSP key pair used to generate the system signature for 134 (53.60%) of the firmware images evaluated by DroidRay.

Broadcast Receivers

Apps can register broadcast receivers for arbitrary custom events, however, we focus our analysis on broadcast receivers listening for system events. Broadcast receivers are by far more widely used in malicious apps than in benign apps: 82.18% of all malware samples register one or more broadcast receivers, while only 41.86% of goodware sample use this feature. Table 3.5 lists the most frequently registered broadcast receivers for both categories. Goodware mainly watches for notifications to update their widgets, install referrers from the market and a user being present, probably to suspend idle mode quickly whenever a user unlocks the phone so that new data can be fetched and the app's status can be updated. Malware, on the other hand, often registers itself as a service, which is running in the background, and does not care for user input. More than half of

Goodware		Malware	
36.76%	AdMob (Google)	5.74%	AdMob (Google)
5.61%	Flurry	3.90%	WAPS
4.00%	Millenial Media	2.94%	Kuogo
2.92%	MobClix	2.92%	domob
2.72%	AdWhirl	2.67%	Adwo
1.94%	InMobi	2.02%	AirPush
1.77%	MobFox	1.97%	YouMi
0.91%	MoPub	1.43%	Vpon
0.78%	Adlantis	1.27%	Wooboo
0.74%	Admarvel	1.15%	MobWIN
0.67%	Smaato	0.91%	Millenial Media
0.63%	YouMi	0.84%	Flurry

Table 3.6: *Ad library statistics.* Most popular advertisement libraries in goodware and malware by the percentage of apps in each set.

all samples listen for the BOOT_COMPLETED event, which triggers as soon as the phone has booted the Android OS, and for the event that is published upon receipt of incoming messages, both text-based (SMS_RECEIVED) and data-based (DATA_SMS_RECEIVED). However, we only saw listeners for data-based SMS in 2012 and 2013 with 24.43% and 10.41% of malware samples listening for this event. We also see growing interest of malicious apps in the CONNECTIVITY_CHANGE and AIRPLANE_MODE receivers since 2012 with a peak of 17.93% and 14.99% in 2013 respectively. Furthermore, malicious apps started using Device Administrator Privileges, which makes them harder to uninstall. The latter are used by 11.94% of malware samples in 2014, which register for the DEVICE_ADMIN_ENABLED event.

Third-Party Libraries

We checked all apps in our dataset against a list of the 53 most popular *advertisement (ad) libraries* according to AppBrain.[5] Fewer malicious (17.45%) than benign (44.32%) apps come bundled with ad libraries, presumably in part because we excluded samples labeled as adware from our malware dataset. However, with ad fraud being one way to monetize malicious app installs, malicious samples include more ad libraries simultaneously: We saw a maximum of 13 ad libraries in a single goodware app and 14 ad libraries in a single malware app with 1.56 and 2.05 libraries on average respectively. Table 3.6 lists the most popular ad libraries for goodware and malware. Besides Google's AdMob being the most popular across both categories, albeit with diverging percentages of over 35% in goodware to only 5.7% in malware, there is little overlap. With mobile malware being particular prevalent in China [130], malicious apps mainly include Chinese ad networks. Malware also favors aggressive ad libraries, such as AirPush and Adwo, often classified by AV scanners as adware and banned from Google's Play Store [174] by policy because they push advertisements to the notification bar.

[5] http://www.appbrain.com/stats/libraries/ad (retrieved July 1, 2014)

Social networking libraries are used in 11.14% of goodware apps (8.86% Facebook, 3.38% Twitter, 1.89% Google+), while the number of malicious apps including such libraries is a negligible 0.78% (0.66% Facebook, 0.13% Twitter, 0.09% Google+), possibly indicating those libraries are shipped with the original app that was targeted by repackaging to include malicious code.

The same as for social networking libraries holds true for the use of *billing libraries*: 3.58% of goodware and only 0.53% of malware apps make use of billing services (3.08% Google Billing, 0.57% Paypal, 0.17% Amazon Purchasing and 0.03% Authorize.net in goodware; 0.35% Google Billing, 0.19% Paypal and 0.05% Amazon Purchasing in malware). Billing services for in-app purchases are harder to monetize for malware since payment providers usually have refund policies.

Master Key Vulnerability

In 2013 researchers reported the Master Key vulnerability [82] in the Android app signing process, which allows an app's content, including its code, to be modified without breaking the signature—essentially allowing attackers to inject malicious code into any legitimate apps without repackaging them. This vulnerability stems from discrepancies between the handling of the ZIP file format between the signature verification and installation process in Android. Shortly after the original Master Key vulnerability was published, two similar vulnerabilities were discovered [64, 65].

Bug 8219321, the original Master Key vulnerability, is based on the fact that the ZIP file format allows two files with the same file name, thus allowing attackers to hide an additional `classes.dex` file that is deployed by the installer instead of the original one that is checked by the signature verifier. We saw this vulnerability being exploited in 1,152 samples (0.11%), all from 2013 and 2014, and only in malware, possibly due to AV scanners automatically flagging apps as there is no legitimate reason for this behavior.

Bug 9695860 stems from a signed unsigned integer mismatch in the length of the extra field of the ZIP file header. In addition to allowing attackers to inject an app with a malicious `classes.dex`, the exploitation of this vulnerability also breaks analysis tools utilizing the unpatched version of the Python `zipfile`,[6] such as Androguard in the default configuration. We saw 4,553 samples (0.44%) triggering the Python bug. However, we only found two samples with an extra field length triggering an integer overflow and thus the vulnerability, one of them being a proof of concept.[7]

Bug 9950697 lies within the redundant storage of the length of the file name in both the central directory of the ZIP file as well as the local file header. Again this vulnerability allows attackers to specify a file name large enough for the installer to skip the original `classes.dex` file and install the injected one. However, we only observed this bug being exploited in 447 (0.05%) of all samples (starting already in 2011 with the majority of samples being from 2013), with 92 malware and 26 goodware samples respectively.

[6] Python Bug Tracker Issue 14315: `http://bugs.python.org/issue14315`

[7] Exploit for Android zip file bugs 8219321, 9695860, and 9950697: `https://github.com/Fuzion24/AndroidZipArbitrage`

3.3.2 Observations from Dynamic Analysis

In contrast to static analysis, dynamic analysis lets us monitor an app's behavior during runtime—including behavior caused by dynamically loaded code. In addition, the obtained information is more comprehensive and includes full paths of file system accesses, called phone numbers, recipients and contents of SMS, leaks of sensitive information, as well as usage of cryptographic algorithms and a full profile of the app's network behavior.

File Activity

Apps can both read and write the internal storage as well as external storage from SD cards. Overall 72.49% of goodware and 95.99% of malware read files, and 83.11% of goodware and 94.70% of malware write to the file system during dynamic analysis in Andrubis. When distinguishing file system access to the primary storage and access to the secondary storage, i.e., the SD card, it becomes apparent that SD card access is far more prevalent amongst malware: 22.02% of malicious apps read and 27.82% write files to the SD card, while only 2.91% of benign apps read and 6.69% write to external storage. Starting in Android 3.2 (Honeycomb) Google restricted third-party apps from accessing the SD card by limiting the `WRITE_EXTERNAL_STORAGE` to the primary storage and requiring the `WRITE_MEDIA_STORAGE`, which is only granted to system apps, for write access to the SD card. However, this change was largely ignored by OEM and custom firmware developers [195]. In our dataset 93.08% of goodware and 97.69% of malware apps that write to the SD card request the first permission, while only 0.59% of goodware and 0.08% request both. Static analysis completely failed to determine the usage of the `WRITE_EXTERNAL_STORAGE` permission and thus the write access to the SD card in any app. Furthermore, despite Google's policy to restrict write access to SD cards, this behavior has been steadily increasing in goodware apps from 2.89% in 2010 to 16.64% in 2014. Writing to SD storage has been a constant behavior in around 30% of all malware samples. This is likely to increase even more in the future with new possibilities for monetization being explored: Recently the Cryptolocker family started encrypting files stored on the SD card and demanding a ransom for the decryption key [126].

Phone Activity

Concerning mobile-specific behavior, only very few apps initiated phone calls during dynamic analysis: 0.24% of goodware apps and only 0.04% of malware apps. For both malware and goodware, 98% of those apps requested the corresponding `CALL_PHONE` permission, however, static analysis failed to determine any usage of this permission from the apps' source.

While the percentage of apps sending SMS in the goodware dataset is as low as the percentage of apps initiating phone calls (only 0.26%), we observed 15.00% of malicious apps sending text messages. This comes as no surprise: Sending SMS to premium numbers is a popular monetization vector of mobile malware [192]. Again, 98.57% (goodware) and 99.15% (malware) of those apps requested the necessary `SEND_SMS` permission,

Goodware		Malware	
12.86%	IMEI	39.68%	IMEI
1.70%	IMSI	25.88%	IMSI
1.51%	PHONE_NUMBER	13.89%	PHONE_NUMBER
1.12%	LOCATION	4.34%	ICCID
1.12%	LOCATION_GPS	1.40%	CONTACTS
0.60%	ICCID	0.40%	PACKAGE
0.08%	PACKAGE	0.11%	SMS
0.06%	CONTACTS	0.11%	CALL_LOG
0.05%	SMS	0.10%	LOCATION
0.02%	CALL_LOG	0.10%	LOCATION_GPS
0.01%	BROWSER	0.07%	BROWSER
0.01%	CALENDAR	0.00%	TAINT_CAMERA

Table 3.7: *Data leak statistics.* Information most commonly leaked to the network by goodware and malware by the percentage of apps in each set.

while static analysis revealed that 85.37% (goodware) and 81.79% (malware) of those apps actually use this permission in their source code—again showing the value and importance of dynamic analysis to uncover behavior from hidden or obfuscated function calls. Phone numbers tend to be shorter for malware, also indicating the use of premium numbers—goodware apps send SMS to 410 unique numbers with an average length of 7.18 digits, while the 1,943 distinct numbers malware sends SMS to is only 4.26 digits on average. Furthermore, we observed malware samples sending up to 120 SMS to premium numbers during just four minutes of dynamic analysis.

Data Leakage

Data leakage is significantly more prevalent in malware than in goodware: Overall, 14.28% of goodware apps leak information over the network, while 42.53% of malicious apps do so. When looking at the dataset as a whole, data leakage to the network overall occurred in 38.79% of all apps and significantly increased from 13.45% in 2010 to 49.78% in 2014. Both goodware and malware leak device specific identifiers, such as the International Mobile Station Equipment Identity (IMEI), International Mobile Subscriber Identity (IMSI), Integrated Circuit Card Identifier (ICCID) and the phone number. Goodware mainly leaks the IMEI, while a quarter of malware leaks the IMSI and almost 14% of malware leaks the user's phone number. Leakage of names and phone numbers from the user's address book is also more common amongst malware than it is amongst benign apps. Instead, goodware mainly leaks the location, an information source less commonly leaked by malware samples. Few samples in general leak information on installed packages, the contents of SMS, the call log, and browser bookmarks. Table 3.7 summarizes the information sources most commonly leaked to the network.

Data leakage via SMS occurred only in 0.04% of goodware and in 0.72% of malware samples. This number, however, has increased over the past years, with 1.87% of malware samples leaking identifiers such as the IMSI, IMEI, ICCID, and the phone number, but also forwarding incoming SMS and the call log via SMS in 2014.

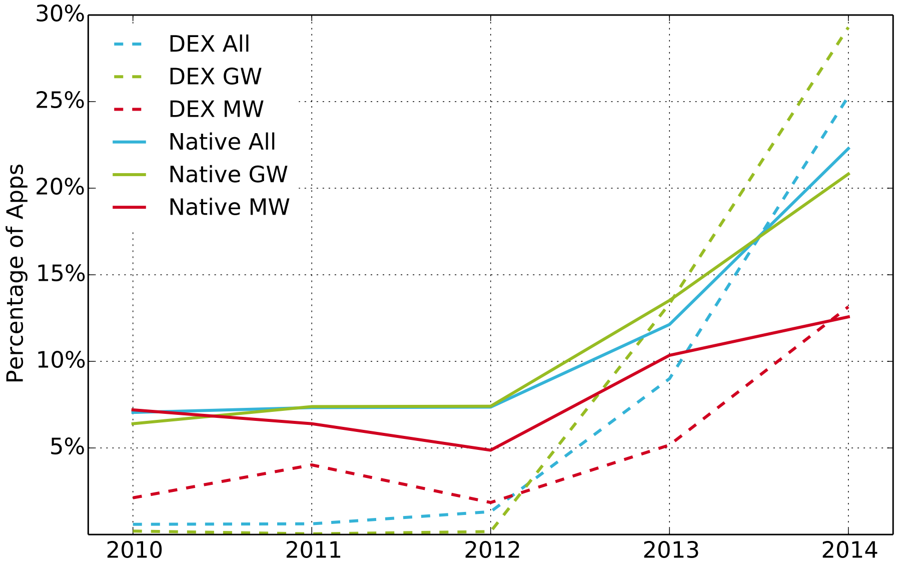


Figure 3.8: *Dynamic code loading trends.* And increasing number of apps, both goodware (GW) and malware (MW) dynamically load DEX classes (dashed line) and native libraries (solid line) at runtime.

Dynamically Loaded Code

Android apps can load code at runtime to dynamically extend their functionality. However, this technique comes with severe security implications. While dynamic code loading is popular for legitimate reasons, such as loading external add-on code, shared library code from frameworks, or dynamically updating code during beta and/or A/B testing, it is especially interesting for malware. Since apps are typically inspected only once, either by an app market or by an AV scanner at installation time, malicious apps can download and load their malicious payload later at runtime to evade detection. Furthermore, the unsafe use of code loading techniques can also make legitimate apps vulnerable to code injection techniques, as shown by Poeplau et al. [159].

DEX classes. One possibility to dynamically extend an app's functionality is to load modules at the Dalvik VM level through the DEX class loader. We observed this behavior for 2.97% of goodware and for 4.46% of malware apps, with a significant increase over the past two years. Static analysis successfully identifies the invocation of the `DexClassLoader` in 98.88% of goodware and 97.20% of malware respectively. On average, goodware loads 1.28 and malware loads 1.59 DEX classes. The maximum of different classes loaded is 37 for the Metasploit payload, 25 classes for samples from the *Android.SmsSpy* family, and 9 classes for goodware in general.

Native libraries. Overall, both goodware and malware apps load native libraries in equal proportions: We observed 8.60% and 8.50% of all benign and malicious apps loading native code during dynamic analysis, with a clear upward trend especially amongst goodware. The sources for the loaded native code and their impact differ: At a finer granularity, we distinguish between the number of system native libraries loaded and custom, non-system, libraries loaded. Custom libraries are far more dangerous than those provided by the Android system itself. The reason for system library usage is simple: Games and graphically demanding apps make use of hardware-accelerated technologies found in modern graphics cards, like OpenGL or video decoding, for both performance reasons and increased battery life. Custom libraries, however, tend to be used by malware for a number of nefarious purposes, including the elevation of privileges through root exploits.

Goodware apps load 52.47% and 52.26% code from the system and the data directory respectively, contrary, only 19.46% of malware samples load native system libraries, while 84.19% load their own bundled native code or fetch it from remote servers. While for malware the percentage of system libraries loaded decreased from 2010 to 2014 by 13 percentage points and the usage of custom libraries increased by 20 percentage points, this trend is more severe for goodware: In 2010, 74.11% of goodware apps loaded native code from the system and only 29.57% loaded custom code; in 2014, 30.95% of apps loaded code from the system and 73.37% loaded it from the data directory.

Static analysis was far less successful in identifying native code loading compared to DEX class loading and only identified the `loadLibary()` call in 54.40% of goodware and 83.25% of malware. These numbers correspond to the number of apps shipping with unencrypted ELF libraries that can be identified based on their file signature alone: 54.29% in the case of goodware and 85.23% in the case of malware.

Most importantly, dynamic code loading significantly increased during our observation period, especially for goodware over the past two years, as shown in Figure 3.8: In 2014, 29.29% of benign apps loaded DEX classes and 20.82% loaded native code, while 13.15% of malicious apps loaded DEX classes and 12.57% loaded native code. Furthermore, loading native libraries and DEX classes is not an either-or decision: 1.25% of all malware (5.43% in just 2014) and 0.45% of all goodware (4.92% in 2014) combine those techniques to load both native and Dalvik code.

Cryptographic API Usage

Another interesting case study is the use of cryptographic protocols. During dynamic analysis, we observed the usage of the Java crypto API in 5.63% of malicious apps, in contrast to only 1.10% of goodware apps. Interestingly, for those apps we could statically determine the use of cryptography in 99.21% of cases for goodware, but for only 43.24% of malware—either due to this part of the code being obfuscated and loaded dynamically at runtime. Overall, static analysis revealed the use of `javax/crypto/*` in 44.83% of goodware apps, increasing from 11.12% in 2010 to 79.18% in 2014. For malware, we did not see such a development with the Java crypto API being used by only 29.84% overall, likely due to malware shipping their own implementations in order to evade detection.

The most popular algorithms observed during dynamic analysis of goodware are AES (66.75%), PBEwithMD5andDES (15.03%), DES (11.98%), and RSA (5.08%). Malware, on the other hand, mainly used AES (74.82%), Blowfish (14.31%), DES (8.78%), and

RSA (1.20%). We also observed a trend toward stronger cryptographic algorithms in malware: While DES was the predominantly used algorithm amongst malware in 2010 (98.44%), its usage declined significantly to 1.53% in 2013. Instead, in 2012 malware authors started adopting the stronger Blowfish algorithm, which is now being used by 31.58% of all malware apps from 2013, while we have not seen a single goodware app using Blowfish.

Network Activity

We observed network traffic in goodware and malware apps alike—71.11% of goodware and 80.36% of malware, with almost 99% of those samples requesting but only 70.97% of benign and 61.43% of malicious samples using the `INTERNET` permission according to static analysis. This numbers decreased for malware in 2014 to only 94.40% requesting and 58.84% using the permission, indicating malware circumventing the permission system by performing network activity through other apps installed on the device, such as the browser, for example.

Almost all apps that access the network query domain names: 99.91% of malware and 97.34% of goodware perform DNS queries, but while one third (32.33%) of the queries by malicious samples fail and result in a non-existent domain (`NXDOMAIN`), only 10% of queries from goodware samples do. Looking at these results on a per-sample basis, 74.36% of malware samples perform at least one DNS query for an invalid domain, while this is only the case for 25.65% of goodware apps. Judging from experience with Windows malware, these results merit further investigation in future work, as frequent domain resolution failures can be indicative of malware using domain generation algorithms (DGA) to resolve C&C server addresses that are either no longer or not yet available [18].

UDP traffic is almost limited to DNS, with only a few samples using NTP. However, 55.33% of malware and 23.62% of goodware also establish TCP connections. This number increased for malware from 27.69% in 2010 to 58.65% in 2013, and decreased to 45.84% in 2014; for goodware it monotonically increased from 12.81% in 2010 to 43.50% in 2014. The most commonly observed network activity for malware occurred on port 443 (HTTPS, 44.09% of samples), port 80 (HTTP, 15.52%), and port 8245 (DynDNS), 5224 (XMPP/Google Talk instant messaging protocol), and 9001 (Tor) with less than 0.2% of samples each. For goodware we observed port 443 (HTTPS, 15.58%), port 80 (HTTP, 7.31%), and port 1130 (CASP[8], 0.46%).

Other protocols were hardly ever used: We only observed 77 apps in our whole dataset establishing FTP connections and 14 samples using IRC. We saw, however, 352 samples from 2013 and 2014 establishing SMTP connections and sending emails. The majority of those samples are classified by AV scanners as malware and they leak sensitive information such as the contents of the address book and incoming SMS via email to addresses from Chinese freemail providers, such as NetEase (`163.com`, `126.com`) and Tencent (`qq.com`).

[8] Manual investigation revealed that this network activity was actually caused by HTTP traffic with the Korean ad network Cauly (`http://downinfo.cauly.co.kr`).

Cross-Platform Malware

In 2013 Android malware started to download a malicious Windows payload (*Backdoor.MSIL.Ssucl*) and saving it together with an `autorun.inf` file in the root directory of the phone's SD card, hoping it would be automatically executed on Windows computers once the phone was connected to the PC via USB [48]. We only saw this behavior in 11 apps overall, nine of which were different versions of the goodware samples iSyncr and RealPlayer that placed their Windows installer together with `autorun.inf` on the SD card. Only 19 goodware samples embedded executables. The only malicious samples we saw exhibiting this behavior were from the *Android.UsbCleaver* [75] family. Overall, we detected 447 malware samples with a total of 27 different embedded executables that are flagged by at least one AV scanner.

There have also been reports of Windows malware attempting to infect Android devices, and even installing the Android Debug Bridge (ADB) to do so [127]. We have only seen 119 Windows samples in Anubis attempting to drop APK files, 16 of which also tried to access the ADB (currently not installed in our Windows environment). The majority of those files, however, failed to download completely or seem to belong to rooting utilities. VirusTotal has labels for 56 out of the 99 dropped APKs, with 33 not being detected by any AV scanners, 20 detected, as root exploits and the remaining three belonging to *Android.AndroRat* and *Android.FakeAngry.*

3.4 Limitations and Future Work

One limitation of any dynamic analysis approach is evasion. As long as a sandbox is not capable of perfectly emulating a system, a possibility to detect it exists. Petsas et al. [158] and Vidas et al. [204] recently explored the possibility to fingerprint Android sandboxes, and found that all, including ours, are susceptible to evasion. Sandbox detection techniques range from static characteristics of the specific Android OS installation to information from sensors, to the detection of the underlying virtualization technology. One proof of concept [137] is able to detect any QEMU-based environment based on binary translation: QEMU (and other emulators) usually take a basic block, translate it, and execute the whole resulting basic block on the host machine. Unfortunately, this property allows for an easy detection of emulated code, since the execution of a basic block cannot be interrupted by the guest operating system's scheduler. As a countermeasure, we enabled QEMU single-step mode, which makes Andrubis undetectable by this evasion technique. However, this mode introduces an analysis overhead of 29% compared to 7% with Dalvik monitoring and 18% with QEMU VMI [210]. Generally, dealing with analysis evasion is a never-ending arms race between security researchers and malware authors.

A further limitation of dynamic analysis is code coverage. While we try to increase behavior seen during analysis through various stimulation techniques, a more intelligent user interface stimulation than the random input stream by the Android Exerciser Monkey, such as CuriousDroid [43], could provide more complex and user-like input and, in turn, trigger much more behavior from the apps under analysis.

Currently public submissions to Andrubis are limited to a file size of 8MB. This limit, however, is simply a limitation of our web interface and not a fundamental limitation of our analysis. We are currently evaluating to increase this limit, while keeping storage requirements at an acceptable level without having to discard apps after analysis.

Finally, a limitation of any analysis system allowing submissions from anonymous sources is the lack of metadata and ground truth. We have no indication when and where samples were found or how widespread they are in the wild. We tried to mitigate this in part by collecting metadata from markets with AndRadar (described in Chapter 5). Lacking ground truth, we have to rely on AV signatures to classify our dataset in goodware and malware for this evaluation, but we are experimenting with machine learning approaches to automatically classify samples with higher accuracy than related work. We implemented our Android malware classification in an extension to Andrubis, called Marvin, which we describe in more detail in Chapter 4.

3.5 Conclusion

In this paper we presented Andrubis, a fully automated large-scale analysis system for Android apps that combines static analysis with dynamic analysis on both Dalvik VM and system level. Andrubis accepts public submissions through a web interface and a mobile app and is currently capable of analyzing around 3,500 new samples per day. With Andrubis, we provide malware analysts with the means to thoroughly analyze Android apps. Furthermore, we provide researchers with a solid platform to build post-processing methods upon based on an app's static features and dynamic behavior. For example, leveraging machine learning approaches one can use our analysis results to tackle the problem of judging whether a previously unseen app is malware significantly more accurate than prior work.

Andrubis has analyzed over 1,000,000 Android apps over the course of two years. On an evaluation of this dataset spanning samples from four years, we showed changes in the malware threat landscape and trends amongst goodware developers. Dynamic code loading, previously used as an indicator for malicious behavior, is especially gaining popularity amongst goodware, and, in turn, loses significant information value when distinguishing between benign and malicious apps. Due to this development, static analysis tools alone are increasingly unable to completely capture an app's behavior, making dynamic analysis indispensable for a comprehensive analysis for a large number of apps.

In future work, we plan to explore the network behavior of Android malware further to identify C&C communication patterns and shared infrastructures with Windows malware. Furthermore, we are exploring the option of releasing a comprehensive malware dataset, once we sorted out legal and confidentiality issues. To date we have been providing subsets of our dataset to researchers upon request.

CHAPTER 4

Marvin: App Classification Through Static & Dynamic Analysis

In contrast to Apple's iOS, which restricts users to the applications (apps) available in the iTunes App Store, Android users are not limited to the official Google Play Store. Instead, they can choose from arbitrary sources, such as alternative application markets, torrents, or direct downloads. Naturally, this liberty attracts the attention of malware authors, who try to lure users into running malicious code, e.g. by repackaging it with paid or very popular apps, such as Angry Birds [57] and more recently Flappy Bird [66]. Although there have been some drive-by download sightings for Android [164], user-based installation is still the most prevalent infection vector.

While recent studies, including our own (presented in Chapter 5), found alternative markets hosting up to 5–8% malicious apps [76, 125], the official Google Play Store is not free from malware either [23, 129]. The anonymity of the developer accounts, the low sign-up fee of US$25, and its popularity amongst users make the Google Play Store an attractive target, even when considering that developer accounts are banned when they are caught uploading malware. As a countermeasure against the growing mobile malware threat Google introduced *Bouncer* in February 2012 [128]. By testing apps for anomalous behavior in Bouncer's dynamic analysis environment before listing them in the Play Store, they claim to have reduced malicious app downloads by 40%. Not much is known about the exact functionality of Bouncer, although two independent studies showed that it can be fingerprinted and bypassed, just like any analysis environment as long as it is acting as an oracle [148,154]. In fact, malware authors have been successful in sneaking their malicious app's past Google Play's defenses time and time again [24,56, 100,161,188]. In November 2012, Google further extended Android's security features by integrating an application verification service in Android 4.2 that is capable of checking apps from any source for malicious functionality [184]. However, an assessment of the effectiveness of this service on a corpus of known malware showed a low detection rate of only 15.32% [109].

Ultimately, when deciding whether or not to install an app, the end user can consult various information sources:

- Trustworthiness of the app's origin
- App reviews by other users
- Permissions required by the app
- Results from antivirus (AV) scanners
- Results from Google's app verification service (Android 4.2 "Verify apps")

All of these sources have major shortcomings. To begin with, trustworthiness is hard to establish—as stated above—even the Play Store is not safe from malware, and the trust end users might put into it is likely making it an even more tempting target for malware authors. The problem with app reviews written by users is that most are quite unlikely to notice malicious behavior and their ratings mainly focus on functionality and performance instead of privacy risks [54]. Furthermore, malware authors can use app rank boosting services to increase download numbers and post fake reviews to encourage users to install their malicious apps [96]. The permissions required by an app might actually indicate what the app could do, but this information is too detailed and thus incomprehensible for a majority of users [80]. Finally, several AV companies offer solutions for mobile devices, however, they are restricted by the limited resources and privileges on mobile devices [53, 149]. Current devices are not designed to accommodate heavy-weight security solutions including behavior-based malware tracking. One reason is that most Android installations lack the needed root privileges to carry out the necessary operations [73]. Furthermore, a constant runtime scan of running apps puts enormous strain on a device's CPU and therefore reduces battery life drastically.

Consequently, researchers started to leverage machine learning techniques to classify apps based on features learned from known benign and malicious apps. We show in Section 6.3 that existing approaches have severe limitations in the their feature extraction as well as their ability to generalize in a real-world setting. We address these shortcomings by extending our large-scale public Android malware analysis sandbox Andrubis [123] (presented in Chapter 3) to provide users with a risk assessment in the form of a *malice score* that can be efficiently calculated and that is easy to grasp and understand. Marvin[1] follows the *hybrid analysis approach* and leverages static and dynamic analysis, both performed off-device (in the cloud), to represent properties and behavioral aspects of an app through a rich and comprehensive feature set. Using a classifier that is trained on a large set of known malicious (malware) and benign apps (goodware), Marvin estimates the risk associated with a previously unknown app. By providing a detailed analysis report in addition to the malice score of the classifier our system is both transparent in its assessment and beneficial to both novices and expert users alike. We further evaluate the long-term benefits and practicality of our approach by showing that it can be efficiently retrained and that it maintains its detection accuracy over time.

[1] Named after the paranoid android in The Hitchhiker's Guide to the Galaxy [14].

In summary, we make the following contributions:

- We introduce Marvin, a system to automatically evaluate the risk of unknown Android apps through a combination of static and dynamic analysis.
- To provide an appropriate end user experience, we developed a mobile app that allows users to submit apps to Marvin and receive malice scores along with a detailed analysis report.
- We evaluate Marvin on a dataset of over 135,000 Android apps, including 15,000 malicious apps, on which it correctly identifies 98.24% of malware samples with less than 0.04% false positives.
- We discuss the most distinguishing features, and features unique to Android apps, that Marvin uses to classify apps.
- We made our solution available to the public by integrating it in Andrubis, a large-scale app analysis sandbox that accepts submissions at `https://anubis.iseclab.org` and via our mobile app that is available in the Google Play Store.[2]

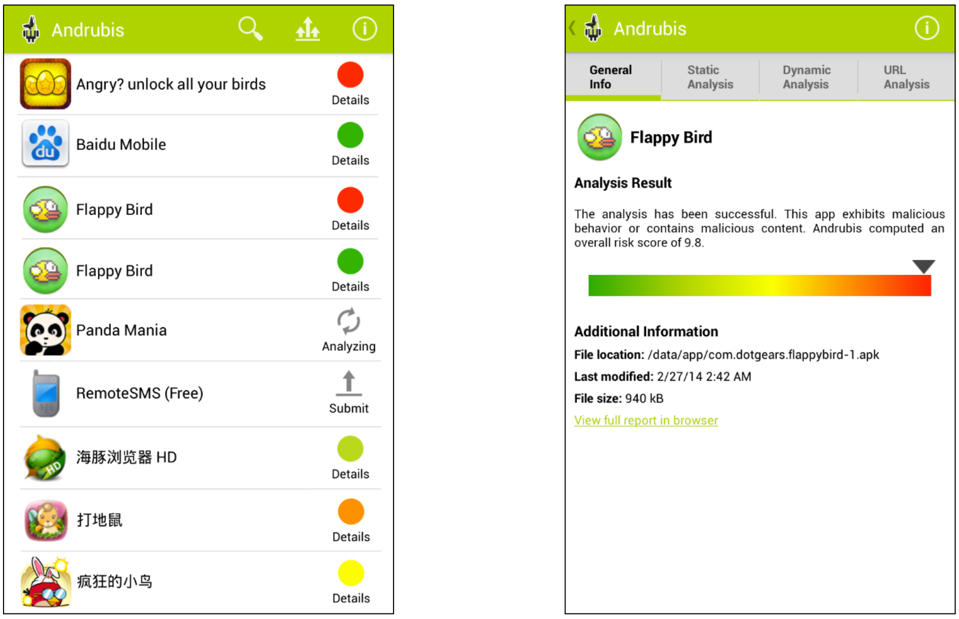
4.1 Approach

Marvin learns to distinguish malicious from benign apps based on a set of known malware and goodware. It assigns malice scores to unknown apps in a range from 0 (benign) to 10 (malicious). This addresses the fact that there is a gray area between malware and goodware, e.g. adware, and hence a binary risk assessment would be unsuitable. Furthermore, the scores allow categorizing apps into discrete levels, which makes the results easier to understand for end users [229].

The core element of our risk assessment engine utilizes machine learning techniques that classify apps based on several characteristics. These features are gathered from dynamic and static analysis, network-level behavior and meta information, like author fingerprints and application lifetime. As a result, Marvin computes the aforementioned malice scores based on a comprehensive set of distinguishing features that are capable of separating malware from benign apps accurately.

Users can submit apps through a web interface or a dedicated mobile app. The mobile interface is purposely kept simple to attract user attention and to not strain people's patience with overly detailed information that might only be useful to researchers or malware analysts (who can access additional information on a separate screen, if desired). This way, the mobile app is useful for novice and expert users alike. Figure 4.1 shows an overview of the user interface of the mobile app: The app's main screen (Figure 4.1a) lists all installed apps and allows users to submit apps that have not been already analyzed. The analysis view for each app (Figure 4.1b) shows in this example a Flappy Bird app repackaged with malware, which was correctly classified as malware with a

[2] `https://play.google.com/store/apps/details?id=org.iseclab.andrubis`



Analysis Result

The analysis has been successful. This app exhibits malicious behavior or contains malicious content. Andrubis computed an overall risk score of 9.8.

Additional Information

File location: /data/app/com.dotgears.flappybird-1.apk
Last modified: 2/27/14 2:42 AM
File size: 940 kB
View full report in browser

(a) Main view **(b)** Analysis view

Figure 4.1: *User interface of* Marvin*'s mobile app.* The app's main view lists all installed apps and allows users to submit apps. The analysis view shows the detailed analysis results for each app, in this case for a Flappy Bird app repackaged with malware, which was correctly classified as malware with a malice score of 9.8.

malice score of 9.8. Optionally, the user can download an exhaustive report if a specific app is of special interest, or view a summary of features in the *Static Analysis* and *Dynamic Analysis* tabs. Additionally, in the *URL Analysis* tab, the app also lists all URLs that were extracted statically from an app and that the app contacted during dynamic analysis, checks them against the Google Safe Browsing API [89], and issues a warning in case Google classifies an URL as suspicious.

In contrast to Google's Bouncer, which scans only apps submitted to the Play Store to approve or reject them, Marvin analyzes apps from arbitrary sources through a public interface and is independent of a device and its software version. As all operations are performed off-device and independently of a particular installation, Marvin can serve as a lightweight alternative to AV scanning apps or be integrated into other services. Marvin can, for example, be utilized by alternative app stores to let users make a more informed decision about which apps to download, or corporate app stores to decide which apps are fit to be distributed further. For the typical Android user, Marvin acts as an advisor to decide which apps can be safely installed on their device. For expert users and researchers, Marvin provides an efficient maliciousness rating tool for large-scale evaluations through its public interface. For example, Marvin provides a feed of apps that it flagged as malicious as an input for AndRadar (see Chapter 5), a tool that scans alternative app markets for malicious apps.

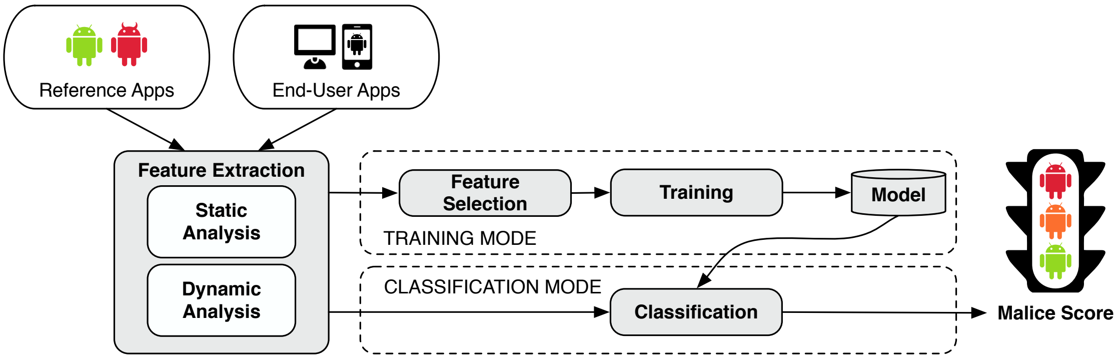


Figure 4.2: *System overview of* Marvin. In the *training mode* Marvin learns features from a set of benign and malicious reference apps. In the *classification mode* Marvin uses the resulting model to classify previously unseen apps and to compute a *malice score.*

4.2 System Description

Marvin operates in two different modes (illustrated in Figure 4.2): A *training mode*, where the existing model can be revised and adopted to new features or new strains of malware, and a *classification mode* in which it assesses the risk of an app and computes the malice score.

Training mode. The training mode is needed to learn the model that is later used to evaluate the risk associated with unknown apps. To this end, we provide a training set of known goodware and known malware to the system. Marvin first extracts a comprehensive feature set from these apps during static and dynamic analysis. Depending on the configured machine learning algorithm, the features then can be subjected to feature selection to avoid overfitting. Here, the most distinguishing features are determined, which are then used to train the machine learning algorithm and learn the corresponding classification model.

Classification mode. In classification mode, Marvin accepts user submissions via a web interface or our own mobile app. Each submitted app is subjected to the same feature extraction as the apps used to train the model. Based on the features exhibited by the app during static and dynamic analysis, and the model created during the training mode, the classifier then assesses the risk and outputs the app's malice score between 0 (benign) and 10 (malicious).

The feature extraction step can be skipped, if the submitted app has already been analyzed before. Otherwise, both static and dynamic analysis have to be performed first to gather the necessary features. Therefore, the timeframe to produce a result in this mode can vary from under a second to several minutes if dynamic analysis still has to be performed. However, with a growing repository of cached analysis reports and a daily throughput of over 3,500 new analysis reports, Marvin is able to instantly assess the risk of a large number of apps—currently amounting to over one million.

4.2.1 Feature Extraction

Feature extraction is an essential part of Marvin. Only a rich and comprehensive set of features that characterizes an app accurately will be sufficient for the classifier to produce meaningful results. Hence, we combine both static and dynamic analysis approaches. For Android apps, static analysis already provides a rich feature set with meta information about the app, such as its name, requested permissions or registered activities, as well as information about the author from the developer's certificate. Furthermore, static analysis can reveal information that we might not see during dynamic analysis, as the latter might suffer from weak code coverage. Thus, we can statically extract the presence of security-critical API calls and the actually used permissions. Dynamic analysis, on the other hand, shows aspects only observable during runtime, such as dynamically loaded code, even from external sources such as the web, packed, or otherwise obfuscated executables, which are unscrambled during execution. Additionally, capturing the app's network behavior during runtime renders dynamic analysis fundamental for detecting malware reliably.

As a first step in the feature extraction we extract information from an app's manifest, a mandatory XML-file that includes the necessary information for installing and running an app. The static analysis also extracts information from the developer certificate used for signing the app and the general structure of the app package. Furthermore, we extract the use of permissions and security-critical API calls from the app's code. Subsequently, we run each app in an emulated analysis environment that runs an instrumented Dalvik virtual machine to record the app's behavior. Marvin expresses the app's static properties and dynamic behavior as binary features in the format `(S|D)_Category-_Name`. For example, a dynamically observed HTTP connection is represented as `D-_HTTPGetHost_www.google.com`, while a statically extracted permission request in the manifest is expressed as `S_PermRequired_android.permission.send_sms`. The final outcome of the feature extraction phase is a sparse feature vector $\vec{x}$ of these binary features, with x_i denoting the i-th feature in the vector.

During our large-scale evaluation on a dataset of 124,189 unique Android apps (detailed in Section 4.3), we extracted 496,943 different features: 154,939 dynamic analysis features and 342,004 static analysis features. Table 4.1 lists the main feature categories and the number of distinct features that we observed for each category. We explain those features in the following paragraphs.

Static analysis features. An important source of information for static analysis features is the manifest that has to ship with every app. It contains essential information that the Android OS needs to install and run an app. Among the data contained in a manifest file, we extract the following:

- The Java package name that uniquely identifies the app in the Google Play Store and many alternative markets.
- The permissions requested by the app. Based on these permissions, the Android OS will grant certain security-critical actions to an app and deny others.

Type	Category	# of Features
static	Class Structure	132,609
static	App Names	93,375
dynamic	File Operations	85,204
static	Certificate Metadata	81,268
dynamic	Network Activity	55,808
dynamic	Manifest Metadata	30,807
static & dynamic	Intent Receivers	10,892
dynamic	Data Leaks	3,662
static & dynamic	Dynamic Code Loading	1,433
static	Used/Required Permissions	1,169
dynamic	Phone Activity	681
static & dynamic	Crypto Operations	35

Table 4.1: *Feature overview.* Number of features for each category and how they were extracted (through static analysis, dynamic analysis, or both).

- The intents the app will respond to by the means of a broadcast receiver. Apps can use this feature to be notified e.g. on system boot, the receipt of SMS or incoming calls.

- Publisher IDs for advertisement (ad) libraries. These are used by ad service providers to identify whom to pay the ad view revenue to.

We also add features that indicate whether the Android Application Package (APK) file and the manifest are valid, i.e. they are parseable by standard tools used for analysis, which is not always the case when examining malware samples. Furthermore, we parse the package structure and look for the presence suspicious files, such as native (shared) libraries, native executables and shell scripts embedded in the resources of the packaged app.

Additionally, we statically determine several aspects of the app's code in case they might not be triggered during the dynamic analysis phase. In detail, we extract the following information:

- The used permissions based on the app's API calls.

- The use of the reflection API.

- The use of the cryptographic API.

- The dynamic loading of code, both native code invoked via the Java Native Interface and Dalvik bytecode.

As a large number of apps are submitted to Marvin without any additional meta information that would help to identify the author of an app, we rely on the app developer's certificate for authorship information. The certificate used to sign an APK file can be issued by anyone and can be self-signed, but it must be the same for all apps of one author account in the Play Store. Thus, the certificate is also useful for attributing multiple apps to the same malware author. Previous work by Apvrille et al. [21] already suggested that information about the certificate's owner, its issuer and its validity can be an indicator for malware. Therefore, we extract the fingerprint, serial number, and owner of each certificate, whether it is self-signed and whether its validity period conforms to the release guidelines of the Play Store [90].

Dynamic analysis features. As research on Windows malware showed [142], static analysis techniques are prone to evasion by code obfuscation techniques. Furthermore, features should inherently represent the malicious behavior to be detected to prevent attackers from evading the learning method, e.g. with mimicry attacks [207]. Thus, any static analysis is ideally complemented with dynamic analysis that captures the harmful behavior inherent to malicious apps.

In order to obtain the dynamic analysis features, we extended the automated and publicly available dynamic analysis sandbox Andrubis that we proposed in previous work (see Chapter 3). Andrubis performs monitoring at the Java-level through a modified Dalvik VM as well as at the system-level through virtual machine introspection (VMI) in the emulator. Additionally, it employs various stimulation techniques to trigger program behavior and increase code coverage. We analyze each app for four minutes. This timeframe yielded the best trade-off between the use of our analysis resources and observed features in previous experiments. Furthermore, this timeframe ensures that the user receives her risk assessment in a reasonable amount of time, even for apps that have not been analyzed in the past. During analysis, we monitor the following events:

- *File operations.* Each file operation is represented as a combined feature of the type (read/write) and the file name.

- *Network operations.* Depending on the protocol level, network operations offer various types of information. Starting at the IP level, we represent destination host and port as distinct features. In the case of SMTP, FTP, DNS, HTTP, and IRC communication we also extract additional higher-level features. For FTP these are username and password of a conversation, for IRC they are username, nickname, password and channel, and for SMTP we extract the sender's address and the message subject. For DNS requests we extract the queried domain names as well as responses, as unsuccessful DNS resolutions (`NXDOMAIN`) can indicate malicious apps using domain generation algorithms [18]. For HTTP, we create a combined feature of the method and the request. For the request, we remove the request parameters as they showed to be extremely noisy in preliminary experiments.

- *Phone events.* Both outgoing phone calls as well as sent SMS are represented by the corresponding phone number.

- *Data leaks.* Observed data leaks are represented depending on the data sink used. For leaks via SMS we record the phone number, network leaks are expressed by host and port, and leaks to the file system by the file name.

- *Dynamically loaded code.* Code loaded at runtime is represented by the type of code (either native code or a DEX class) and the file name, respectively the class name.

- *Dynamically registered broadcast receivers.* Broadcast receivers are represented by the intent they are registered for.

To reduce the dimensionality of the feature vector and to avoid overfitting, we keep the features as generic as possible by replacing app or runtime specific identifiers such as process IDs, file descriptors and the package name of the app under analysis with tokens.

Ad libraries. A large percentage of free apps include one or more ad libraries that exhibit features that are difficult to separate from the behavior of the app itself and that might have an impact on our classification. One method to deal with ad libraries is to whitelist their features and exclude them from the classification process. However, this would mean maintaining and constantly updating a list of features for a number of different ad libraries. The second approach is to include the features of ad libraries in the classification process and assume that non-distinguishing features will be eliminated during feature selection. We chose this second approach as we assume that ad libraries will be included in both benign and malicious apps. One source of revenue for authors of malicious Android apps is the repackaging of legitimate apps to include their own publisher ID and thus receive the rewards instead of the original author. Furthermore, as the authors of AdRisk [92] and the recent discovery of a “vulnaggressive” ad library [220] showed, ad libraries themselves can be a privacy and security concern.

4.2.2 Choosing a Classifier

The classifier is a core component of Marvin, as its accuracy will immediately be reflected in the malice score. When choosing a classifier, we are bound to the requirements of our domain:

- *High-dimensional feature space.* The number of features in our evaluation dataset exceeds 490,000.

- *Sparse data.* The apps in our dataset only exhibit a small subset of the possible features.

- *Performance.* Both the training and classification process should take a limited amount of time, to enable short retraining intervals and to provide the end user with an as-instant-as possible risk assessment.

- *Scoring.* Since we want to address the gray area between malware and goodware, a classifier providing only a binary assessment, e.g., a decision tree, is unsuitable for our purposes.

Given these characteristics, we explore two machine learning approaches: a linear classifier and a support vector machine. For both approaches we leverage open source implementations, LIBLINEAR [77] for the linear classifier and LIBSVM [47] for the Support Vector Machine.

Linear classifier. Given a feature vector $\vec{x}$, a linear classifier computes the dot product with a weight vector $\vec{w}$:

$$y = \vec{x} \ \cdot \ \vec{w} = \sum_i x_i w_i \tag{4.1}$$

The outcome, y, is the margin of the classification. In essence, the weight vector $\vec{w}$ can be visualized as a hyperplane that splits the feature space into two sections, representing the classes the classifier can distinguish. While the sign of the margin specifies on which side of the hyperplane $\vec{x}$ is on, its absolute value $|y|$ can be interpreted as the confidence in this classification, with larger values corresponding to more confident predictions.

When operating on a binary feature space as in our case, linear classification speed directly scales with how sparse a given feature vector is, because the computational effort of computing the dot product $\vec{x} \cdot \vec{w}$ is proportional to the number of features expressed by a given sample.

Prior to actual classifying, the classifier's weights need to be determined in a training process. For training we experimented with both, L_1- and L_2-regularized logistic regression. L_1 regularization tries to minimize the sum of the absolute values of the feature weights to be small and tends to assign zero weight for a large majority of the features. In contrast, L_2 regularization optimizes the sum of the squares of the weights to be small and tends to assign non-zero weights to most features. As suggested by Andrew Ng [146] L_1 regularization proved superior to L_2 regularization when dealing with many irrelevant features, while logistic regression with L_2 regularization is extremely sensitive to the presence of irrelevant features. We show in our evaluation in Section 4.3 that both methods lead to similar results during classification, while the L_2 classifier performs noticeable better on previously unseen malware samples with outdated training data after several months. This suggests that the implicit feature selection observed with the L_1 classifier is only appropriate when using very short retraining intervals, which is why we chose the L_2 classifier in the current deployment of Marvin.

Support vector machine (SVM). In principle, an SVM works similar to a linear classifier, with a hyperplane splitting the feature space into two sections that maximizes the margin for both classes. However, it does address one problem of linear classifiers: As the name already suggests, the latter classifies samples only accurately if the problem is linearly separable.

To overcome this limitation, SVMs use the "kernel trick," implicitly mapping the input into a higher-dimensional space, where the problem is more easily separable.

The kernel of an SVM can be seen as the similarity measure. Given two feature vectors $\vec{x}$ and $\vec{z}$, for a pure linear classification their similarity would be their dot product:

$$K_{linear}(\vec{x}, \vec{z}) = \vec{x} \ \cdot \ \vec{z} \tag{4.2}$$

Since we want a non-linear classifier we use the standard Gaussian radial basis function (RBF) kernel instead:

$$K_{RBF}(\vec{x}, \vec{z}) = \exp\left(-\gamma || \vec{x} - \vec{z} ||^2\right) \tag{4.3}$$

For training, an SVM requires the cost factor C, which controls the trade-off between margin and erroneous classification. To determine C and the RBF-specific parameter γ, we perform 10-fold cross-validation.

As further detailed in our evaluation in Section 4.3, pure linear classification performed at least as accurate as the SVM and can be trained significantly faster. Thus, Marvin uses a purely linear classifier in our current deployment.

Prediction probabilities as scores. In the common case, the final result of a binary linear classifier is given by the sign of the margin: It either attributes the input to one or the other class. However, in the case of a malware/goodware distinction, a more precise differentiation is desirable. For instance, adware might express some features that are characteristic of malware, yet adware itself is not necessarily malicious. To address this problem, we do not want the classifier to merely predict classes, but also output the confidence or probability that an app belongs to a specific class.

To calculate this probability, we use the standard method for probability estimations, an exponential function of the margin y:

$$\frac{1}{1 + e^{-y}} \tag{4.4}$$

The higher the amount of malicious features an app under classification exhibits, the further away it is from the boundary of the separating hyperplane and the higher this probability will be. For display purposes we scale this probability to the interval $[0, 10]$ as the malice score for an app. Additionally, we can display the malicious features that contributed most to an app's malice score to let users make a more informed decision whether to trust an app or not.

4.2.3 Feature Selection

The features detailed in Section 4.2.1 include a large number of features that we extract from the static and dynamic behavior of Android apps. However, this does not necessarily mean that they are useful for classification. Some features are available throughout (almost) all apps in our dataset while other features are random or simply vary from one app to another. In order to reduce the dimensionality of our feature vector and to use only the most discriminative features, we perform feature selection.

The linear classifier performs *implicit feature selection* depending on the regularization algorithm and we trained both linear classifiers with L_1- and L_2-regularized logistic regression on the same feature set. For the SVM classifier, we performed an additional *explicit feature selection* step that is implemented in the LIBSVM tools.[3] It iteratively runs its parameter selection with $N, \lfloor N/2 \rfloor, \lfloor (N/2)/2 \rfloor, \ldots, 1$ features ranked by their

[3] `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/#feature_selection_tool`

respective F-score (Fisher score) and with N being the total number of features. Chen et al. [52] describe the F-score as follows:

$$F(i) \equiv \frac{(\bar{x}_i^{(+)} - \bar{x}_i)^2 + (\bar{x}_i^{(-)} - \bar{x}_i)^2}{\frac{1}{n_+ - 1}\sum_{k=1}^{n_+}(x_{k,i}^{(+)} - \bar{x}_i^{(+)})^2 + \frac{1}{n_- - 1}\sum_{k=1}^{n_-}(x_{k,i}^{(-)} - \bar{x}_i^{(-)})^2} \tag{4.5}$$

It is calculated on feature vectors $\vec{x}_k,\ k = 1\ \ldots\ m$, with n_+ and n_- being the numbers of positive/negative samples, $\bar{x}_i, \bar{x}_i^{(+)}, \bar{x}_i^{(-)}$ being the average of the i-th feature of the whole, positive and negative sets, and $\bar{x}_{k,i}^{(+)}$ and $\bar{x}_{k,i}^{(-)}$ are the ith feature of the kth positive and negative instance respectively. The higher the F-score for a certain feature i, the more discriminative and important this feature is to the overall classification accuracy. The final result of this procedure is the set of features that produced the highest accuracy on the sample set. We used a set of known malware samples and benign apps from the Google Play Store to determine the subset of discriminative features and their corresponding F-score to achieve the best classification results using an SVM with an RBF kernel. We did not include our whole set of labeled data in the feature selection process to avoid overfitting the training data.

4.3 Evaluation

In this section we provide a detailed evaluation of Marvin. We present our datasets, our training procedure, and how Marvin fares when assessing known malicious and benign apps. Furthermore, we evaluate how Marvin performs on apps from unknown origins, and how well it maintains its classification performance over time. Finally, we investigate the most decisive features when distinguishing benign from malicious apps.

4.3.1 Dataset Selection

Our dataset includes a total of 124,189 Android apps collected between June and October 2012 that we used for training and testing. We further extended this dataset with an additional 11,634 apps collected from January 2013 to May 2014 to evaluate Marvin's retraining effectiveness. Table 4.2 lists the distribution of apps across all datasets.

Ground truth. The ground truth and input for our feature selection and training are two labeled datasets: (1) a set of known benign apps, and (2) a set of known malicious apps. We collected the benign apps from the Google Play Store and scanned them with VirusTotal [11], excluding all apps that triggered a response from any of the 43 AV scanners used by VirusTotal. For the malware dataset we retrieved samples for 30 variants of the 16 most widely distributed malware families according to F-Secure [74] from VirusTotal. In order to diversify our malware collection, we extended this dataset with 1,894 malware samples belonging to various families that were first seen by VirusTotal in September 2012 and that matched at least 10 AV signatures. Additionally, we include the Android Malware Genome Project [227] and the Contagio malware dump [3] as known malware corpora.

Dataset	Total	Malware		Goodware		Labeled?
Feature Selection	9,180	4,580	49.89%	4,600	50.11%	✓
Training Set	66,891	7,406	11.07%	59,485	88.93%	✓
Test Set	28,670	3,175	11.07%	25,495	88.93%	✓
Genome Project	1,152	1,152	100%	0	0%	✓
Unknown Set	27,476	-	-	-	-	-
Total	124,189	11,733	9.45%	84,980	68.43%	
Malware Retraining	1,134	1,134	100%	-	0%	✓
Mixed Retraining	10,500	2,874	27.37%	7,626	72.63%	✓
Total	135,823	15,741	11.59%	92,606	68.18%	

Table 4.2: *Dataset overview.* Our dataset includes (1) samples collected between June and October 2012 for feature selection, training and testing (top), and (2) samples collecting from January 2013 to May 2014 for retraining (bottom).

We labeled 78% of the apps in our dataset as either goodware (around 68%) or malware (around 10%). The remaining 22% of apps did not receive any labels. We received them from unknown or untrusted sources mainly through anonymous submissions to the web interface of Andrubis (which Marvin is integrated in), sharing with other researchers, or torrents and direct downloads from one-click hosting sites. Unlabeled apps also include samples from the Google Play Store that were detected by one or more AV scanners. We also did not label samples that we retrieved from VirusTotal that matched below 10 AV signatures or matched signatures for grayware such as adware, spyware and riskware.

Sample activity. We summarize the number of apps for each class as well as the number of features extracted statically and dynamically in Table 4.3. On average, an app expresses 40 features, with two thirds extracted statically and one third extracted during dynamic analysis. The six features that were always extracted from an APK file are the app's name and features from the certificate. For 0.61% of all apps, the static analysis failed to extract any further features and only dynamic analysis features remained. The number of dynamic analysis features depends on the activity of an app in the sandbox and is zero for 5% of our apps that did not exhibit any behavior during the analysis timeframe, but which were nevertheless classified based on static analysis features alone. Apps that could not be classified because they produced no static analysis results and exhibited no behavior in the sandbox amounted to only 0.63% of our dataset.

Training and test set. We randomly split the set of labeled apps in a training set (70%), and a test set (30%). Both sets contain 11.07% malicious and 88.93% benign apps, reflecting the proportion of malware to goodware in submissions to Andrubis in November 2012. The malware set includes samples from the most prevalent families according to F-Secure, Contagio, and submissions to VirusTotal in September 2012. We retained samples from the Genome Project as an independent test set to verify Marvin's classification accuracy on previously unseen malware. We also retained a set of labeled apps for feature and parameter selection of the SVM.

Class	# of Apps	All Features			Static Analysis Features			Dynamic Analysis Features		
		min	*max*	*mean*	*min*	*max*	*mean*	*min*	*max*	*mean*
Malware	11,733	7	196	37.33	6	186	24.04	0	88	13.28
Goodware	84,980	6	1,023	34.58	6	1,019	25.60	0	488	8.9
Unknown	27,476	6	1,529	40.72	6	1,510	29.82	0	321	10.90
Total	124,189	6	1,529	36.20	6	1,510	26.39	0	488	9.81

Table 4.3: *Number of features per class.* Minimum, maximum, and mean number of features Marvin extracted through static and dynamic analysis for malware, goodware, and unlabeled samples.

4.3.2 Classification Results

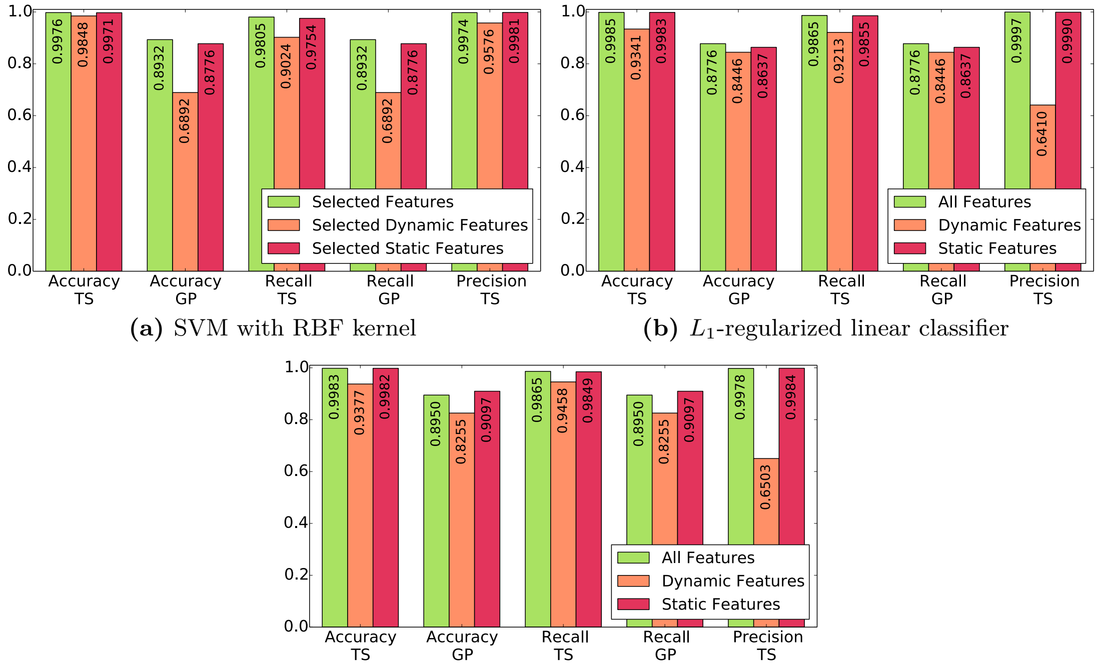
As we have detailed in Section 4.2.2, we experimented with three different classifiers: an SVM with RBF kernel and two linear classifiers with L_1- and L_2-regularized logistic regression. We evaluated both the overall classifier performance in terms of accuracy, and the time necessary to train the classifier. As we show in this section, all classifiers produced comparable classification results although with vast differences in training and testing times. The SVM with the RBF kernel and pre-selected features took on average 16.5 minutes to train and 27 seconds to classify the full test set (on average 0.95 ms per sample). The linear classifier with L_1 and L_2 regularization on all features took only two to three seconds for training and under a second for classifying the test set (on average 0.02 ms per sample).

All classifiers output a prediction whether an app is goodware or malware and the probability with which the app belongs to either of those classes. In the following, we provide a discussion of the classification results on the test set and unlabeled set, as well as the samples from the Genome Project as an independent test set.

Classification of the labeled test set. All three classifiers predict the correct class for the apps in the test set with extremely high accuracy, as illustrated in Figure 4.3: 99.76%, 99.85%, and 99.83% for the SVM, the L_1-regularized, and the L_2-regularized linear classifier respectively. The SVM misclassifies 62 malware and eight goodware apps (1.9528% false negatives, 0.0314% false positives), the L_1-regularized linear classifier misclassifies 43 malware apps and only one goodware app (1.3543% false negatives, 0.0039% false positives) and the L_2-regularized linear classifier misclassifies 45 malware and seven goodware apps (1.3543% false negatives, 0.0275% false positives).

Since our test set is unbalanced (11.07% malicious vs. 88.93% benign apps) we also take the base rate of malicious to benign apps in the test set into account and further calculate the *Bayesian detection rate* [26], i.e. the probability that an app classified as malicious by Marvin indeed is malware. Here, the SVM, the L_1-regularized and the L_2-regularized linear classifier achieve detection rates of 97.98%, 99.74%, and 98.24%, respectively.

Another important metric for the practicality of Marvin is the number of false alarms raised. Considering the false positive rate in relation to the average number of



(a) SVM with RBF kernel

(b) L_1-regularized linear classifier

(c) L_2-regularized linear classifier

Figure 4.3: *Classification results on labeled datasets.* We evaluate each classifier's performance in terms of accuracy, precision[4] and recall on the test set (TS) and the Genome Project (GP), based on all features, and static and dynamic features alone (for the SVM we used the features determined by explicit feature selection)

apps a user has installed on her device (around 90 according to a recent study [198]), Marvin raises false alarms for 0.004 apps with the best configuration (L_1-regularized linear classifier) and 0.025 apps with the worst configuration (SVM). If Marvin were used to analyze a whole app store, such as Google Play with currently almost 1,500,000 apps,[5] Marvin would raise false alarms for 58.5 apps in the best case with the L_1-regularized linear classifier and 471 in the worst case with the SVM.

Manual examination of the false positives reveals that all misclassified goodware apps raise red flags by requesting between ten and 32 permissions including the permissions to send and receive SMS, start a service on startup, install packages, remount the file system, or modify the APN settings that control the cellular data configuration. One misclassified app is a mobile security app that also includes embedded executables and dynamically loads native code in addition to requesting dangerous permissions. The majority of misclassified malware apps receive very low scores due to their inactivity during dynamic analysis or a limited amount of static analysis features.

[5] `http://www.appbrain.com/stats/number-of-android-apps` (retrieved April 27, 2015)

[5] Note that the precision on the Genome Project is always 1.0 because it contains only malware, hence we omitted it from the graphs.

Classification of the Genome Project. We excluded all samples from the Genome Project from our training and test set to evaluate the accuracy of our classification on previously unseen malware. This set consists of around 1,200 malware samples from 49 different families and includes the majority of prevalent malware families from August 2010 to October 2011 [227]. Despite the age of this dataset, which restricts the behavior observed during dynamic analysis when required network resources are no longer available, Marvin classifies close to 90% of the samples correctly. The majority of misclassified samples belongs to only three families that were not part of our training set, but can easily be added to increase detection accuracy: *DroidDreamLight*, *jSMSHider*, and *GoldDream*.

Static vs. dynamic analysis features. As illustrated in Figure 4.3, classifying apps using a combination of static and dynamic analysis features yields the best results, while classification based on static analysis features alone outperforms classification based on only dynamic analysis features. However, dynamic analysis features are the only distinctive features in several cases and are indispensable for describing an app's network behavior and classifying apps that dynamically load code at runtime, an increasingly popular practice as we found in our large-scale study on Android app behavior (see Chapter 3). Clearly, Marvin could benefit from incorporating more dynamic analysis features, such as system calls, to make the dynamic analysis more decisive, which we leave for future work (see Section 4.4). Furthermore, in ongoing work we explore the benefit of a more comprehensive GUI stimulation technique to increase the discriminative power of dynamic features with promising results.

Classification of unlabeled apps. While we were able to label the majority of apps in our dataset as either malware or goodware, we could not assign 22% of apps to either class due to a lack of ground truth. Table 4.4 lists the different sources of apps, the AV detection rates on unlabeled samples (including labels for malware and grayware), and the number of apps that are predicted as malware by each classifier. In order to estimate Marvin's performance on this dataset, and as the majority of market apps was crawled before Bouncer was deployed, we investigate the apps from the Google Play Store that Marvin classified as malware in more detail. For each classifier the majority of high scores are assigned to variants of the *Android.Trojan.IconoSys* family. This assessment is based on certificates reused among apps, SMS-related permissions and sending SMS, and contacting the URLs `blackflyday.com`, `iconosys.com` and `smsreplier.net` during dynamic analysis. Other correct malware detections come from the *Android.Trojan.FakeDoc* and *Plankton* families. Marvin also flags spyware that requests permissions to send and receive SMS, leaked the IMSI, and dynamically registered broadcast receivers for `sms_received` and `new_outgoing_call` events. The majority of benign apps that receive high scores are apps utilizing the aggressive AirPush ad library, which has been banned from the Google Play Store in the meantime [185] and which is considered adware by most AV scanners. Also flagged are two mobile AV apps that start a service on startup, load native code, dynamically register broadcast receivers for package installation and uninstallation events, and request permissions to read and send SMS as well as modify the APN settings.

Source	Total Apps	Labeled Apps	Unlabeled Apps	Detected by AVs			Detected by Marvin		
				>5	>10	>20	SVM	Linear L_1	Linear L_2
Google Play Store	90,951	85,008	5,943	201	105	3	180	24	101
Torrents	4,241	132	4,109	10	9	1	108	16	24
Direct Downloads	1,565	16	1,549	4	3	0	22	5	5
Sample Sharing	4,716	729	3,987	166	140	71	567	331	382
VirusTotal	10,949	9,950	999	913	783	188	767	759	794
Contagio	324	198	126	126	126	86	112	112	114
Genome Project	1,152	0	1,152	1,150	1,134	454	1,029	1,011	1,031
User Submissions	12,863	0	12,863	3,865	3,104	122	4,739	4,388	4,330
Unique Apps	124,189	95,561	28,628	5,955	4,930	698	7,102	6,244	6,365

Table 4.4: *Classification results on unlabeled samples.* Number of unlabeled apps by source that were classified as malicious (a) by >5, >10, and >20 of the 43 AV scanners used by VirusTotal (including labels for malware and grayware), and (b) by different classifiers evaluated for Marvin [6].

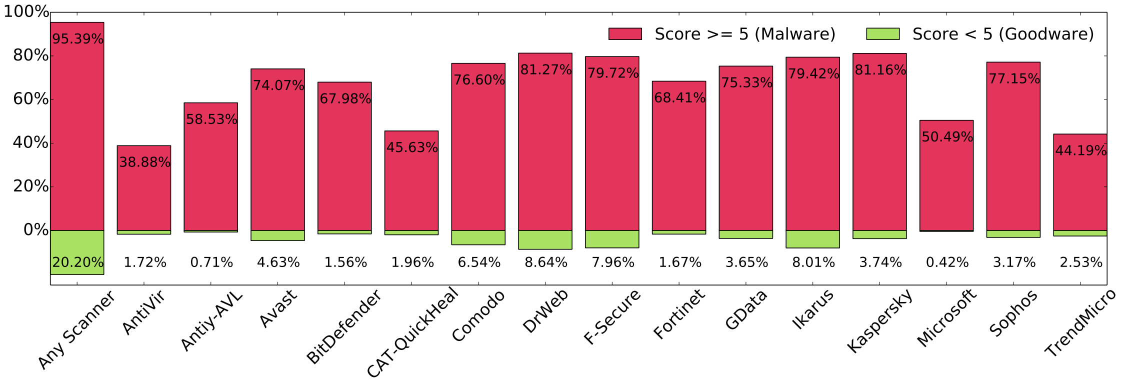


Figure 4.4: *Percentage of apps detected by AV scanners.* We distinguish between apps that Marvin assigns high (top, red bars) and low malice scores to (bottom, green bars).

Comparison with AV results. In order to investigate our results on the unlabeled samples further, and to compare our method to traditional malware detection techniques, we evaluated Marvin's detection rate on the unlabeled set against the individual AV scanner results provided by VirusTotal in more detail. Here, we compare Marvin against the 15 best-performing AV scanners in terms of detection rates, i.e., the AV scanners that labeled the largest number of apps as either malware or grayware. Figure 4.4 illustrates that there seems to be considerable disagreement between the individual scanners. While 95.39% of apps that Marvin flags as malware (i.e., assigns a malice scores >= 5 with the L_2-regularized linear classifier) are detected by any of these 15 scanners, those scanners individually detect a maximum of around 80% of apps each. In contrast, the same scanners detect between 0.42% and 8.64% of apps that receive scores <5 as malware or grayware. In total 20.20% of apps that are detected by any scanner receive a low

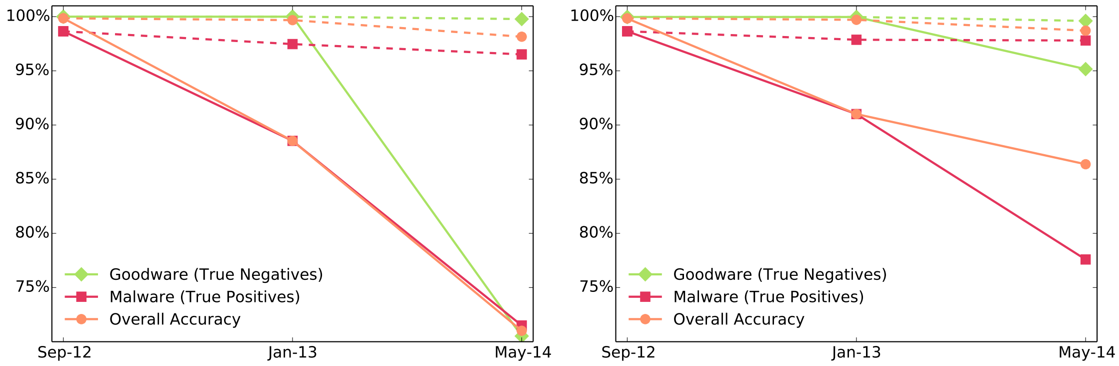
[6] Note that there is some overlap between the apps we received from different sources. Hence, a subset of apps from untrusted sources e.g. torrents and direct downloads, which we also crawled from the Play Store and that were not detected by any AV scanner are thus labeled as goodware.

score from Marvin, however, a large number of those apps are labeled as grayware: For example F-Secure labels the majority of apps Marvin does not detect as either adware or *Application:Android/FakeApp*, and for Comodo almost all apps that Marvin failed to assign high scores to trigger a generic detection heuristic for *Unclassified Malware.*

Retraining strategy. As we already investigated in our work on the development life cycle of Windows-based malware (see Chapter 2), the malware landscape changes over time, due to new monetization opportunities being exploited, and the never-ending arms race between malware authors and security researchers. Consequently, Marvin's classification model will age and will be outdated at some point. Thus, frequent retraining is an absolute necessity. Our retraining intervals are, however, dependent on the availability of ground truth. Apvrille et al. [21] estimated that it takes security researchers up to three months to detect new Android malware in the wild. Our original training data includes malware samples seen until September 2012. Thus, as a worst case scenario we obtained an additional retraining set comprised of 1,134 malicious apps that we randomly selected from apps that were first submitted to VirusTotal in January 2013 and that matched 10 or more AV signatures. With the three months old training data Marvin is still able to correctly classify 88.54%/91.01% (L_1/L_2-regularized linear classifier) of these new apps correctly.

In order to evaluate the retraining effectiveness, we then split the new test data in a training set (70%) and a testing set (30%). When training the classifier on the new training set only, Marvin achieves a detection rate of 96.47% and 94.71% (L_1- and L_2-regularized linear classifier) on the new test set. However, the detection rate on the original testing data deteriorates to 57.48% and 43.18%. When combining the original training data with the new training set Marvin's detection rate recovers to over 98%. In order to guarantee the optimal performance of our classifier our strategy is thus to retrain Marvin regularly with new apps as soon as new AV labels become available while still keeping part of older training apps in the training set.

With another retraining set comprised of both malware and goodware we further evaluate how malicious and benign features shift over longer periods of time. We randomly selected 2,874 malicious apps from new submissions to VirusTotal from January 2013 to May 2014. Additionally, we crawled 7,626 of the most popular apps from Google Play in April and May 2014. The proportion of malware to goodware (27% to 73%) in this set reflects the increasing number of malware submissions to Andrubis. On this dataset the L_1-regularized classifier correctly classified 71.50% of malware and 70.52% of goodware, while the L_2-regularized classifier proved superior by accurately classifying 77.59% of malware and 95.16% of goodware. Note that in this pathological case no retraining was performed for over one year. Yet, these results indicate that benign features are more stable over time and retraining with additional goodware is necessary less frequently than with malware. When extending the original training data with the apps from this retraining set Marvin achieves a true positive rate of 96.52%/97.80% (L_1/L_2-regularized linear classifier) and a true negative rate of 99.78%/99.61%, demonstrating the long-term practicality of Marvin without requiring any adjustments to the feature extraction process. We illustrate the results of our retraining experiments in Figure 4.5.



(a) L_1-regularized linear classifier (minimizing number of features)

(b) L_2-regularized linear classifier (maximizing number of features)

Figure 4.5: *Retraining effectiveness.* We re-evaluated the accuracy of Marvin after three months, and again after more than a year without *any* intermediate retraining. Even though its accuracy deteriorated significantly when using only the original training data from September 2012—solid lines, it recovered to its original performance when retrained with new samples—dashed lines. Using more features for classification (L_2 regularization) proved beneficial in the long term. Furthermore, features of goodware and thus detection accuracy on this class are far more stable than features of malware.

4.3.3 Distinctive Malware Features

Finally, we examine the number and nature of the most decisive features when distinguishing benign from malicious apps to better understand Marvin's risk assessment.

Number of relevant features. For classification with the SVM the F-scores indicate how discriminative the individual features are. We calculate the F-score for three different subsets of the labeled dataset: the subset exclusively used for feature selection of the SVM, the subset used for training the SVM, and the labeled dataset as a whole. Additionally, we calculate the percentage of features out of the whole evaluation feature space that are expressed in a particular subset and judged to be at least marginally relevant by this measure. The results can be found in Table 4.5: On the feature selection set, 11.74% of the feature space is assigned an F-score > 0, on the labeled training set 62.33%, and on all labeled data 81.43%. Static analysis features are assigned the highest-ranking scores, with `send_sms`, `receive_sms`, and `read_phone_state` permissions being the top three. Related work by Felt et al. [79] also determined that those three permissions were characteristic of malware. However, more dynamic analysis features than static analysis features were assigned high scores. This is also reflected by the feature subset selected by the feature selection: It predicts the highest accuracy for a set of the 27,808 highest ranked features with a minimum F-score of 0.000109. 18,335 of those features are dynamic, 9,473 are static. We use this subset of features for the evaluation of the SVM detailed in this section.

Dataset	Static Analysis Features Non-zero F-score		Dynamic Analysis Features Non-zero F-score		All Features Non-zero F-score	
	#	maximum	#	maximum	#	% of all
Feature Selection Set	32,022	1.52	26,299	0.27	58,321	11.74%
Labeled Training Set	214,487	1.24	95,239	0.31	309,726	62.33%
All Labeled Samples	279,685	1.24	124,976	0.32	404,661	81.43%

Table 4.5: *Number and percentage of features considered relevant for SVM classification,* i.e., features that received an F-score > 0. The F-score indicates how discriminative a feature is for classification of either class. While static analysis features individually are ranked more highly, generally more dynamic analysis received high F-scores in our feature selection, which considers 11.74% of features important for classification.

	Static Analysis Feature Weights				Dynamic Analysis Feature Weights				All Feature Weights	
	Malware		Goodware		Malware		Goodware			
	#	mean	#	mean	#	mean	#	mean	#	% of all
L_1	783	1.19	394	0.50	272	1.04	120	0.29	1,569	0.32%
L_2	6,558	0.15	208,161	0.01	23,638	0.01	71,624	0.01	309,981	62.38%

Table 4.6: *Number and percentage of features considered relevant for linear classification,* i.e., features that received weights != 0. We distinguish between negative and positive weights and thus features used for the classification of malware and goodware. L_1 regularization results in only 0.32% of all features (and more static analysis than dynamic analysis features) being used for classification. L_2 regularization considers 62.38% of all features (and more dynamic analysis features for the malware class) important for classification.

For the linear classifiers, their model contains the weights assigned to each feature and thus its importance in the classifying process. Weights can either be negative or positive, depending on which class a feature is an indicator of. Features that are assigned a weight of zero have no part in the decision making process. As already mentioned in Section 4.2.2 the weighting strategy differs greatly for L_1- and L_2-regularized linear classification. While L_1 regularization tends to minimize the number of features used for classification, L_2 regularization tends to include all features in the classification process. This fact is illustrated in Table 4.6: L_1 regularization considers only 0.32% of all features for classification, while L_2 regularization considers 62.38% of features relevant. Nevertheless, both strategies yield comparable results on our test set, although L_2 regularization performs slightly better on the previously unseen Genome Project. Most importantly, L_2 proved superior in the long run and performed noticeably better with outdated training data after several months and even more than a year, as illustrated in Figure 4.5. Both strategies performing comparably well on the datasets that are very similar to the training data, but L_2 regularization performing better on unrelated test sets suggests that L_1 regularization is underfitting the training data and the resulting model is too simple.

Relevant feature categories. In order to determine the categories of features that are the most meaningful when distinguishing between malware and goodware, we calculate the mean of all F-scores and of all weights for each category of features. The results can be found in Table 4.7a for the SVM and in Tables 4.7b and 4.7c for the linear classifiers. One drawback of the F-score is that it only outputs the degree of discrimination a feature provides, but it is no indication in favor of which class. In the case of feature weights, however, we can assume that features with a positive weight are indicative of the positive class, i.e. malware in our case. We can also highlight those features in the analysis report to provide an explanation on why a certain app was classified as malicious by Marvin.

Among the usual suspects, when it comes to features characteristic of malware, are a number of features that are unique to Android apps, such as required permissions, sending SMS, leaking information, and features related to the dynamic loading of code. The high ranking of SMS related activities and permissions among common malware features is not surprising as sending premium SMS is a popular monetization vector of Android malware [227], and mobile variants of banking Trojans such as ZeuS intercept text messages containing Mobile Transaction Authorization Numbers (mTANs) [179]. The classification that dynamic code loading is as an indicator for maliciousness is also in line with observations in related work [93, 227], although we found it to be less decisive on more recent samples in our large-scale study on Android malware behaviors (see Chapter 3). Other characteristics for malware are features extracted from the certificates used to sign the apps. Our results confirm that malware authors often reuse the same certificate across variants, or use public testing and debug certificates as stated in previous work [21]. Other distinguishing features are network-related activities that have already been successfully applied to the classification of Windows malware in the past. Looking at the list of hosts contacted during dynamic analysis, especially the top-level domains `.cn` (China), `.ru` (Russia), `.in` (India), and `.biz` are more frequently contacted by malware, while `.kr` (Korea), `.de` (Germany), `.com`, and `.mobi` domains are more likely to be contacted by goodware. The individual hosts among the highest-ranking features include C&C servers associated with malware, such as `client.a1b2c3d4e5.in` for *Android.Frogonal*[7] and `depot.bulks.jp` for *Android.Dougalek*[8].

Applications for distinctive features. The result of Marvin provides users with the malice score of an app and a detailed report on the app's static and dynamic analysis features. With the knowledge of how individual features rank in the decision making process of a classifier, we are able to highlight high ranking features and thus give users an indication as to why an app received a certain malice score. Another application of the feature ranking produced by Marvin is the integration of highly ranked features in blacklists: Marvin can reveal the hosts frequently contacted by malware and thus provide a candidate set for URL blacklists. Similarly, Marvin can also disclose the certificates that are commonly misused by malware authors to sign their apps and which, therefore, should not be trusted.

[7] `http://www.symantec.com/security_response/writeup.jsp?docid=2012-062205-2312-99`

[8] `http://www.symantec.com/security_response/writeup.jsp?docid=2012-041601-3400-99`

Mean F-score	Type	Feature Category
0.0255393	static	PermRequired
0.023951	static	UsesCrypto
0.0196447	dynamic	SMS
0.0141904	dynamic	LeakSource
0.00984	static	CertShortTerm
0.00947679	static	APKName
0.008777	static	CertValidity
0.00824016	dynamic	FileRead
0.00801642	static	ReceiverStatic
0.0065745	dynamic	DNSResult
0.005421	static	EmbeddedFileExe
0.00514366	static	CertSN
0.0051357	static	CertOwner
0.004307	dynamic	HTTPPost
0.00425988	dynamic	ReceiverDynamic
0.00350983	dynamic	FileWrite
0.003395	static	UsesDynamicCode
0.00236072	dynamic	DNSDomain
0.00230411	static	Metadata
0.00222916	static	Java
0.002118	dynamic	HTTPGet
0.0019654	dynamic	HTTPPostData
0.001671	static	ValidManifest
0.00164017	dynamic	DNSToplevelDomain
0.00161581	dynamic	HTTPPostHost
0.00147765	dynamic	HTTPPostRequest
0.001351	static	UsesNativeCode
0.00131	static	UsesReflection
0.001275	static	EmbeddedLib
0.0011585	dynamic	LoadDexClass

(a) SVM classification

Mean Weight	Type	Feature Category
2.3943	dynamic	LoadLib
2.3008	static	CertSN
2.1495	static	CertOwner
1.8109	static	EmbeddedFileExe
1.7917	static	APKName
1.7527	dynamic	SMS
1.4361	dynamic	CryptoAlg
1.1736	dynamic	DNSDomain
1.1230	static	CertShortTerm
1.0852	static	Metadata
1.0259	dynamic	ReceiverDynamic
0.9792	static	ReceiverStatic
0.9729	dynamic	FileWrite
0.9203	dynamic	HTTPPostData
0.8001	dynamic	TCPConnHost
0.7938	dynamic	LeakSource
0.7502	dynamic	DNSToplevelDomain
0.7409	dynamic	TCPHost
0.7216	static	Java
0.6678	dynamic	TCPConnPort
0.6605	static	PermRequired
0.6206	dynamic	FileRead
0.5714	dynamic	LeakToFile
0.3420	dynamic	HTTPPostIP
0.3261	static	EmbeddedScript
0.2462	dynamic	HTTPGetIP
0.2076	dynamic	HTTPPostPort
0.2060	dynamic	HTTPGetData
0.1572	dynamic	HTTPPostHost
0.1208	dynamic	HTTPGetHost

(b) L_1-regularized linear classification

Mean Weight	Type	Feature Category
2.9450	static	EmbeddedFileExe
2.3927	static	CertShortTerm
0.9612	static	CertOwner
0.8139	static	UsesCrypto
0.7963	dynamic	DNSResult
0.6521	dynamic	LeakSource
0.5984	static	UsesReflection
0.5662	static	UsesDynamicCode
0.5310	dynamic	CryptoAlg
0.4735	dynamic	Call
0.4273	static	CertSN
0.4193	static	PermRequired
0.3166	dynamic	HTTPPostData
0.3052	dynamic	TCPConnPort
0.2959	static	EmbeddedLib
0.2431	static	ReceiverStatic
0.2413	dynamic	SMS
0.2332	static	ValidManifest
0.2176	dynamic	DNSDomain
0.1988	dynamic	LoadLib
0.1961	dynamic	HTTPGetData
0.1831	static	UsesNativeCode
0.1803	dynamic	ReceiverDynamic
0.1572	dynamic	HTTPGetRequest
0.1525	dynamic	DNSToplevelDomain
0.1498	static	APKName
0.1483	dynamic	HTTPGetHost
0.1447	dynamic	HTTPGetIP
0.1381	dynamic	TCPConnHost
0.1256	dynamic	TCPHost

(c) L_2-regularized linear classification

Table 4.7: *Categories of features relevant for classification.* For the SVM we rank features according to their F-score, for the linear classifiers we rank features according to their positive weights (i.e., weights for features indicative of malware). We present the 30 highest ranked feature categories for each classifier.

4.4 Limitations and Future Work

Similar to other approaches leveraging dynamic analysis or machine learning to analyze and distinguish malicious from benign apps, Marvin has some limitations.

One shortcoming of our method to check apps on real devices is the way they are submitted. While users can submit arbitrary apps through our web interface, our mobile app currently can only access the APK archive of an app after it was successfully installed on the target device. If the user decides to try the downloaded app before submitting it to Marvin, chances are high that an infection already took place. To overcome this problem, our app would need to intercept downloads for apps from the official market. Although no official API for the Google Play Store is available, the proprietary protocol used for downloading apps has already been successfully reverse engineered [191]. To intercept installs from alternative markets, our app would need instrumentation for each available market. Alternatively, we could consider implementing changes to the Android Package Installer framework like the ones proposed for ThinAV [108]. These changes to the underlying Android OS, however, would severely obstruct the widespread usage of our system, especially by end users. On the other hand, apps are not directly installed if they are downloaded from an alternate source. Instead, the app is stored as an APK file on the device first. At this point, it can be easily analyzed and rated by Marvin without the possibility of infecting the target device.

Another limitation of any dynamic analysis approach is evasion. Possibilities to detect analysis environments range from iterating telltale device properties to reveal the underlying virtualization technology, to fingerprinting characteristics of a specific Android installation, to exploiting certain properties of emulated code within virtual machines [51,120,158,204]. However, Marvin does not rely on features extracted from dynamic analysis alone and static analysis features proved highly effective during our evaluation. Therefore, we argue that this is currently not a problem and that it can be addressed in the future by either building a more transparent analysis environment, or leveraging a system such as BareDroid [143] to perform dynamic analysis on bare metal.

Finally, malware authors who are aware of the way Marvin extracts features might try to subvert the classification by attempting a mimicry attack. To this end they would need to make their malware express as many goodware-specific and as little malware-specific features as possible. However, they would then have to keep up with the pace of our retraining and hide all malicious behavior from the dynamic analysis environment to achieve this. Furthermore, dynamic analysis is more resilient to mimicry attacks than static analysis and often able to discover the malicious behavior regardless.

For future work we plan to extend the features extracted during dynamic analysis. When malware uses native code to perform malicious activities, we currently only detect that native code was loaded, through both static code analysis and during dynamic analysis. We plan to incorporate the behavior of native code by using system-level events to enrich dynamic features. In our evaluation we show that static analysis features are equally, or even more decisive to create the model for an accurate assessment than dynamic analysis features (but not both). Incorporating system calls into our feature space can improve the behavioral models and, in turn, lead to more accurate results for the combined system. Furthermore, by using system-level events, malware utilizing root exploits can be identified and characterized more precisely. Additionally, we aim to extend dynamic analysis features by increasing code coverage during dynamic analysis. We are currently exploring how Marvin could benefit from more intelligent user interactions than the current state-of-the-art user interface stimulation in its underlying dynamic analysis environment with promising results [43].

Furthermore, application markets provide a wealth of information about the offered apps. Therefore, we plan to integrate meta information from markets, such as the number of downloads, user ratings, other apps by the same author, and the lifetime of an app in the market as additional features, which we are already collecting for malicious apps with AndRadar (see Chapter 5).

As another direction for future work we plan to evaluate the practicality of our feature set for other applications such as the detection of malicious repackaged apps. Marvin could raise alarms for apps that have a large overlap of features but are assigned very different malice scores in our ranking.

4.5 Conclusion

We presented Marvin, an effective and efficient analysis tool to assess the maliciousness of previously unknown Android apps in the form of a *malice score*. Marvin utilizes machine learning techniques to classify apps based on a rich and comprehensive feature set, which it extracts from static and dynamic analysis of a set of known malicious and benign apps. Our evaluation showed that Marvin is capable creating an accurate snapshot of malware behavior that it can leverage to assess the risk associated with apps under investigation accurately and comprehensively. In our large-scale dataset it correctly identifies 98.24% of malicious apps with less than 0.04% false positives. Furthermore, we showed that it can be efficiently retrained to maintain its detection accuracy in the long term, and to adapt to changes in the malware landscape and analysis evasion techniques.

To provide an appropriate end user experience helpful for novice and expert users alike, Marvin accepts submissions and displays analysis results through a web interface and a dedicated mobile app. In addition to the added benefit for ordinary users, Marvin provides malware analysts with the means to pre-select samples for manual investigation. By setting a report threshold, our system is able to filter malware candidates with high precision and provide detailed information about their static analysis features and dynamic behavior to facilitate further manual analysis, or integration into other automated malware analysis tools.

CHAPTER 5

AndRadar: Discovery of Android Apps in Alternative Markets

Due to its popularity with over 80% market share [101] and open model, Android has become the mobile platform most targeted by cybercriminals. In spite of a small infection rate of devices with mobile malware in the wild [118,131], the remarkable increase in the number of malicious applications (apps) shows that cybercriminals are actually investing time and effort as they perceive financial gain [112]. Indeed, the typical malicious app includes Trojan-like functionalities to steal sensitive information (e.g., online banking credentials), or dialer-like functionalities to call or text premium numbers from which the authors are paid a commission. The degree of sophistication of Android malware is rather low, although samples of current malware families found in the wild include command and control (C&C) functionalities and attempt to evade detection with in-app downloads of the malicious payload after the installation of a legitimate-looking app. Cyber criminals are focusing more on widespread distribution [112] and naïve signature evasion [167, 223] rather than attack vector sophistication.

Security vendors and researchers agree [112, 138, 192, 197] that there is an increasing trend of malicious Android apps spotted in the wild, which indicates that criminals indeed consider this a source for profits. This phenomenon created a business opportunity for new security companies, which according to Maggi et al. [133], created about a hundred anti-malware apps for Android. Interestingly, about 70% of such companies are new players in the antivirus (AV) market.

As with traditional malware, the research community has been focusing on developing systems, such as the ones presented in Chapter 3 and 4, to analyze suspicious programs to identify whether they are malicious. For Android apps this requires analyzing the application package file (APK), a compressed archive that contains resources (e.g., media files, manifest) and code, including Dalvik executables, libraries, or native code (e.g., ARM or x86). To this end, researchers have mainly ported dynamic, static, and hybrid program analysis approaches to Android and adapted them for the analysis of Android apps. There is, however, a key difference between traditional malware and Android malware: their *distribution channels*. For Windows malware a whole ecosystem developed around services for the stealthy distribution ("silent installs") of malware through pay-per-install (PPI) services [40] and drive-by downloads utilizing browser exploits [95]. In contrast, as we will discuss in Section 5.1, Android malware is distributed through *application marketplaces*, which means that there is a wealth of metadata associated with each sample, in addition to the resources contained in each APK. Additional contextual information comes from the infection mechanism; "bait and switch", malware authors branding their apps as an actual benign app distributed through other marketplaces to in order to attract victims.

Efficiency is a key requirement for monitoring malware campaigns in the large Android ecosystem. However, we observe that meta information has not been fully leveraged to this end. Indeed, as discussed in Chapter 6, related work revolves around features extracted from APK files, which in turn implies that each and every sample needs to be downloaded and processed using static and dynamic analysis techniques, which is time and space consuming.

Motivated by the need for tracking the *distribution* of Android malware across markets, we follow a different approach and propose an alternative way to *identify* them. We demonstrate that the *combination* of lightweight identifiers such as the package name, the developer's certificate fingerprint, and method signatures, creates a very strong identifier, which allows us to track apps across markets. We implemented our approach by building AndRadar, which uses a flexible workflow: It applies lightweight fingerprinting to quickly determine if a known sample has been found in a particular market. AndRadar postpones computationally expensive tasks that need access to APK files, such as binary similarity calculation, so that they can be lazily executed. This allows AndRadar to scan a full market for malware in real-time. Using AndRadar we can infer useful insights about malicious app distribution strategies and the lifetime of malware across multiple markets. For example, for a total of 20,000 crawled apps AndRadar recorded more than 1,500 deletions across 16 markets in a period of three months. Nearly 8% of those deletions were related to apps that were *hopping* from market to market, meaning the authors republished their apps in one or more different markets after they were already deleted from another market. Some markets reacted and deleted new malware within tens of days from the publication date, whereas other markets did not react at all. Interestingly, we were able to measure that the community reacts fast, flagging apps as malicious faster than the market moderation in some cases.

In summary, we make the following contributions:

- We conducted an in-depth measurement on 8 alternative Android marketplaces. In contrast to previous work, we collected the entire set of apps (318,515 overall) and not simply a random subset drawn from each market. With this dataset, we provide preliminary insights on the role of these alternative markets, with a focus on malicious or otherwise unwanted apps.

- We expand our set of observed markets and present AndRadar, a tool for searching a set of markets in real-time to discover apps similar to a seed of malicious apps. Using a set of distinctive fingerprints that are robust to commonly used repackaging and signature-evasion techniques, AndRadar can scan markets in parallel and only needs a few seconds to discover a given Android app in tens of alternative application markets.

- Using AndRadar we study and expose the publishing patterns followed by authors of malicious apps on 16 markets (the official Google Play Store and 15 alternative marketplaces). Moreover, our evaluation shows that AndRadar makes harvesting marketplaces for known malicious or unwanted apps fast and convenient.

- As part of ongoing work [60] we provide the findings of AndRadar to the research community through a public web service: `http://admire.necst.it`.

5.1 Market Characterization

As we detail in our discussion of related work in Chapter 6, previous research shows that in 2011 the majority of malicious or otherwise unwanted Android apps were distributed through so-called *alternative marketplaces* (often also called third-party markets). An alternative marketplace is any web service whose primary purpose is to distribute Android apps. For instance, blogs or review sites that occasionally distribute apps do not qualify as marketplaces. According to our definition, we were able to find 89[1] markets as of June 2013. The *raison d'etre* of such alternative markets depends on three main factors: *country gaps* (i.e., the Google Play Store is inaccessible from certain countries), *promotion* (i.e., markets tailored to help users find new interesting apps), and *specific needs* (i.e., markets that publish apps that would be bounced by the Google Play Store).

Regarding malware distribution, since the first measurements conducted in 2011 a lot has changed: Researchers, security vendors and media continuously raise concerns about the explosive growth of Android malware. According to a recent estimate [182], as of 2013, companies have invested about US$9 billion in mobile device and network security, and installation of anti-malware software has become the de-facto requirement for mobile devices.

5.1.1 The Role of Alternative Marketplaces

Given the above premises, we wanted to investigate whether alternative marketplaces employ any security countermeasure to avoid the spread of malicious apps. To this end, we conducted a series of probing experiments, in July 2013, aimed at assessing the response of these markets to dangerous apps. We submitted known malicious apps taken from the Android Malware Genome Project [227] to 7 markets (i.e., *andapponline*, *androidpit*, *appzoom*, *brothersoft*, *camangi*, *opera*, *slideme*) and analyzed their reaction. To

[1] Although previous work reported 194 markets in 2011 [203], and as many as 500 in 2012 [112], no details such as the URL or name were mentioned.

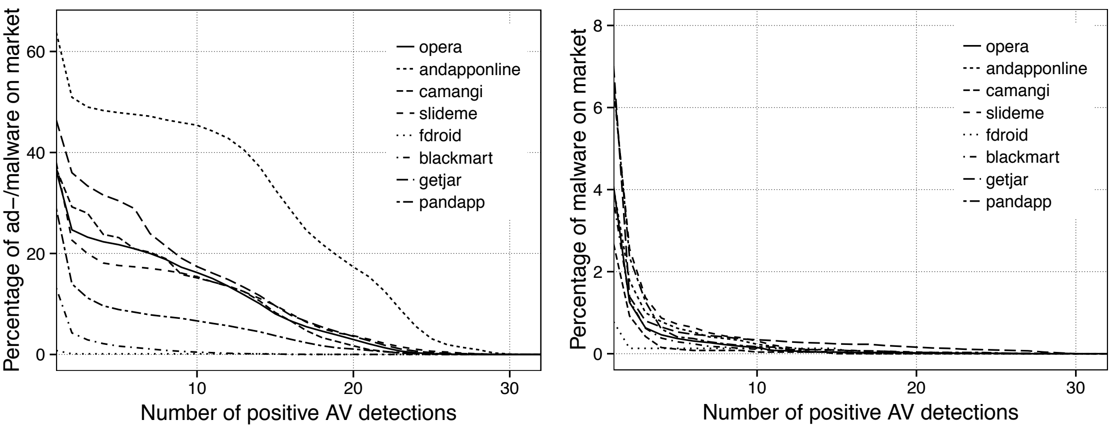


Figure 5.1: *Presence of known, unwanted apps in markets.* Percentage of apps on alternative markets classified as positives by [1-32] AV scanners, including adware (left) and excluding adware (right).

deter users from downloading the apps, we included explicit indications that they were malicious and should not be installed. To the best of our knowledge (i.e., by tracking the download counts), those apps were not downloaded. However, certain markets such as *andapponline* never bounced/removed samples from 10 known families (e.g., *DroidKungFu*, *BaseBridge*). This motivated us to conduct a more thorough analysis. Therefore, we crawled 8 alternative marketplaces (see Table 5.3 in Section 5.3 for a complete listing) between July and November 2013 entirely, obtaining 318,515 apps along with their metadata, which varies across markets (e.g., application name, version, uploader's nickname, category, price, download count, required permissions). We then extended this crawling experiment, including metadata from a larger set of markets, as described in Section 5.3.

5.1.2 Preliminary Findings

Using this initial collection of apps, we set out to answer the following questions:

Do alternative markets distribute known, unwanted apps? We submitted our entire dataset to VirusTotal in order to collect AV labels. As illustrated in Figure 5.1, our analysis showed that the infection rate is non-negligible. Even if we exclude adware, there are still about 5–8% malicious apps overall on the crawled markets (15,925–25,481 distinct apps detected by at least 10 AV scanners). This is clearly an underestimation due to the nature of malware labels and our exclusion of adware. Interestingly, some markets are specializing in distributing adware. This finding is in line with Symantec's recent report [200] discussing the "madware" phenomenon, the practice of creating ad-aggressive mobile apps to obtain revenue.

We conducted the remaining preliminary experiments on the apps marked as malicious, excluding adware. We list the ranking of the top families found in Table 5.1.

Label	#
Android/Generic	2,397
Trojan/AndroidOS.eee	2,119
Trojan.AndroidOS.Generic.A	1,020
AndroidOS/Denofow.B	768
AndroidOS/Denofow.B	765
Suspect.Package.RLO	682
WS.Reputation.1	593
UnclassifiedMalware	555
Android/DrdLight.D!tr	517
AndroidOS/FakeFlash.C	455
Android-PUP/Hamob	443
AndroidOS/FakeFlash.C	428
Application:Android/FakeApp.C	358
Trojan:Android/Downloader.F	339
Andr.Trojan.Zitmo-2	223
Android/DDLight.D!tr	204
Trojan.AndroidOS.FakeFlash.a (v)	192
Android Airpush	182
AndroidOS/FakeFlash.A	174

Table 5.1: *Top malware families* found in our preliminary experiments.

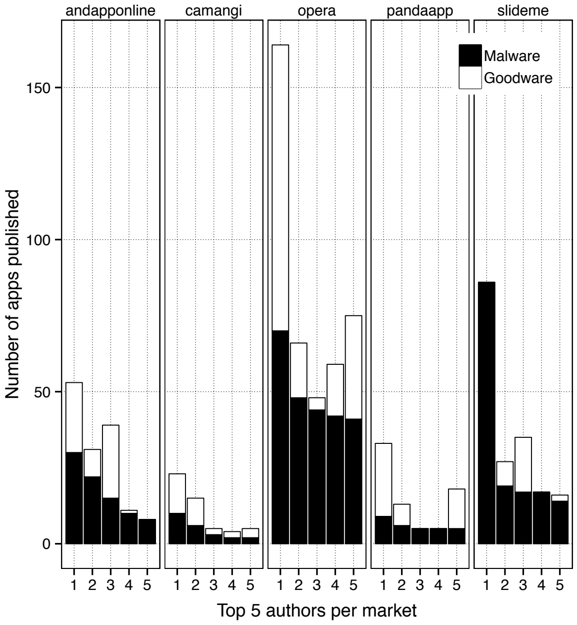


Figure 5.2: *Top 5 authors* in each market ranked by the number of apps they published.

Do alternative markets allow the publication of malicious apps? Based on the number of apps published, we ranked the authors of those 5 markets that reported author information reliably (e.g., *blackmart* simply caches that information from Google Play Store). Unfortunately, as shown in Figure 5.2, these markets permit the top authors to freely to publish both malicious and benign apps. This finding further amplifies the previous results, because top authors are supposedly well visible and known to the market's operators and community due to the larger number of apps published with respect to other authors.

Do malicious apps have distinctive metadata? Previous work focused on devising static and dynamic features, extracted through program analysis techniques applied to the APK files, that characterize malicious apps (see our discussion of related work in Chapter 6). However, given the central role of alternative markets in Android malware distribution, we wanted to understand if malware can be identified solely by its *metadata*, meaning all ancillary data available on each market (file size, download count, etc.). As Figure 5.3 shows, due to repackaging, the file size is a feature to consider: Statistically speaking, malware samples are slightly larger than goodware samples because of the additional malicious code added during repackaging. Similarly, we observed that for those markets that report the download count (e.g., *getjar*), malicious apps are downloaded more often than benign apps by at least an order of magnitude. One possible explanation for this finding is that malware authors reportedly use *app rank boosting services* to inflate download numbers in order to improve their app's ranking and thus visibility to users [96].

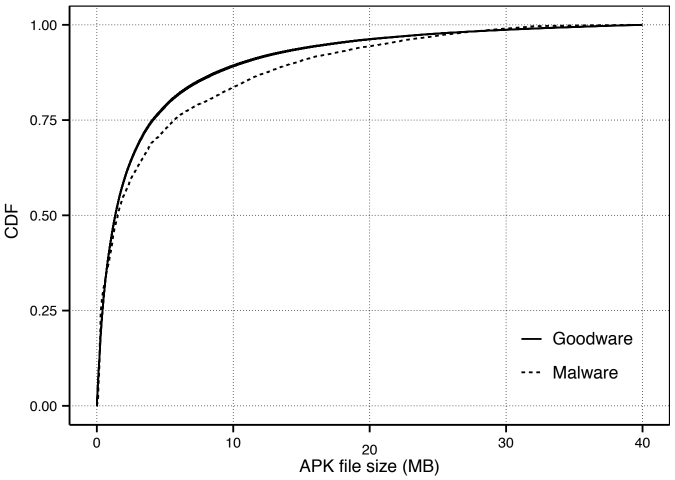
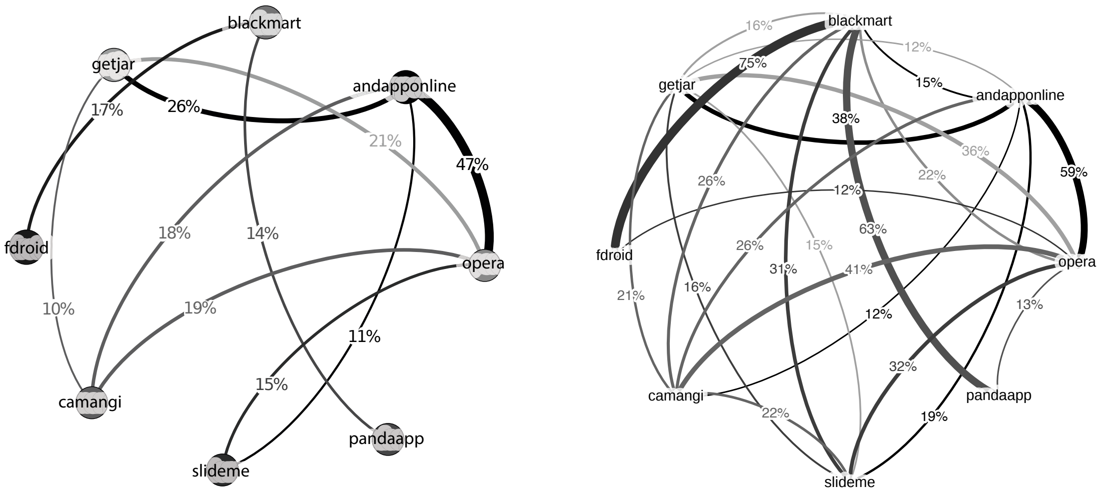


Figure 5.3: *File size comparison of malware and goodware.* Malicious apps are slightly larger than benign apps due to repackaging.



(a) Intersection of apps by MD5 **(b)** Intersection of apps by package name

Figure 5.4: *Flow of apps across markets.* Percentage and thickness of edges indicate the percentage of apps in common between two markets.

How are markets related to each other? We calculated the set intersection of apps across markets by taking the MD5 hash of the APK file or the package name as the identifier. As we can immediately see from the results presented in Figure 5.4, the number of shared apps across markets is significant, with notable examples such as *andapponline–opera* sharing 47%/59% of MD5s/package names, and *andapponline–getjar* sharing 26%/38% respectively.

Conclusion. From this preliminary analysis, it appears that alternative markets are not proactively removing malicious apps from their databases. Understandably, the volume of apps to be screened is large and the current analysis methods rely on running expensive and error-prone analyses on each submitted APK. Moreover, given the non-negligible flow of apps across markets, we are concerned that malicious developers may be able to implement a "failover" strategy to have their samples migrate from market to market in order to hinder removal.

These findings motivate us to devise an Android market radar, called AndRadar. AndRadar uses lightweight and transparent techniques that permit the quick scanning of alternative markets for malicious or otherwise unwanted apps and allow us to track apps and their metadata across different markets.

5.2 System Description

In this section we present the architecture of AndRadar. First, we discuss various challenges we faced while designing and implementing AndRadar, and then we describe its various components in detail.

5.2.1 Challenges

AndRadar aims at discovering a particular Android app, possibly indicated as malware or otherwise unwanted apps by an AV scanner, in the official Google Play Store as well as alternative markets. This is a non-trivial task as we show in this section. Below we list the most significant challenges we had to overcome while building our prototype.

Marketplaces plethora. During our preliminary experiments (see Section 5.1), we found 89 alternative marketplaces, run by companies or individuals, whose quality in terms of security aspects is questionable. As demonstrated by our long-term marketplace study, which took months to complete, crawling markets is challenging. First, space and time requirements increase quickly with the number of markets. Second, and most important, each market runs its own software. This essentially means that for each market we want to monitor we need to analyze its API for searching and downloading apps. Normally, this involves discovering two URLs, one for searching for an app and one for downloading a discovered app along with its metadata. Unfortunately, for many markets this process is not straightforward. For example, many of them strictly require user authentication—especially markets with specialized content, like adult content—or are provided in the form of a mobile app, which needs manual reverse engineering for revealing the market API. Finally, while running AndRadar we also experienced cases where markets, for example Google Play and *appchina*, changed their web templates during our experiments. Changes in a market's web templates essentially require us to carry out further adjustments in the engine we use for extracting app metadata.

App mutation. The diversity of the marketplaces is not the only challenge we have to overcome. Apps can slightly mutate from market to market. This might be due to legitimate reasons, for example two markets host two different versions of a particular app.

App may also be repackaged by another author either to add additional functionality missing from the original app, or to profit from a popular app by including advertising libraries or malicious code [203]. Detecting repackaged apps, maybe the most popular form of Android malware, has been the target of recent related work [58, 225, 226]. AndRadar's primal goal is not to detect if a particular app has been repackaged, but locating an app—possibly malware—across different marketplaces. Research in repackaged app detection is orthogonal to AndRadar. Nevertheless, it can substantially assist AndRadar in discovering repackaged versions of apps across different alternative markets. Recall that the common wisdom suggests that popular apps hosted in the official market are enhanced with malicious functionality, repackaged and published to alternative marketplaces. We envision that, due to the immediate popularity gained by alternative markets and due to the continuously growing defense systems in the official Google Play Store, consisting of automated app vetting with *Bouncer* [128] and more recently also a manual app review process [156], malware authors will further target alternative markets. Therefore we expect them to start repackaging legitimate apps found in popular (alternative) markets and then publishing the produced malware in less popular markets that employ fewer or even no security mechanisms at all. In such cases, AndRadar can use existing algorithms and heuristics for real-time detection of repackaged apps across multiple marketplaces.

5.2.2 Architecture Overview

In a nutshell, AndRadar's task is to probe a number of marketplaces for malware and, if found, track changes in the app and its metadata. Figure 5.5 shows how the components of AndRadar interact to achieve this task.

Essentially, AndRadar has three core components: The first one is the *seed*, which is composed of apps that have been flagged as malware by a set of tools or services. This is the input set that AndRadar uses for locating apps across alternative markets. The second component is the *search* component. For each app in the seed, AndRadar uses a set of crawlers for discovering the app in alternative markets as well as the Google Play Store. Finally, the third component is the *tracker*, which, once an app is found, downloads the actual APK file and its additional metadata and keeps it in storage for further statistics. We now look into each of the three components in detail.

5.2.3 Malicious App Seed

To begin with, AndRadar requires a set of known malware or otherwise unwanted apps that we call the *seed*. Because of its dynamic, online functionality, AndRadar works best with a continuous, accumulating feed of malicious apps in contrast to a static set. Apps for the seed can come from a variety of sources including new additions to manually vetted malware repositories, feeds from submissions to AV scanning services that are detected as malicious by multiple scanners, or submissions to public dynamic app analysis sandboxes.

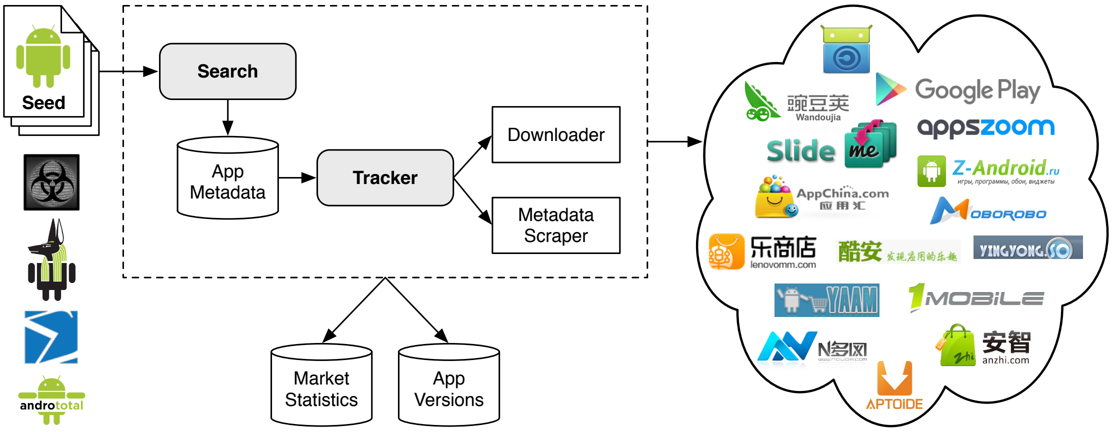


Figure 5.5: *System overview of* ANDRADAR. The *seed*, which is composed of apps that have been flagged as malware, is used as input to the *search* for locating apps across markets. Once an app is found, the *tracker* instructs the *downloader* to fetch and store the APK file and the *metadata scraper* to store additional metadata on a regular basis. As a result, we obtain metadata from apps and their updated versions, as well as statistics about the distribution of malware across markets.

For our prototype ANDRADAR receives feeds from (1) the VirusShare [10] malware repository, (2) submissions to VirusTotal [11] that trigger > 10 AV signatures, and (3) submissions that ANDRUBIS [123] flagged as suspicious during dynamic analysis based on the classifier, MARVIN, we presented in Chapter 4 [122]. However, ANDRADAR could be easily extended to add further sources for malware such as submissions to AndroTotal [133] or any other services that provide some form of malware classification and access to their samples feeds.

We characterize each app in the seed by four app identifiers that we use as similarity features to match two apps at different levels of confidence (summarized in Table 5.2):

Package name. The package name is the "official" identifier of an app. It serves as an installation-time ID, i.e., no two apps on a given device can share the same package name. Some markets, such as Google Play, use it also as a unique reference, but in principle developers are not restricted from creating an app with an already existing package name. Therefore, in the context of ANDRADAR, which operates on a multi-market domain, we use the package name to locate apps inside a market (see Section 5.2.4) and treat it as a *weak match* between two apps. However, ANDRADAR is not restricted to this identifier, as we further will discuss in Section 5.4.

Certificate fingerprint. Apps in Android are signed with the private key of their developer. Android uses this signature to enforce update integrity by only allowing updates signed with the same key, as well as resource sharing and permission inheritance between apps from the same author [30]. We can thus use the fingerprint of the certificate used to sign the app as a further identifier. Since the key is specific to an author, a match of the fingerprint is a strong indicator that the matching apps stem from the

App Identifier	Match Level
Package name	weak match
Package name + certificate fingerprint	strong match
Package name + method signatures	strong match
Package name + certificate fingerprint + method signatures	very strong match
MD5 hash	perfect match

Table 5.2: *Different match levels based on app identifiers.* Based on the combination of four app identifiers we match two apps with varying levels of confidence.

same author, unless the author has shared her private key or is using the key pair that is publicly available with the Android source code. We thus treat a match of package name and author fingerprint as a *strong match*.

Method signatures. By leveraging Androguard [1,62,163] we can generate signatures of the methods in the application code. A signature is an abstract model of a method's intra-procedural control flow, enriched with information on the package of further called methods. To compare signatures, Androguard uses the normalized compression distance. For AndRadar, we limit the scope of the signatures to methods that are either in the main package or in the package that contains the app's main activity, thus excluding third-party libraries that would skew the comparison results and improving performance. We define everything above 90% code similarity to be a *strong match*. In addition, we define the combination of a method signature, fingerprint and package name match as a *very strong match*.

MD5 hash. In a very straightforward way, a match between the MD5 hash of two APK files means that two apps are identical, i.e., a *perfect match*.

5.2.4 App Discovery

The *search* component probes markets for a given app, based on its package name. We chose the package name for our search procedure, since it provides a strong heuristic to identify a sample from the seed inside a market and some markets use it to uniquely identify apps in their app catalog. This allows us to efficiently locate apps from the seed in markets based on a *weak match* without the need to download and process the candidate apps first.

Of course, as we discuss in Section 5.4, a malware author could randomize the package name from market to market, but this would actually run against the malware author's own scheme when trying to trick users into downloading her repackaged version of a popular app. Thus, a malicious app trying to remain hidden from AndRadar would substantially reduce its visibility to potential victims. As a future extension, we may add options to search for application names, words appearing in the title of market listings, or through other metadata that users might use to locate an app inside a market. This, however, would require AndRadar to track and download multiple candidate apps and their metadata from each market in order to locate samples matching the seed app.

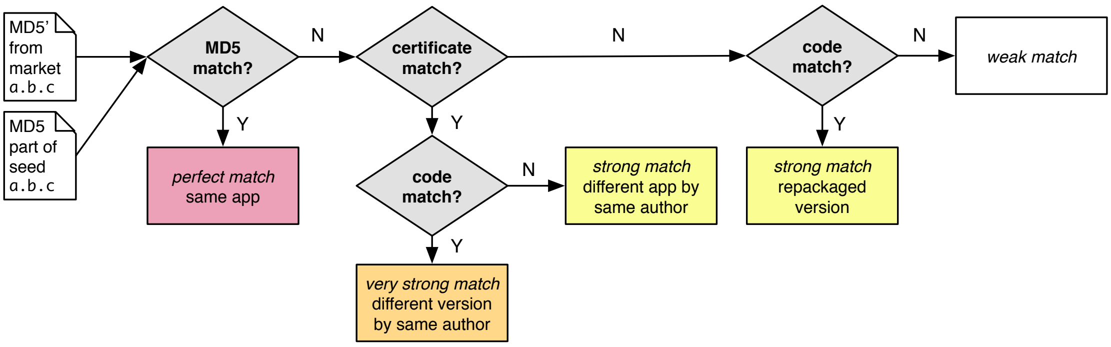


Figure 5.6: *Flow chart of* AndRadar*'s app matching.* For each app found in a market we match it against the seed app using MD5 hashes (*md5 match*), certificate fingerprints (*certificate match*) and code similarity based on method signatures (*code match*).

For markets such as Google Play, *appchina*, *anzhi*, *wandoujia* or *coolapk*, which use the package name as an internal reference to the apps in their catalog, the lookup is straightforward, as the package name is typically part of the app's URL in the market. Other markets use different internal identifiers and thus require a more elaborate search procedure, however *all* of the markets currently part of our study provide an app's package name as part of its market listing. In case a market does not provide the functionality to query apps by package name directly, we split the package name along the separators and feed the individual parts to the market's search interface, discarding well-known common parts such as, e.g., "com." Once we locate the package name on the results page, the search is considered finished. Otherwise, we continue by crawling the individual market listings that are returned by the search query.

Finally, based on an author's publishing habit, apps might appear in our seed before they are released to one of the markets we monitor. As a consequence the search component probes all the markets for all malicious apps at regular intervals regardless whether they have been located before or not.

5.2.5 App Tracking

Once the search component finds an app in a market, the *tracker* investigates the corresponding market listing. The tracker first invokes the *downloader* to fetch the app from the market. The downloaded app is matched with the sample in the seed using the set of app identifiers summarized in Table 5.2 as similarity features. We present the flow chart of AndRadar's matching algorithm in Figure 5.6. The tracker then uses the *metadata scraper* to obtain market-based metadata for each sample, from each monitored market at regular intervals, i.e., currently once a day. Metadata includes the reported version of an app as well as its price, update date, information about the developer, and popularity metrics such as download count, user ratings and reviews, etc. We further infer an app's delete date once we can no longer locate it in a market that we were previously tracking it in. Furthermore, if we detect a change in an app's metadata information, indicating a possible update, we download the new version of the app and keep it in storage.

Marketplace	Website	Language	Market Study	AndRadar
1mobile	`http://www.1mobile.com`	English		✓
andapponline	`http://www.andapponline.com`	English	✓	
anzhi	`http://www.anzhi.com`	Chinese		✓
appchina	`http://www.appchina.com`	Chinese		✓
appszoom	`http://www.appszoom.com`	English		✓
aptoide	`http://www.aptoide.com`	English		✓
blackmart	`http://www.blackmart.altervista.org`	English	✓	
camangi	`http://www.camangimarket.com`	English	✓	
coolapk	`http://www.coolapk.com`	Chinese		✓
f-droid	`http://f-droid.org`	English	✓	✓
getjar	`http://www.getjar.mobi`	English	✓	
google-play	`http://play.google.com`	English		✓
lenovo	`http://app.lenovo.com`	Chinese		✓
moborobo	`http://store.moborobo.com`	English		✓
nduoa	`http://www.nduoa.com`	Chinese		✓
opera	`http://apps.opera.com`	English	✓	
pandaapp	`http://download.pandaapp.com`	English	✓	
slideme	`http://slideme.org`	English	✓	✓
wandoujia	`http://www.wandoujia.com`	Chinese		✓
yaam	`http://yaam.mobi`	English		✓
yingyong	`http://www.yingyong.so`	Chinese		✓
z-android	`http://z-android.ru`	Russian		✓

Table 5.3: *Overview of studied (alternative) marketplaces.* Markets part of our preliminary market study and monitored by AndRadar as part of our extended experiments.

5.3 Evaluation and Case Study

In this section we evaluate AndRadar in terms of performance, and use it to reveal insights about the behavior of malware or otherwise unwanted apps in markets, including their life cycle and the publishing behavior of their authors across multiple markets.

5.3.1 Performance

AndRadar tracks apps in multiple markets in a parallel fashion. For the purposes of this study, we have incorporated 16 different markets (see Table 5.3)—15 alternative marketplaces as well as the Google Play Store. The time needed to search and download a particular app across the individual markets is illustrated in Figure 5.7. Naturally, downloading is slower than searching, but both operations take just a few seconds to complete for the majority of markets. However, the download of an app is only initiated when an app is located in a market for the first time or when metadata information indicates a possible update. Furthermore, both operations depend on the network conditions, as well as the load the market is experiencing at the time, but since AndRadar crawls all markets in parallel, we are only constrained by the slowest market. We list the amount of apps we can track in each market per day in Table 5.4. As it can be seen we are able to track tens of thousands of apps daily.

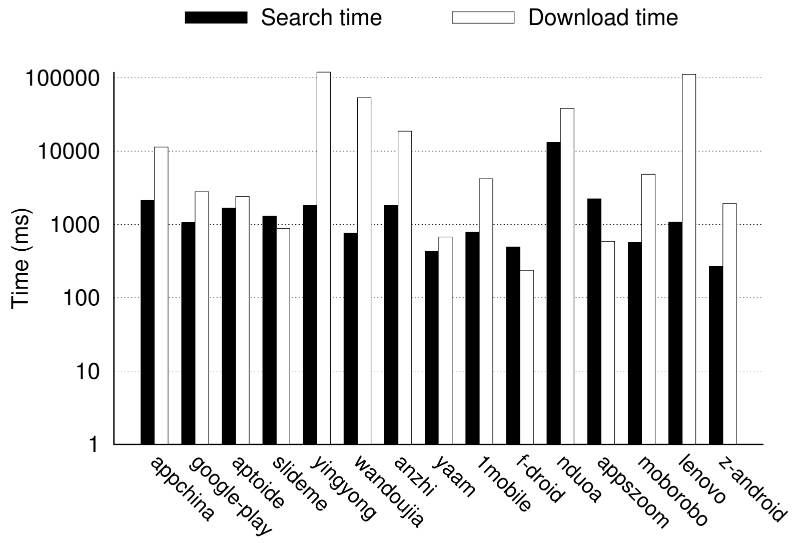


Figure 5.7: *Average time needed for searching and downloading an app on each market.* Since AndRadar handles all markets in parallel, searching and downloading a particular app on *all* markets is constrained by just the slowest market. Furthermore, we only need to download apps when metadata information indicates a possible update.

Market	Search	Download	Total	# Apps/Day
f-droid	0.49s	0.24s	0.73s	118,163
yaam	0.43s	0.67s	1.11s	77,996
slideme	1.30s	0.88s	2.18s	39,662
z-android	0.27s	1.93s	2.20s	39,319
appszoom	2.23s	0.59s	2.82s	30,605
google-play	1.06s	2.79s	3.85s	22,441
aptoide	1.67s	2.41s	4.08s	21,199
1mobile	0.79s	4.20s	4.99s	17,305
moborobo	0.57s	4.83s	5.40s	15,994
appchina	2.13s	11.36s	13.49s	6,406
anzhi	1.80s	18.69s	20.49s	4,217
nduoa	13.06s	38.18s	51.25s	1,685
wandoujia	0.76s	53.65s	54.41s	1,587
lenovo	1.08s	111.43s	112.51s	767
yingyong	1.80s	119.56s	121.35s	711

Table 5.4: *Average search, download, and total (search + download) processing time of an app on each market.* The total number of apps we can track in each market per day shows that we are able to track tens of thousands of apps daily in the majority of markets.

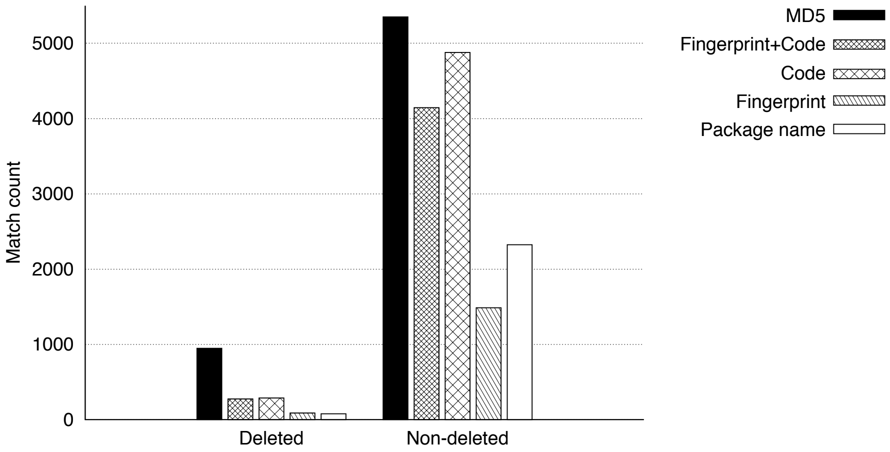


Figure 5.8: *Confidence of app matches.* Number of *deleted* and *non-deleted apps* per match type across all markets. We were able to match the majority of apps based on perfect (MD5) and very strong (certificate fingerprint and code similarity) matches, especially in the case of deleted apps.

5.3.2 Case Study

AndRadar gives us the opportunity to collect data about an app in multiple markets, study the multi-market behavior of the app, and, possibly, identify publishing patterns followed by app developers. For instance, if we use AndRadar with a sample of (possibly) malicious apps, we can understand how malicious apps behave across different markets. In this section, we present the insights we obtained by crawling 20,000 apps in a daily manner between August and December 2013 in 16 markets. These apps matched apps in our seed at least by package name and were identified according to the process described in Section 5.2.

For the purpose of this case study we split the set of tracked apps into two subsets: (a) *deleted*, a set that contains all apps that have been deleted at least once from a market during our observation period, and (b) *non-deleted*, a set of apps that have never been deleted from any of the markets.

Since AndRadar checks each app located in one of the markets against the malicious app from the original seed using a set of similarity features (detailed in Table 5.2 and Figure 5.6), we have a spectrum of confidence regarding the maliciousness of the collected apps. In Figure 5.8 we plot the number of apps in the collected dataset (across both deleted and non-deleted apps) that we could match with each similarity feature.

If we identify an app with a perfect match (MD5 match) that is removed after a period of time (corresponding to the black bar in the deleted group in Figure 5.8), we assume that the market administrators did this for a reason and found something malicious about the app, thus strengthening our initial suspicion. Conversely, on a weak package name match, a missing reaction from the market administrators (corresponding to the white bar in the non-deleted group in Figure 5.8) indicates that the app located in the market is the benign version the author of the malicious seed app used as a disguise.

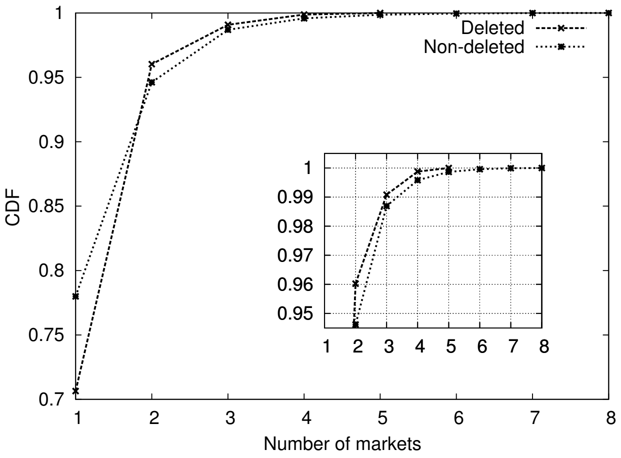
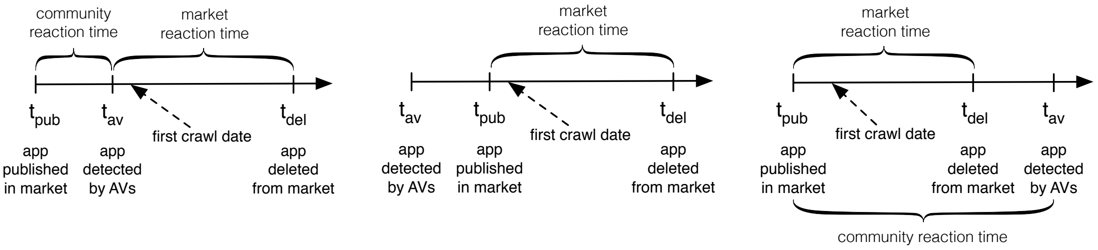


Figure 5.9: *CDF of the simultaneously distribution of malware through multiple markets.* We found a number of deleted and non-deleted apps located in more than five markets at the same point in time.

Notice that AndRadar can track a significant fraction of apps having a high confidence level about their maliciousness based on their similarity to malicious apps in the seed: As Figure 5.8 shows, we found perfect and very strong matches for the majority of apps in our dataset. In detail, for the deleted set we found a perfect (MD5) match for 56.54% of apps, a very strong (fingerprint and code) match for 16.42% of apps, and strong matches for 17.13% (code) and 5.29% (fingerprint) of apps. For the non-deleted set we found 29.45% perfect matches, 22.79% very strong matches, and 26.83% and 8.17% strong matches based on code and fingerprints respectively. Only 4.70% of deleted and 12.78% of non-deleted shared nothing more than a weak package name match with apps from the seed.

In Figure 5.9 we plot the CDF of the two sets, deleted and non-deleted apps, over the number of markets each app has been located in by AndRadar at the same point in time. This figure justifies our initial concern that malware authors indeed leverage the plethora of app markets to distribute malware. A significant portion of apps simultaneously leverages more than five markets for distribution (roughly 1/3 of the markets we have been monitoring). As an example, we were able to locate the malicious app *King Pirate* (`com.letang.game101.en.f`) in five different markets. In some of those markets the app has been available for over a year and thus reached a considerable amount of downloads. By the end of our experiments, it was only deleted from one market:

1. *appchina:* online since March 2012, 1,000-5,000 downloads
2. *aptoide:* online since May 2012, 270 downloads
3. *wandoujia:* online since August 2012, 1,430 downloads
4. *1mobile:* online since July 2013, 25,808 downloads
5. *lenovo:* online from October 2012 to October 2013



(a) *Normal life cycle:* An app is deleted from a market after it has been flagged by AVs.

(b) *Malware hopping:* An app is published to a market after it already has been flagged by AVs.

(c) *Market self-defense:* An app is deleted from a market before it has been flagged by AVs.

Figure 5.10: *Life cycle patterns of a malicious app in a market.* We distinguish three cases based on the order of the time an app is published in a market (t_{pub}), the time an app is detected by AV scanners (t_{av}), and the time it is deleted from a market (t_{del}). AndRadar starts tracking a malicious app in a specific market when it becomes part of its seed at any time after t_{pub} (*first crawl date*).

The app advertises itself as a legitimate game previously available in the Google Play Store.[2] The repackaged version adds the functionality to manipulate SMS, install additional packages, and perform payments. It was first submitted to VirusTotal in September 2012, flagged by the first AV scanners in December 2012, and since then identified by 16 scanners as a Trojan, under the names *Android/Ksapp.D* or *Android/Qdplugin.A*. Clearly, in cases like this, it is desirable for market operators to remove the app from their catalog as soon as possible. To aid them in doing so, we are planning to integrate an automated notification system into AndRadar as discussed in Section 5.4.

Finally, we take a look at how fast both the security community and the application markets react to new malware and whether a multi-market strategy enhances the lifetime of malware. We identified three typical patterns for the life cycle of a malicious app:

Normal. In the most common case, an app is first published in a market at t_{pub}, it is later identified by the community and flagged by (some) AVs at t_{av}, and at a later point deleted from the market at t_{del}. We define as the *community reaction time* the period $t_{av} - t_{pub}$, i.e., the time it takes the community to find a malicious app in a market, and as *market reaction time* the period of $t_{del} - t_{av}$, i.e., the time it takes the market operator to delete a malicious app. We depict this behavior in Figure 5.10 (a).

Malware hopping. In this scenario, malicious apps are *republished* in different markets after being flagged as malware by AVs and potentially being deleted from other markets. In this pattern, an app is published in a market at t_{pub}, but has been identified by the community at an earlier point t_{av}. At a later point the app is deleted from the market at t_{del}. We define the period of $t_{del} - t_{pub}$ as the market reaction time. We depict this behavior in Figure 5.10 (b).

[2] `https://play.google.com/store/apps/details?id=com.letang.kpe` (no longer available)

Type	# of Apps	Percentage
Normal	1,508	90.57%
Possibly malware hopping	131	7.86%
Possibly market self-defense	26	1.56%

Table 5.5: *Distribution of the life cycle patterns for all deleted apps.* The large majority of apps follows the “Normal” case, but we also found evidence of malware hopping from market to market and market self-defense.

Market	# of Deleted Apps
google-play	1,281
appchina	236
anzhi	83
wandoujia	48
lenovo	15
1mobile	1
aptoide	1

Table 5.6: *Number of deleted apps across markets.*

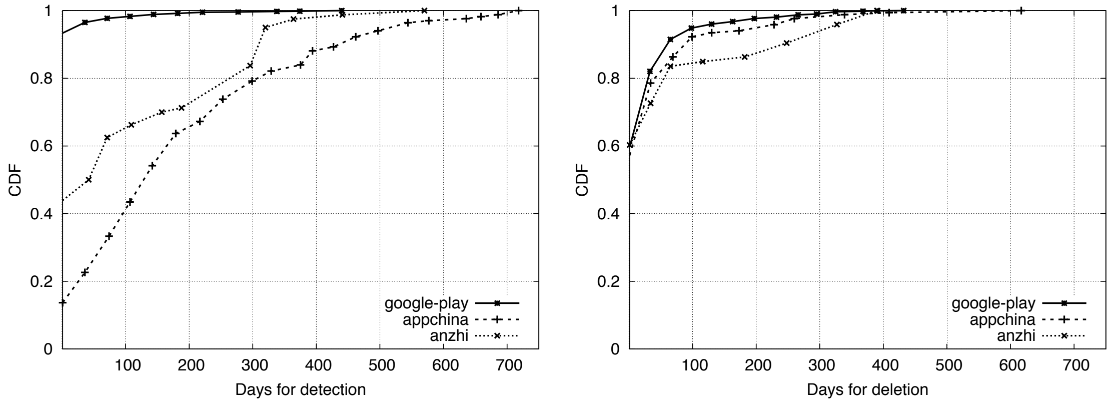


Figure 5.11: *CDF of community and market reaction times.* Time needed for AVs to detect apps as malware (*community reaction time*, left) and time needed for markets to delete apps after they have been flagged as malware (*market reaction time*, right) on three markets. Malicious apps on Google Play are detected much faster by the community than in other markets, the majority in only a matter of days. Google Play is also faster in deleting malicious apps, in most cases within tens of days.

Market self-defense. Markets can sometimes filter malicious apps even before they are flagged by AVs. In some instances, an app is published in a market at t_{pub}, at a later point the app is deleted from the market at t_{del}, and at even a later point the app is flagged as malware by AVs. Again, $t_{del} - t_{pub}$ is the market reaction time. We depict this behavior in Figure 5.10 (c).

We present the distribution of all deleted apps among these three scenarios in Table 5.5. The majority of apps follows the “Normal” case, but AndRadar could also identify apps that followed the other two cases, finding evidence that malicious apps jump from market to market, possibly for survival, and also evidence that some markets remove apps using some internal security mechanism.

For all deleted apps that follow case (a) in Figure 5.10 we measured the community reaction time, which is the time needed for AVs to flag a particular app, once this app was published in a market, and the market reaction time, which is the time needed

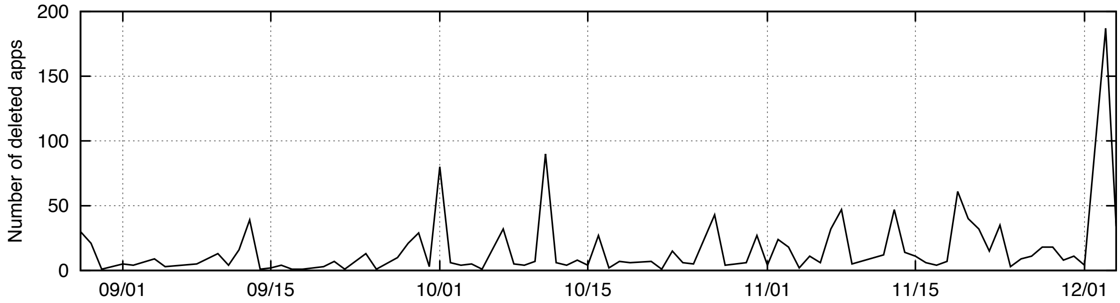


Figure 5.12: *App store cleaning of Google Play.* Number of apps deleted from Google Play on a daily basis between September and December 2013.

for a market to delete an app that was flagged by an AV as malware. We present the distribution of app deletions per market in Table 5.6. We further depict the community reaction time and the market reaction time for the three markets that deleted the most apps in Figure 5.11. The following insights can be gained from this figure:

First, each market has a different reaction behavior. It is evident that apps that are published in Google Play reach the AVs community faster than those in other markets. The majority of Google Play apps are submitted to AVs just a few days after publication.

Second, Google Play is also the fastest market to react when apps are flagged as malicious by AVs. It takes tens of days for Google Play to delete the malicious apps. The other two markets (*appchina* and *anzhi*) have a similar, but significantly slower, behavior.

Third, there is a small but not negligible fraction (less than 4%) of apps, which are deleted from markets only after several months (in some cases after more than a year). After manual inspection of these incidents, we discovered that such malicious apps fall into the *gray area* of adware, and are thus sometimes considered not dangerous enough to be removed. For example, due to policy changes Google only recently decided to remove apps including intrusive ad libraries such as AirPush from the Play Store [174]. In another recent example, researchers discovered "vulnaggressive" (aggressive and vulnerable) versions of the ad library AppLovin being used in popular apps that were subsequently updated or removed [221].

As illustrated in Figure 5.12, AndRadar can record developments like this. Google seems to clean its store in regular intervals, with the number of deletions increasing after the market policy changes concerning adware came into effect at the end of September 2013 and the vulnerabilities in AppLovin were disclosed. In fact, out of all the apps that we observed being deleted from Google Play between August 28, 2013 and December 4, 2013, 86.73% of those apps are detected at least by one AV scanner as adware. Furthermore, almost 90% of those adware apps include libraries such as AirPush, Leadbolt, AdWo and Apperhand that display push notification ads [183] that are now banned by Google's new policy. Some of those apps were available in the market for more than a year and were downloaded 100,000–500,000 times. For example, the app `com.airbit.soft.siii.oceano` was deleted from Google Play after 409 days of its upload and many AV vendors flag it as AirPush adware.

5.4 Limitations and Future Work

For our prototype AndRadar was configured to discover apps by their package name as the monitored markets either distinguish apps by this identifier or at least allow discovering apps by their package name without the need to download them first. Also, previous work reported that malware authors tend to use valid and legitimate looking package names in an effort not to attract attention [203, 227]. A recent report by F-Secure [76] found 23% of malicious apps posing as legitimates ones by imitating their package name. Consequently, they classified apps using the original package and application name but requesting additional permissions as malicious. We also found a tendency of malware authors to reuse package names in our own large-scale study [123] (presented in Chapter 3), even in the case of random looking package names, which we observed being reused for thousands of individual apps.

Alternatively, in order to counteract malicious app authors randomizing the package name on a per-app basis or simply modifying single letters similar to typosquatting, AndRadar can query markets for other identifiers. Possible candidates are application titles, parts of their description or image characteristics of the icons and screenshots advertising an app's functionality. In order to attract users and lure them into downloading their apps, malicious authors need an identifiable "brand," e.g. by piggybacking on popular apps from other markets. Thus, if malicious authors decide to evade the discovery by AndRadar, this would invariably lower their visibility to users.

Another limitation of AndRadar is its calculation of code similarity between apps, as current binary similarity measurements for Android exhibit accuracy and scalability issues. AndRadar tries to mitigate this by limiting the scope of the comparison to the main application's code, and by lazily executing such computationally expensive tasks. However, due to its flexible architecture, AndRadar can be extended to use more scalable binary comparison techniques and also include other characteristics from the apps' resources and their visual similarity.

For future work we can incorporate a notification system that warns market operators about the presence of malicious apps in their app catalog. Depending on the type of match between the malicious seed and the apps found in the markets, AndRadar could issue warnings with different levels of confidence.

Furthermore, in the presented form AndRadar does not fully take advantage of all the metadata available though marketplaces, such as information about individual developers and other apps they are publishing besides the ones AndRadar is tracking. Follow up work [60] thus explores how the dataset and infrastructure of AndRadar can be leveraged to build malware detection methods based on reputation: App developers can be judged by the type of apps they are publishing, i.e., if they publish malware and adware, and by their publishing habits across markets. More generally, the trustworthiness of markets can be evaluated based on the number of malicious apps in their catalog and their reaction. The results of this system, called Admire (Android Developers & Marketplaces Intelligence and Reputation Engine), are available through a public web service: `http://admire.necst.it`.

Finally, since AndRadar tracks different versions of malicious apps across markets, as well as discovers all updated versions of an app it tracks in a single market, we can leverage this data to identify further publishing patterns over longer periods of time. This data also allows us to observe the development life cycle of Android malware and evolution of the malicious functionality over time, similar to our work on Windows malware presented in Chapter 2.

5.5 Conclusion

Our work started from an in-depth measurement performed on 8 alternative Android marketplaces, by collecting their entire set of apps and analyzing various characteristics. This measurement provided us with significant preliminary insights on the role of these alternative markets, with a focus on malicious or otherwise unwanted apps. This is by far the most current measurement of the alternative marketplaces. Even the most recent work that we surveyed is based on data collected back in 2011.

Our findings motivated us to design and implement AndRadar, a complete system to monitor alternative markets for malware in real-time, leveraging the wealth of metadata associated with each sample. We demonstrated that the *combination* of lightweight identifiers such as the package name, the developer's certificate fingerprint, and method signatures, creates a very strong identifier, which allows us to track apps across markets.

Thanks to the efficiency of AndRadar, we were able to measure the lifetime of malware across multiple markets in real-time. For example, we tracked more than 1,500 app deletions across 16 markets over a period of three months. We discovered that nearly 8% of the deletions were related to apps that were hopping from market to market.

AndRadar was also able to identify and track malicious apps still available in a number of alternative app markets. For future work we plan to integrate an automated notification system that informs market operators about potentially malicious apps in their catalog. We believe that efforts such as ours can be successfully leveraged by marketplaces to "predict" upcoming malware distribution campaigns, so as to provide early warnings and prompt remediations. Indeed we found out that, for some markets (i.e., Google Play Store), the community contribution is essential to quickly react against published malicious and unwanted apps.

As part of ongoing follow up work the dataset and infrastructure of AndRadar helps in building a reputation for markets and developers. The results are available through a public web service, which also allows security researchers to query apps and their metadata tracked by AndRadar.

Furthermore, we can also leverage the different versions of malicious apps that AndRadar tracks to identify further publishing patterns such as how malware authors change the malicious functionality of their apps over time as part of our future work.

CHAPTER 6

Related Work

Malware targeting the Windows operating system has been a significant threat for Internet users for more than a decade. Consequently, a large body of related work has focused on the analysis and detection of Windows malware. These techniques, and most of the techniques we are applying in this work, revolve around two core concepts: (1) Static analysis, to inspect a program without actually executing it. Since source code for malware samples usually is not available, static analysis has to be performed on binary code. (2) Dynamic analysis, to inspect a program during execution, usually in a sandbox. Both techniques have their strengths and weaknesses: While static analysis could cover all possible execution paths of a malware sample theoretically, it is easily obstructed by code obfuscation techniques in practice [142]. Dynamic analysis on the other hand can observe high-level behavior of the monitored malware. However, its main drawback is limited code coverage and that it is prone to analysis evasion when a sample is able to fingerprint the analysis environment and, in turn, does not show any malicious activity in the sandbox [51, 120]. Analyzing Android apps, which recently has become a highly active field of research, uses the same basic principles, but has to address challenges unique to the mobile environment, such as the limited resources of mobile devices, the UI-driven nature of Android apps, and unique monetization and distribution techniques.

In this chapter we first focus our discussion on Windows malware analysis techniques on work related to Beagle concerning binary code comparison techniques (Section 6.1). We also present related work calculating the similarity between Android apps, which is related to how we match apps as part of AndRadar. We then discuss work related to Andrubis on the analysis and detection of Android malware (Section 6.2), and the classification of Android malware related to Marvin (Section 6.3). Finally, we discuss Android market studies related to AndRadar (Section 6.4).

6.1 Binary Code Similarity

We apply techniques to measure binary code similarity for Beagle to detect code changes between multiple versions of the same malware family. We further apply Android app similarity measures in AndRadar to match apps discovered in market against malicious apps from the seed.

While related to the field of software similarity and classification (as well as plagiarism and theft detection), the degree of difficulty when comparing malware samples is aggravated by the adversarial setting—malware employs obfuscation and run-time packers [199] to impede static analysis. Also, while for benign software source code is available in some cases, this is very rarely the case for malware. Cesare and Xiang [45] present a comprehensive review of existing methods to analyze the similarity of non-binary and binary code, including malicious code. The discussed methods extract syntactic and semantic features at different levels of abstraction (e.g., raw binary code, instructions, control and data flow) to use as *birthmarks* for finding similar code.

Similarity of Windows malware. Several approaches address the issue of finding shared code ("code clones") between binaries. BitShred [106] uses n-grams over binary code and bloom filters to efficiently and scalably locate reused code in large datasets. Since it relies on raw byte sequences, however, this approach is less robust than Beagle's graph-based approach to syntactic changes in binary code. Sæbjørnsen et al. [176] identify code clones in binaries of benign software using normalized assembly instruction, making their approach vulnerable to instruction reordering and insertion of junk code. Jin et al. [110] propose semantic hashes for functions based on the input-output behavior of basic blocks. Their hashes capture the output of each block for a set of pseudorandom input values. Similarly, BinJuice [117] extracts a semantic "juice" from basic blocks through symbolic interpretation. The juice captures algebraic constraints between variables and generalizes registers and constants. Above approaches are optimized to find exact code clones or similar code fragments in a large corpus of unrelated binaries. In contrast, Beagle's aim is to find shared functions between related versions of the same malware family and to quantify their differences.

BinHunt [85] facilitates patch analysis by accurately identifying which functions have been modified by a patch and to what extent. For this, it compares the program call graphs (CG) and control flow graphs (CFG) using symbolic execution and theorem proving to prove that two basic blocks are functionally equivalent, regardless of transformations such as register substitution and instruction reordering. BinHunt, however, does not work on packed code and its efficiency decreases as the amount of code differences increases. iBinHunt [140] makes BinHunt more resistant to function obfuscation by extending it with taint tracking and compares binaries on an inter-procedural CFG instead of an intra-procedural one. Most recently, Ming et al. [141] implemented several optimizations to speed up iBinHunt and extended it with a generic unpacker. Beagle's binary comparison component is also based on CFGs, and is robust to some code transformations such as basic block and instruction reordering and register substitution.

While less precise than BinHunt, our approach is faster and highly scalable and can also be used to contrast a binary against multiple others.

Automatically identifying the significant changes between two malware samples is a task for which also commercial software is available, such as BinDiff [2]. Although BinDiff's exact comparison algorithm is proprietary, similar to BEAGLE it performs structural comparison of the CFG, approximating the graph isomorphism problem using a set of heuristics [67, 81]: It first identifies exact matches based on the number of nodes and edges as well as calls of each function in the CFG, and then expands the search to similar functions in the neighborhood [187]. BinSlayer [38] combines BinDiff's approach with the Hungarian algorithm to perform a bipartite graph matching and improve BinDiff's matching accuracy. BEAGLE goes well beyond such approaches, however, as we propose an automated approach to map these structural changes to the behaviors expressed during runtime. In other words, we automate part of the analysis task that is traditionally still being performed by a human analyst.

Most recently, Egele et al. proposed Blanket Execution (Blex) [70], which performs dynamic similarity testing of binaries. Blex matches functions by executing them in a randomized controlled environment and comparing their side effects. The main goal of Blex is to compare binaries that have been produced by different compilers toolchains with different optimization levels. Similar to the above-mentioned approaches, Blex does not assign behavioral semantics to function matches.

Instead of measuring the similarity between different binaries, iLine [107] aims at inferring the lineage of software, i.e., establishing a temporal relationship between versions of a given binary based on their binary similarity. However, iLine focuses primarily on benign binaries, for which ground truth, for instance in code of open source projects, is available. Ruttenberg et al. [175] use clustering of features extracted with BinJuice to identify shared malware components and thus establish evolutionary relationships between (unpacked) malware samples. However, both approaches do not attach behavioral semantics to the identified components, making their results less meaningful than BEAGLE's. Furthermore, BEAGLE is already aware of the lineage of the monitored samples by design, given the way it collects them—through letting them automatically update themselves.

The work most closely related to our own is Reanimator [139], which identifies the code responsible for a malware behavior observed in dynamic analysis at instruction granularity and uses CFG fingerprints [116] as signatures to detect the presence of the same code in other malware samples. Our techniques for mapping behavior to code are less precise, and produce results at function granularity. The advantage, however, is that BEAGLE does not require (extremely slow) instruction-level logging, and we are thus able to apply our techniques to a significantly larger dataset.

Malware development life cycle. Roberts [169] presents some early, high-level insights on the malware development life cycle, such as trends in the implementation of delivery and protection mechanisms, largely based on manual analysis efforts. Part of the malware development process is the testing phase—submitting samples to AV scanners and sandboxes to fine-tune evasion techniques [216]. Recently, Graziano et al. [94]

proposed a system to automatically identify such samples under development from submissions to Anubis. Beagle's techniques to find similarity between updated malware versions can be used on top of their system to automatically identify changes between different iterations of the identified samples, e.g., to track parts of the code that changed between submissions in order to successfully evade detection. BotWatcher [29] performs long-term tracking of botnets to observe their life cycle. In contrast to Beagle, the system requires continuous execution of each bot sample, and the authors limited their evaluation to 3 to 7 days for 4 case studies. Furthermore, while inferring high-level behaviors from low-level system state changes during dynamic analysis, similar to the behaviors observed by Beagle, BotWatcher does not perform any static analysis to find differences between the updated bot samples.

Further related work studying the development of malware focused on its distribution mechanisms: Rossow et al. [171] performed a long-term analysis of malware downloaders and the C&C infrastructure they rely on. Caballero et al. [40] performed measurements on pay-per-install services (PPI) and harvested the malware samples distributed through them. They also observed the repacking rate of the distributed binaries—6.5 days on average—but also encountered samples which are being repacked on the fly for every download. However, both studies did not perform an in-depth analysis of the code and behavior changes of subsequent malware versions, as we performed with Beagle.

Similarity of Android apps. Quantifying the similarity between two Android apps is currently an active research topic. One challenge, compared to finding similarity between Windows binaries, is, that Android apps often contain a large number of third-party libraries, e.g., advertisement libraries. Existing approaches thus often whitelist external libraries, either for scalability reasons, or because shared libraries skew the results of code clone detection. Ready-to-use tools such as Androsim [163], which is part of Androguard [1, 62] and which we use in the current implementation of AndRadar, can assist reverse engineers in determining the similarity of Android apps, but exhibit accuracy and scalability issues. Indeed, we had to limit the scope of Androsim to make it usable in a large-scale scenario with AndRadar.

Measuring the similarity between Android apps has been mainly applied for the detection of repackaging, i.e., detecting legitimate apps that have been modified to include additional malicious code. PiggyApp [225] proposes to decouple primary from non-primary application modules. The authors observe that the malicious payload, which is piggybacked to legitimate apps, simply adds non-primary modules. Based on this finding, they propose a feature vector—based on the Android APIs, requested permissions, and intents—to distinguish repackaged apps from their respective legitimate apps. DroidMOSS [226] uses a fuzzy hashing similarity metric on instruction sequences to compare two apps and to determine whether one is the repackaged version of the other. DNADroid [58] leverages information from the program dependence graph (PDG) to create a structural comparison criterion based on graph isomorphisms, which allows finding pairs of matching methods to detect plagiarized apps. AnDarwin [59] also leverages the PDG for finding similar code segments between apps, however, it focuses on scalability and performs clustering of similar code on a market-scale. ViewDroid [217] detects repackaged apps based on their visual similarity to the original app, using directed graphs of their user interface views and navigation relationships as features. This approach is more resilient to code obfuscation than the above code-based repackaging detection approaches, and scalable enough to perform repackaging detection on a large-scale. WuKong [208] performs a two-phase repackaging detection approach to improve scalability. The first, coarse-grained, detection step compares app based on the call frequencies of Android APIs. The second, fine-grained detection step compares code segments based on the usage of variables. Finally, Juxtapp [97] determines based on opcode n-grams whether apps contain instances of known flawed code, exhibit code reuse that indicates plagiarism or piracy, or are (repackaged) variants of known malware. Differently from the above-mentioned approaches, this approach does not explicitly concentrate on repackaging (although it effectively finds repackaged apps), thus it is more generic. Moreover, it has a strong focus on scalability, proposing a similarity metric that is applicable to map-reduce frameworks.

Approaches for repackaged app detection are orthogonal to AndRadar. Although their goals are different from ours, their methods can in principle be applied to AndRadar to track versions of malicious apps across markets, as long as they fulfill our scalability requirements.

6.2 Android Malware Analysis

Compared to countermeasures against Windows malware, countermeasures for Android malware, and arguably the sophistication of the malware itself, are still at an early stage. However, the analysis and classification of malicious Android apps has been a highly active field of research in the past years.

As one of the main differences compared to Windows malware analysis, related work in the domain of Android malware identified the limited processing resources of smartphones and the need to move security defenses off-device and to the cloud: SmartSiren [53] uses collaborative virus detection and detects anomalies in the communication activity collected from a smartphone running its agent. Oberheide et al. [149] proposed reducing on-device resource consumption and software complexity by sending samples to a server where they can be scanned with a behavioral detection engine. Paranoid Android [162] replicates the execution of an app in the cloud to perform resource intensive detection techniques, such as AV scanning or dynamic taint analysis, to identify exploits. ThinAV [108] modifies the Android Package Installer to consult web-based AV scanners before installing an app. We also implemented Andrubis as a cloud-based service and analyze Android apps off-device in our sandbox. Its analysis results can also readily be integrated into the aforementioned services using the malice score computed by Marvin in addition to the AV scanning results.

Dynamic analysis techniques. While Andrubis also performs static analysis, its core component is the dynamic analysis. In theory, the same techniques used for the analysis of Windows binaries can be applied for the analysis of Android apps and we can leverage a wealth of insights from related work on dynamic analysis of Windows malware. Egele et al. [69] provide a summary of dynamic analysis techniques proposed for the analysis of Windows binaries. However, the analysis of Android apps comes with its own unique challenges.

One unique characteristic of mobile devices is the wealth of personal information stored on them. Thus, the detection of data leaks is of particular interest during dynamic Android app analysis. TaintDroid [71] was the first work to propose taint tracking for monitoring data flow dependencies and data leakage in Android apps and is now at the core of many sandboxes, including Andrubis, to track data leaks within the Dalvik VM. DroidScope [214] is a dynamic analysis system solely based on virtual machine introspection (VMI) on the system level, outside of the Android OS. While this approach has advantages, such as whole-system taint analysis instead of just at Dalvik VM level, the delicate reconstruction of Java objects and the like from raw memory regions requires substantial adaption effort with each Android OS update. Andrubis also performs system-level analysis (in addition to monitoring on Dalvik VM level) implemented through QEMU VMI, however, limited to the analysis of native code, which is complementary to its dedicated Dalvik VM monitoring and provides a more complete picture about apps loading native code and using root exploits.

Another challenge when it comes to analyzing Android apps is the interaction with the apps under analysis: Android apps are largely UI driven and in addition to a main and other activities, apps can register additional entry points, such as broadcast receivers for certain (system) events and services running in the background. In order to fully observe an apps behavior during runtime, an appropriate stimulation of all those entry points is thus paramount. SmartDroid [222] and AppsPlayground [166] aim at improving the stimulation of apps during dynamic analysis by driving the app along paths that are likely to reveal interesting behavior through targeted stimulation of UI elements. Their approaches can be seen as intelligent enhancements of the Application Exerciser Monkey and our custom stimulation of activity screens in Andrubis. They are largely complementary to our work, which focuses on stimulating broadcast receivers, services and common events, instead of UI elements. In fact, CuriousDroid [43] shows how automated human-like user interactions can be integrated into Andrubis.

Large-scale dynamic analysis. Our main goal with Andrubis was to build a system for the large-scale analysis of Android apps and make it publicly available. A number of other Android app analysis platforms has been a proposed in the literature, however, not all of them are publicly available or suitable for large-scale analysis. Bläsing et al. [37] were the first to propose a dynamic analysis platform for Android, AASandbox, based on system call monitoring. ANANAS [68] is a dynamic analysis framework focusing on extensibility through modules for logging file system, network and system call activity. DroidRanger [228] pre-filters apps based on a manually created permission-fingerprint before subjecting them to dynamic analysis. In contrast to this approach, we analyze every app, yielding full behavioral profiles to base our evaluation on. Furthermore, DroidRanger performs monitoring through a kernel module instead of VMI and focuses only on system calls used by existing root exploits, possibly missing future exploits. Finally, DroidRanger does not employ stimulation techniques. Andlantis' [36] aim is to perform Android app analysis at scale and consequently focuses on the design of the analysis backend. In addition, Andlantis only provides a virtual network and simulates Internet services, severely limiting the scope of the observed network behavior. In contrast to ANDRUBIS, none of the above tools are publicly available.

Other dynamic analysis systems that are publicly available are CopperDroid [4, 168], Tracedroid [9, 201], SandDroid [8], Mobile-Sandbox [7, 186], and A5 [205]. CopperDroid performs out-of-the-box system call monitoring through VMI and reconstructs Dalvik behavior by monitoring Binder communication. Tracedroid generates complete method traces by extending the Dalvik VM profiler and was subsequently integrated into ANDRUBIS, but it is also available as a standalone service. SandDroid performs monitoring of the Dalvik VM, but does not allow any network connections to the outside and therefore misses behavior in apps checking for Internet connectivity [204]. Mobile-Sandbox monitors native code through `ltrace` in addition to instrumenting the Dalvik VM. A5 [205] focuses on scalability, and currently only performs network analysis to generate IDS signatures. However, while A5 has only received around 3,000 submissions to date, both SandDroid and Mobile-Sandbox seem to be unable to cope with their submission load: SandDroid has only analyzed around 25,000 samples at the time of our experiments and samples we submitted have been stuck in the input queue for almost nine months, while Mobile-Sandbox reports a backlog of over 300,000 samples with no samples seemingly being analyzed (and the system is no longer being available as of October 2015). We emphasize that, to the best of our knowledge, ANDRUBIS is the only dynamic analysis sandbox operating on a large-scale, providing a thorough analysis on both Dalvik and system level, and typically returning a report in ten minutes or less.

Malware characterization. ANDRUBIS allows us to build a comprehensive Android malware dataset and gain insights in their behavior from studying over 1,000,000 different apps, similar to a study performed by Bayer et al. [31] on a dataset of 900,000 Windows samples ANUBIS received within its first two years of operation. To the best of our knowledge, we have provided the most comprehensive study on Android malware behavior study to date. Other studies characterized malware on significantly smaller datasets, and using almost exclusively manual analysis. Felt et al. [79] analyzed a total of 46 iOS, Symbian and Android malware samples collected between 2009 and 2011 to provide one of the first surveys on mobile malware and their authors' incentives. The Android Malware Genome Project [227] was a further attempt to systematize Android malware behavior and provided a publicly available dataset used in many following evaluations. The dataset, however, is now significantly outdated and does not reflect the current Android malware landscape any longer: The samples that were collected between 2010 and 2011 behave significantly different than apps from 2012 to 2014, as we have shown in our evaluation of ANDRUBIS (see Section 3.3). Another available malware dataset is the one used by Drebin [22] for classifying Android malware. This dataset also includes the Genome Project and the most recent samples were collected in 2012, resulting in similar evolution-related problems. Further studies on malware behavior mainly focused on the practice of repackaging and the pervasiveness of repackaged apps in alternative app stores, as we further discuss in Section 6.4.

6.3 Android Malware Classification

The approaches presented in the previous section provide reports on an app's static features and their behavior during runtime, but they offer no automated means to assess the maliciousness of an analyzed app. We built Marvin on top of such a tool, Andrubis, to enhance the analysis report by providing a malice score and by highlighting malicious features. In this section we discuss related machine learning approaches used for the classification of Android malware.

Detection based on static analysis features. Peng et al. [153] were the first to propose risk scoring for Android apps to improve the communication of the risk that comes with installing an app to users. They evaluate different probabilistic generative models and base their scoring exclusively on the permissions that an app requests. The Android permission system also has been a core element for several other approaches: Kirin [72] ranks security-critical combinations of requested permissions based on a manual assessment. Similarly, Felt et al. [79] build a simple classification approach based on the requested permissions. Another static approach is AppProfiler [170], which alerts end users to privacy-related behavior by matching an app against a knowledge base of over 200 rules of API calls related to critical behavior. The Android Observatory [30] focuses on relationships between apps and provides an interface for users to check whether an app shares its certificate with known malware. Apvrille et al. [21] propose a heuristic engine that statically pre-processes and prioritizes samples to accelerate the detection of new Android malware in the wild. They devised 39 flags for static features that they found to be common in current malware. In contrast, Marvin does not rely on heuristics and automatically determines weights of malicious features. MAST [46] considers statically extracted features such as permissions, intent filters, and the presence of native code to perform market-scale triage and to select potentially malicious samples for further analysis. Lastly, Zhu et al. [229] proposed an app recommender system that ranks apps based on their popularity as well as their security risk, but again only consider requested permissions.

Considering the ultimate goal of Marvin—automatically classifying unknown apps—RiskRanker [93], DroidAPIMiner [13], DroidSIFT [218], and Drebin [22] are the most closely related research. RiskRanker detects high and medium risk apps according to several predetermined features, such as the presence of exploit code, the use of functionality that can cost the user money without her interaction, the dynamic loading of code that is stored encrypted in the app, and the leakage of certain information. DroidSIFT classifies apps based on their API dependency graphs with the goal of classifying Android malware into families as well as detecting zero-day malware. DroidAPIMiner and Drebin classify apps based on features learned from a number of benign and malicious apps during static analysis. However, all of those approaches lack the ability to analyze code in goodware and malware alike that is obfuscated or loaded dynamically at runtime, a prevalent feature as we have discovered in our large-scale study with Andrubis (see Section 3). In contrast, Marvin does not suffer from these limitations. With a lower false positive rate than DroidAPIMiner and Drebin, Marvin also classifies more malware correctly (98.24%) than DroidAPIMiner, which reports a maximum detection rate of 97.8%, and Drebin, which detects only 94% of malware.

The remainder of related work on classifying Android malware based on static features evaluates their approaches on non-representative datasets: The dataset of DroidMat [212] only contains 238 malware samples, PUMA [178] on the other hand, in addition to only learning from requested permissions, is trained and tested on only 249 malicious apps. Finally, Sahs et al. [177] also include features from the control flow graph in addition to permissions. However, they only train on benign samples and use less than 100 malware samples with unknown diversity for verification. Consequently they achieve unusual results: One configuration classifies half of benign apps as malware, while another configuration exhibits a higher false negative than true negative rate.

Detection based on dynamic analysis features. Andromaly [180] is an anomaly detector for Android devices based on on-device monitoring and evaluated different feature selection and machine learning algorithms. However, its evaluation suffers from a lack of available malware samples. STREAM [17] is a framework for evaluating mobile malware classifiers based on the same features as Andromaly with an equally limited testing set of only 50 apps. Additionally, the tested classifiers achieve substantial false positive rates ranging from 14.55% up to 44.36%, rendering them completely impractical.

Crowdroid [39] made a first step towards the use of dynamic analysis results for Android malware detection by performing k-means clustering based on system call invocation counts. Afonso et al. [15] dynamically analyze Android apps to use the number of invocations of API and system calls as coarse-grained features to train various classifiers. Their monitoring approach relies on modifying the app under analysis, which is easily detectable by malware. The only other related approach combining static with dynamic analysis is DroidDolphin [213]. Again, the approach relies on repackaging an app with monitoring code. While the authors observed that the accuracy increased with the size of the training set, DroidDolphin achieves an accuracy of only 86.1% in the best case.

Real-world classification. Related work recently studied the importance of constructing representative datasets and evaluation strategies to ensure the real-world applicability of machine learning malware classification approaches [16, 172]. However, the majority of approaches related to MARVIN learns from very small datasets, often comprising of only a few hundred apps, and an even more limited and less diverse set of malicious apps. Furthermore, related work failed to investigate the long-term practicality of machine learning techniques for Android malware classification. These practices limit the ability of related approaches to generalize in a real-world setting and their robustness when faced with changes in the Android app landscape. One of the major developments that we observed with ANDRUBIS is the increasing importance of dynamic analysis to completely capture an app's characteristic features, in turn, rendering many existing approaches solely relying on static analysis obsolete. Apps can either use reflection and code obfuscation to hinder static analysis [142] or bypass static approaches entirely by dynamically loading code at runtime. This code can either be packaged with the app itself, possibly encrypted and even hidden in an innocuous looking image [19], or downloaded from external sources, making static analysis of the complete app impossible.

Such approaches are not only being used by malicious apps, instead, benign apps use dynamic code loading, reflection, and obfuscation alike for application upgrades, statistical A/B testing, premium features (e.g., features purchased in-app), and/or copyright protection.

Most of the existing classification approaches solely rely on static features for classification, thus missing characteristic features in almost 30% of current apps that dynamically load code at runtime [123]. Conversely, approaches relying only on dynamic analysis can be defeated by apps detecting and evading the analysis environment [158, 204]. By combining results from static and dynamic analysis Marvin can automatically identify common features of Android malware and is more accurate and more robust to evasion than prior work. Furthermore, it provides a fine-grained distinction between goodware and malware in the form of a malice score, which is beneficial to both novices and experts alike.

6.4 Android Application Markets

Many generic measurements of Android application markets have been conducted. Related work on studying malware distribution in markets has mainly focused on measuring the amount of repackaged apps in alternative markets. While monitoring and crawling single, smaller market, or even crawling the whole Google Play Store once, is feasible, continuously tracking all apps in a market is more challenging. Furthermore, the size and plethora of alternative app markets—a current report estimates 200 app markets in China alone [215]—makes crawling multiple (all) markets exhaustively intractable. However, as we showed with AndRadar, tracking apps in multiple markets using an efficient app discovery mechanism for different markets gives us a unique vantage point and provides insights about the publishing behavior and development strategies of malware authors.

General market studies. Petsas et al. [157] performed a systematic study of the mobile app ecosystem by monitoring four alternative marketplaces (*appchina*, *anzhi*, *1mobile*, *slideme*), focusing on app popularity distribution and app pricing, download patterns and revenue strategies. PlayDrone [206] crawled the entire Google Play Store, to study its content, the library usage amongst apps, and the effectivity of authentication mechanisms in apps. Carbunar et al. [41] performed a further longitudinal study of the Google Play Store, investigating update frequencies and pricing trends of apps. Those measurements are generic and focused on benign apps; in the following, we discuss work related to AndRadar investigating the distribution of malware in alternative markets.

Discovery of malware in markets. In March 2011, DroidMOSS identified 5–13% of apps found on 6 alternative marketplaces (*slideme*, *freewarelovers*, *eoemarket*, *goapk*, *softportal*, *proandroid*) as repackaged versions of apps obtained from the Google Play Store. At the same time, PiggyApp applied repackaging detection to 84,767 apps collected from 7 markets (*slideme*, *freewarelovers*, *eoemarket*, *goapk*, *softportal*, *proandroid*, Google Play Store), and identified 0.97–2.7% of apps as repackaged versions of other apps. Vidas et al. [203] conducted a large-scale measurement on 194 alternative Android markets (of which the list was not disclosed, to the best of our knowledge) in October 2011, collecting 41,057 apps. Their key finding was that certain markets almost exclusively distribute repackaged apps containing malware. TraskRank [147] evaluated the trustworthiness of alternative markets in China in January 2014. The authors compare the official versions of the 25 most popular apps with versions found in 20 alternative markets as well as the Google Play Store. Their results indicate, that out of a total of 506 evaluated apps, only 26.09% are safe to download. All four approaches require downloading the APKs and processing the manifest and code offline. Consequently, for instance the results of the study by Vidas et al. [203], which is by far the most extensive of the four, suggest that the authors have sampled only an average of 211 apps per market, that is, very few compared to the overall market sizes. With our lightweight market monitoring technique in AndRadar we can monitor even the largest alternative markets such as *lenovo*, containing around 400,000 apps, or the official Google Play Store with around 800,000 apps at the time of our experiments.[1] Since then, app stores have been growing rapidly: The Google Play Store currently lists over 1.7 million apps, reinforcing the need for efficient methods for app discovery.[2]

Another example of applying repackaging and plagiarism detection is AdRob [49,86], which concentrates on detecting ad-aggressive apps based on monitoring the HTTP advertising traffic generated by 265,359 apps obtained from 17 alternative markets. Indeed, repackaging (paid) apps to incorporate advertisement libraries and distribute the resulting apps on alternative markets seems to be a profitable, illicit business. As ad-based revenue models are not considered malware, this work is orthogonal to ours: In our preliminary market characterization as part of AndRadar, described in Section 5.1, we explicitly removed adware samples. Moreover, AdRob depends on a static and dynamic analysis phase, which is more expensive than our lightweight, metadata-based approach.

The authors of DroidRanger [228] conducted a measurement on 5 markets (including the Google Play Store) in May 2011, analyzed 204,040 apps, and determined that 211 apps were exhibiting malicious patterns. In the same fashion, RiskRanker [93], and other tools presented in Section 6.3, try to identify certain behaviors, observed in malware, in a given app and associate a risk with it. All of these approaches focus on finding or inferring malware on markets; we took the idea a step further, and proposed an approach that is fast enough to *track* malware across markets over time.

Finally, several other related approaches have a goal similar to AndRadar's: Finding fast analysis techniques that are scalable enough to match the extensiveness of today's markets. MAST is trained on a small set of benign and malicious apps, from which features such as permissions, intents, or native code information are extracted. Then, it uses multiple correspondence analysis (MCA) to triage new apps for analysis. Similar to MAST, DroidSearch [165] is a search engine for Android apps, which is designed to triage apps based on certain characteristics, such as required permissions or API calls, for further analysis. DroidSearch also includes a market crawler and indexes

[1] `http://www.onepf.org/appstores/` (retrieved February 1, 2014)

[2] `http://www.appbrain.com/stats/number-of-android-apps` (retrieved October 16, 2015)

the metadata of apps downloaded from markets. By crawling 6 different app stores (*fdroid*, *freewarelovers*, *slideme*, *apkhiapk*, *appchina*, Google Play Store) the crawler collected 235,000 apps—again only sampling the markets and not employing any intelligent search algorithm. SherlockDroid [20] crawls the Google Play Store and 4 alternative markets, collecting 120,000 apps during two campaigns in July and October 2014. Different from AndRadar, its main goal is the detection of only zero-day malware in markets, and in contrast to AndRadar the authors do not leverage the markets' metadata in their analysis. Most recently, MassVet [50] proposed an approach for markets to detect malicious submissions based on apps found on other markets, and claims it can vet submissions at scale with a vetting delay of less than 10 seconds per app. The vetting process itself requires static analysis and compares apps based on similarities in their UI structure and their control flow graph. In a large-scale study on 1.2 million apps from 33 app markets, MassVet found 10.93% suspicious apps and also observed apps being republished after disappearing from stores, confirming our observations on malware hopping that we made with AndRadar. However, in contrast to AndRadar, MassVet requires crawling of entire app stores to extract a baseline of apps and to perform static analysis on them.

Above-mentioned related work requires crawling (a subset of) apps from alternative markets and downloading all files to perform expensive static and dynamic analysis, in many cases with modified Android platforms. Contrarily, AndRadar requires only a public market interface to query apps, and is therefore much faster, scalable and lightweight. While we studied 8 alternative markets in their entirety for our preliminary study on the distribution of malware through markets, our intelligent app discovery based on a seed of malicious apps further allows us to track suspicious apps in the Google Play Store and a variety of alternative markets in real-time.

CHAPTER 7

Summary and Future Work

Today, malware is the basis for many forms of cybercrime and has become part of a multi-million dollar industry. Malware authors thus are strongly motivated by financial gains. As a result, malicious code is constantly evolving, trying to exploit new avenues for monetization, and evading current defenses developed by researchers and antivirus companies, leading to a never-ending arms race. Motivated by the need to enhance the current state-of-the-art of malware analysis techniques we proposed several solution to current challenges posed by the evolving malware threat.

We proposed and implemented an automated approach to understand the evolution of malware over time by observing code changes between versions of self-updating malware and augmenting these changes with information about their high-level behavior. Beagle allows analysts to get a bigger picture of the behavior of a malware family, including the most frequently updated functional components, and new functionality being added over time. As a result, analysts can effectively triage malware samples and focus their analysis efforts on the most important portions of code instead of repacked versions of already known samples.

To deal with the emerging threat of Android malware we further built a large-scale analysis sandbox for Android apps, Andrubis, which we operate as a public service. Andrubis allows us to collect a comprehensive dataset of Android apps in the wild, including a large portion of malware. Based on our diverse dataset comprised of over 1 million apps we studied the evolution of Android malware behavior over time from 2010 to 2014, providing insights on the prevalence of dynamic code loading and the importance of dynamic analysis. We then extended Andrubis with Marvin, a machine learning approach to automatically distinguish benign from malicious Android apps. During our evaluation we demonstrated the real-world practicability of our approach and showed that it can be efficiently retrained, without changes in the underlying feature extraction process, to maintain its detection accuracy over time.

One unique aspect distinguishing Android apps from Windows malware are application markets as the main app distribution channel. We performed an in-depth market study on the distribution of malware in alternative app markets, crawling 8 markets in their entirety. Motivated by our findings, we implemented an Android market radar, AndRadar, that allows us to discover and track malicious Android apps in the Google Play Store and numerous alternative app markets in real-time. As a result we gain important insights into publishing habits of malicious app authors and can facilitate fast remediation procedures at the market level.

As part of future work we plan to extend our Android analysis to study app collusion, i.e., malware authors combining the functionality of different benign looking apps running on the same system to implement their malicious payload. While current analysis techniques analyze each app individually, we plan to devise methods to monitor the interaction between multiple apps under analysis to obtain a more complete picture about their behavior. As another part of future work we also want to explore malware analysis techniques on other platforms, and investigate whether we can build largely platform independent malware defenses. We already implemented a prototype for the analysis of Mac OS X malware [121] and studied the prevalence of cross-platform malware families [124]. While there are already several cases of malware that infect Windows, Linux, and Mac OS X, and cases of Windows threats being ported to new platforms, we expect this threat to become more severe in the future. Especially the increasing use of multiple devices such as smartphones and tablets that are interconnected with each other makes them vulnerable to new kinds of attacks. Instead of simply implementing the same functionality in malware across platforms, malware authors can *combine the functionality of software of different platforms* to achieve their goals. Attackers can use one infected device to spread to other connected devices, for example through a piece of malware on a desktop machine installing its mobile counterpart on a mobile device, or an infected mobile device pushing malware to a connected desktop host. A malicious desktop and smartphone app can then work together to, for example, steal two-factor authentication tokens or mobile TANs. Our goal is to develop malware detection and mitigation techniques on different layers that are platform agnostic or easily adaptable to different platforms. Based on these techniques we can then further derive high-level analysis methods to identify shared functionality as well as the authors of malware campaigns across platforms.

Bibliography

[1] Androguard. `https://github.com/androguard/androguard`.

[2] BinDiff. `http://www.zynamics.com/bindiff.html`.

[3] Contagio. `http://contagiominidump.blogspot.com`.

[4] CopperDroid. `http://copperdroid.isg.rhul.ac.uk`.

[5] ForeSafe Mobile Security. `http://www.foresafe.com`.

[6] Joe Sandbox Mobile. `http://www.joesecurity.org`.

[7] Mobile Sandbox. `http://mobilesandbox.org`.

[8] SandDroid. `http://sanddroid.xjtu.edu.cn`.

[9] Tracedroid. `http://tracedroid.few.vu.nl`.

[10] VirusShare. `http://www.virusshare.com`.

[11] VirusTotal. `http://www.virustotal.com`.

[12] VisualThreat. `http://www.visualthreat.com`.

[13] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *Proceedings of the 9th International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2013.

[14] Douglas Adams. *The Hitchhiker's Guide to the Galaxy*. Del Rey, 1979.

[15] Vitor Monte Afonso, Matheus Favero de Amorim, André Ricardo Abed Grégio, Glauco Barroso Junquera, and Paulo Lício de Geus. Identifying Android Malware Using Dynamically Obtained Features. *Journal of Computer Virology and Hacking Techniques*, 11(1):9–17, 2014.

[16] Kevin Allix, Tegawende Bissyande, Jacques Klein, and Yves Le Traon. Are Your Training Datasets Yet Relevant? - An Investigation into the Importance of Timeline in Machine Learning-Based Malware Detection. In *Proceedings of the 7th International Symposium on Engineering Secure Software and Systems (ESSoS)*, 2015.

[17] Brandon Amos, Hamilton A. Turner, and Jules White. Applying Machine Learning Classifiers to Dynamic Android Malware Detection at Scale. In *Proceedings of the 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2013.

[18] Manos Antonakakis, Roberto Perdisci, Yacin Nadji, Nikolaos Vasiloglou, Saeed Abu-Nimeh, Wenke Lee, and David Dagon. From Throw-away Traffic to Bots: Detecting the Rise of DGA-based Malware. In *Proceedings of the 21st USENIX Security Symposium (SEC)*, 2012.

[19] Axelle Aprville and Ange Albertini. Hide Android Applications in Images. Black Hat Europe, 2014.

[20] Axelle Apvrille and Ludovic Apvrille. SherlockDroid: A Research Assistant to Spot Unknown Malware in Android Marketplaces. *Journal of Computer Virology and Hacking Techniques*, 11(39):1–11, 2015.

[21] Axelle Apvrille and Tim Strazzere. Reducing the Window of Opportunity for Android Malware: Gotta catch 'em all. *Journal in Computer Virology*, 8(1-2):61–71, 2012.

[22] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[23] Irfan Asrar. Android Apps Get Hit with the Evil Twin Routine Part 2: Play It Again Spam. `http://www.symantec.com/connect/blogs/android-apps-get-hit-evil-twin-routine-part-2-play-it-again-spam`, July 2012.

[24] Irfan Asrar. Android.Dropdialer Identified on Google Play. `http://www.symantec.com/connect/blogs/androiddropdialer-identified-google-play`, July 2012.

[25] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[26] Stefan Axelsson. The Base-rate Fallacy and Its Implications for the Difficulty of Intrusion Detection. In *Proceedings of the 6th ACM Conference on Computer and Communications Security (CCS)*, 1999.

[27] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario. Automated Classification and Analysis of Internet Malware. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[28] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010.

[29] Thomas Barabosch, Adrian Dombeck, Khaled Yakdan, and Elmar Gerhards-Padilla. BotWatcher: Transparent and Generic Botnet Tracking. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2015.

[30] David Barrera, Jeremy Clark, Daniel McCarney, and Paul C. van Oorschot. Understanding and Improving App Installation Security Mechanisms Through Empirical Analysis of Android. In *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2012.

[31] Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. A View on Current Malware Behaviors. In *Proceedings of the 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[32] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. TTAnalyze: A Tool for Analyzing Malware. In *Proceedings of the 15th European Institute for Computer Antivirus Research (EICAR) Annual Conference*, 2006.

[33] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauscheck, Christopher Kruegel, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.

[34] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic Analysis of Malicious Code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[35] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2005.

[36] Michael Bierma, Eric Gustafson, Jeremy Erickson, David Fritz, and Yung Ryn Choe. Andlantis: Large-scale Android Dynamic Analysis. In *Proceedings of the 3rd IEEE Mobile Security Technologies Workshop (MoST)*, 2014.

[37] Thomas Bläsing, Aubrey-Derrick Schmidt, Leonid Batyuk, Seyit A. Camtepe, and Sahin Albayrak. An Android Application Sandbox System for Suspicious Software Detection. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE)*, 2010.

[38] Martial Bourquin, Andy King, and Edward Robbins. BinSlayer: Accurate Comparison of Binary Executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)*, 2013.

[39] 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 (SPSM)*, 2011.

[40] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring Pay-per-install: The Commoditization of Malware Distribution. In *Proceedings of the 20th USENIX Security Symposium (SEC)*, 2011.

[41] Bogdan Carbunar and Rahul Potharaju. A Longitudinal Study of the Google App Market. In *Proceedings of the IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2015.

[42] Lewis Carroll. *Through the Looking Glass And What Alice Found There*. Rand, McNally, 1917.

[43] Patrick Carter, Collin Mulliner, Martina Lindorfer, William Robertson, and Engin Kirda. CuriousDroid: Automated User Interface Interaction for Android Application Analysis Sandboxes. In *Proceedings of the 20th International Conference on Financial Cryptography and Data Security (FC)*, 2016 (to appear).

[44] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. Anti-Taint-Analysis: Practical Evasion Techniques Against Information Flow Based Malware Defense. Technical report, Secure Systems Lab at Stony Brook University, 2007.

[45] Silvio Cesare and Yang Xiang. *Software Similarity and Classification*. Springer-Verlag, 2012.

[46] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. MAST: Triage for Market-scale Mobile Malware Analysis. In *Proceedings of the 6th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2013.

[47] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A Library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27:1–27:27, 2011.

[48] Victor Chebyshev. Mobile attacks! `http://www.securelist.com/en/blog/805/Mobile_attacks`, February 2013.

[49] Hao Chen. Underground Economy of Android Application Plagiarism. In *Proceedings of the 1st International Workshop on Security in Embedded Systems and Smartphones (SESP)*, 2013.

[50] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. In *Proceedings of the 24th USENIX Security Symposium (SEC)*, 2015.

[51] Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In *Proceedings of the 38th Annual IEEE International Conference on Dependable Systems and Networks (DSN)*, 2008.

[52] Yi-Wei Chen and Chih-Jen Lin. Combining SVMs with Various Feature Selection Strategies. In *Feature Extraction, Foundations and Applications*. Springer, 2006.

[53] Jerry Cheng, Starsky H.Y. Wong, Hao Yang, and Songwu Lu. SmartSiren: Virus Detection and Alert for Smartphones. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2007.

[54] Pern Hui Chia, Yusuke Yamamoto, and N. Asokan. Is This App Safe? A Large Scale Study on Application Permissions and Risk Signals. In *Proceedings of the 21st International Conference on World Wide Web (WWW)*, 2012.

[55] Mihai Christodorescu, Christopher Kruegel, and Somesh Jha. Mining Specifications of Malicious Behavior. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2007.

[56] Filip Chytry. Google Play: Whats the newest threat on the official Android market? `http://blog.avast.com/2014/03/07/google-play-whats-the-newest-threat-on-the-official-android-market`, March 2014.

[57] Graham Cluley. Android malware poses as Angry Birds Space game. `http://nakedsecurity.sophos.com/2012/04/12/android-malware-angry-birds-space-game`, April 2012.

[58] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *Proceedings of the 17th European Symposium on Research in Computer Security (ESORICS)*, 2012.

[59] Jonathan Crussell, Clint Gibler, and Hao Chen. AnDarwin: Scalable Semantics-Based Detection of Similar Android Applications. In *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS)*, 2013.

[60] Matteo Danelli. ADMIRE: Android Developers & Marketplaces Intelligence and Reputation Engine. Master's thesis, Politecnico di Milano, 2015.

[61] Charles Darwin. *The Voyage of the Beagle*. P.F. Collier, 1909.

[62] Anthony Desnos and Geoffroy Gueguen. Android: From Reversing To Decompilation. In *Black Hat Abu Dhabi*, 2011.

[63] Artem Dinaburg, Paul Royal, Monirul I. Sharif, and Wenke Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, 2008.

[64] Paul Ducklin. Anatomy of a file format problem - yet another code verification bypass in Android. http://nakedsecurity.sophos.com/2013/11/06/anatomy-of-a-file-format-problem-yet-another-code-verification-bypass-in-android, November 2013.

[65] Paul Ducklin. Anatomy of another Android hole - Chinese researchers claim new code verification bypass. http://nakedsecurity.sophos.com/2013/07/17/anatomy-of-another-android-hole-chinese-researchers-claim-new-code-verification-bypass, July 2013.

[66] Paul Ducklin. Flappy Bird really *is* dead - beware of infected fakes that promise to keep him alive! http://nakedsecurity.sophos.com/2014/02/11/flappy-bird-really-is-dead-beware-of-infected-fakes-that-promise-to-keep-him-alive, 2014.

[67] Thomas Dullien and Rolf Rolles. Graph-Based Comparison of Executable Objects. In *Proceedings of the Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, 2005.

[68] Thomas Eder, Michael Rodler, Dieter Vymazal, and Markus Zeilinger. ANANAS - A Framework For Analyzing Android Applications. In *Proceedings on the 1st International Workshop on Emerging Cyberthreats and Countermeasures (ECTCM)*, 2013.

[69] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A Survey on Automated Dynamic Malware Analysis Techniques and Tools. *ACM Computing Surveys Journal*, 44(2):6:1–6:42, 2012.

[70] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Security Symposium (SEC)*, 2014.

[71] William Enck, Peter Gilbert, 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. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[72] 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 (CCS)*, 2009.

[73] Michael Endler. Does Mobile Antivirus Software Really Protect Smartphones? http://www.informationweek.com/security/antivirus/does-mobile-antivirus-software-really-pr/240008673, October 2012.

[74] F-Secure. Mobile Threat Report Q2 2012. http://www.f-secure.com/weblog/archives/MobileThreatReport_Q2_2012.pdf, August 2012.

[75] F-Secure. Android Hack-Tool Steals PC Info. http://www.f-secure.com/weblog/archives/00002573.html, July 2013.

[76] F-Secure. Threat Report H2 2013. http://www.f-secure.com/static/doc/labs_global/Research/Threat_Report_H2_2013.pdf, March 2014.

[77] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A Library for Large Linear Classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[78] 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 (CCS)*, 2011.

[79] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A Survey of Mobile Malware in the Wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[80] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the 8th Symposium on Usable Privacy and Security (SOUPS)*, 2012.

[81] Halvar Flake. Structural Comparison of Executable Objects. In *Proceedings of the 1st Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2004.

[82] Jeff Forristal. Android: One Root to Own Them All. In *Black Hat USA*, 2013.

[83] Thomas Fox-Brewster. Cops Knock Down Dridex Malware That Earned 'Evil Corp' Cybercriminals At Least $50 Million. http://www.forbes.com/sites/thomasbrewster/2015/10/13/dridex-botnet-takedown, October 2015.

[84] Jason Franklin, Vern Paxson, Adrian Perrig, and Stefan Savage. An Inquiry into the Nature and Causes of the Wealth of Internet Miscreants. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, 2007.

[85] Debin Gao, Michael K. Reiter, and Dawn Song. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Proceedings of the 10th International Conference on Information and Communications Security (ICICS)*, 2008.

[86] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. AdRob: Examining the Landscape and Impact of Android Application Plagiarism. In *Proceedings of 11th International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2013.

[87] Jan Goebel, Thorsten Holz, and Carsten Willems. Measurement and Analysis of Autonomous Spreading Malware in a University Environment. In *Proceedings of the 4th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2007.

[88] Google. ART and Dalvik. `https://source.android.com/devices/tech/dalvik/index.html`.

[89] Google. Safe Browsing API. `https://developers.google.com/safe-browsing`.

[90] Google. Signing Your Applications. `https://developer.android.com/tools/publishing/app-signing.html`.

[91] Google. UI/Application Exerciser Monkey. `https://developer.android.com/tools/help/monkey.html`.

[92] Michael Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2012.

[93] 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 (MobiSys)*, 2012.

[94] Mariano Graziano, Davide Canali, Leyla Bilge, Andrea Lanzi, and Davide Balzarotti. Needles in a Haystack: Mining Information from Public Dynamic Analysis Sandboxes for Malware Intelligence. In *Proceedings of the 24th USENIX Security Symposium (SEC)*, 2015.

[95] Chris Grier, Lucas Ballard, Juan Caballero, Neha Chachra, Christian J. Dietrich, Kirill Levchenko, Panayiotis Mavrommatis, Damon McCoy, Antonio Nappa, Andreas Pitsillidis, Niels Provos, M. Zubair Rafique, Moheeb Abu Rajab, Christian Rossow, Kurt Thomas, Vern Paxson, Stefan Savage, and Geoffrey M. Voelker. Manufacturing Compromise: The Emergence of Exploit-as-a-service. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[96] Lion Gu. The Mobile Cybercriminal Underground Market in China. Technical report, Trend Micro, March 2014. `http://www.trendmicro.com/cloud-`

content/us/pdfs/security-intelligence/white-papers/wp-the-mobile-cybercriminal-underground-market-in-china.pdf.

[97] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2012.

[98] Matthew Hayes, Andrew Walenstein, and Arun Lakhotia. Evaluation of Malware Phylogeny Modelling Systems Using Automated Variant Generation. *Journal in Computer Virology*, 5(4):335–343, 2009.

[99] Francis Heylighen. The Red Queen Principle. http://pespmc1.vub.ac.be/redqueen.html, 1993.

[100] Summer Hirst. Lookout Discovers SocialPath Malware in Google Play Store. https://vpncreative.net/2015/01/10/lookout-socialpath-malware-google-play, January 2015.

[101] IDC. Smartphone OS Market Share 2015, 2014, 2013, and 2012. http://www.idc.com/prodserv/smartphone-os-market-share.jsp, August 2015.

[102] Bernadette Irinco. First Android Trojan in the Wild. http://blog.trendmicro.com/trendlabs-security-intelligence/first-android-trojan-in-the-wild, August 2010.

[103] Paul Jaccard. The Distribution of Flora in the Alpine Zone. *The New Phytologist*, 11(2):37–50, 1912.

[104] Gregoire Jacob, Herve Debar, and Eric Filiol. Malware Behavioral Detection by Attribute-Automata using Abstraction from Platform and Language. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2009.

[105] Gregoire Jacob, Ralf Hund, Christopher Kruegel, and Thorsten Holz. Jackstraws: Picking Command and Control Connections from Bot Traffic. In *Proceedings of the 20th USENIX Security Symposium (SEC)*, 2011.

[106] 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 (CCS)*, 2011.

[107] Jiyong Jang, Maverick Woo, and David Brumley. Towards Automatic Software Lineage Inference. In *Proceedings of the 22nd USENIX Security Symposium (SEC)*, 2013.

[108] Chris Jarabek, David Barrera, and John Aycock. ThinAV: Truly Lightweight Mobile Cloud-based Anti-malware. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, 2012.

[109] Xuxian Jiang. An Evaluation of the Application ("App") Verification Service in Android 4.2. `http://www.cs.ncsu.edu/faculty/jiang/appverify`, December 2012.

[110] Weiwei Jin, Sagar Chaki, Cory Cohen, Arie Gurfinkel, Jeffrey Havrilla, Charles Hines, and Priya Narasimhan. Binary Function Clustering Using Semantic Hashes. In *Proceedings of the 11th International Conference on Machine Learning and Applications (ICMLA)*, 2012.

[111] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P)*, 2011.

[112] Juniper Networks. Juniper Networks Third Annual Mobile Threats Report. `http://www.juniper.net/us/en/local/pdf/additional-resources/jnpr-2012-mobile-threats-report.pdf`, June 2013.

[113] Md.Enamul. Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. Malware Phylogeny Generation Using Permutations of Code. *Journal in Computer Virology*, 1(1):13–23, 2005.

[114] Brian Krebs. Criminal Classifieds: Malware Writers Wanted. `https://krebsonsecurity.com/2011/06/criminal-classifieds-malware-writers-wanted`, June 2011.

[115] Brian Krebs. Banking on Badb in the Underweb. `https://krebsonsecurity.com/2012/03/banking-on-badb-in-the-underweb`, March 2012.

[116] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005.

[117] Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. Fast Location of Similar Code Fragments Using Semantic 'Juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)*, 2013.

[118] Charles Lever, Manos Antonakakis, Bradley Reaves, Patrick Traynor, and Wenke Lee. The Core of the Matter: Analyzing Malicious Traffic in Cellular Carriers. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, 2013.

[119] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. Lines of Malicious Code: Insights Into the Malicious Software Industry. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, 2012.

[120] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting Environment-Sensitive Malware. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[121] Martina Lindorfer, Bernhard Miller, Matthias Neugschwandtner, and Christian Platzer. Take a Bite - Finding the Worm in the Apple. In *Proceedings of the 9th International Conference on Information, Communications and Signal Processing (ICICS)*, 2013.

[122] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer. Marvin: Efficient and Comprehensive Mobile App Classification Through Static and Dynamic Analysis. In *Proceedings of the 39th Annual International Computers, Software & Applications Conference (COMPSAC)*, 2015.

[123] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proceedings of the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[124] Martina Lindorfer, Matthias Neumayr, Juan Caballero, and Christian Platzer. POSTER: Cross-Platform Malware: Write Once, Infect Everywhere. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[125] Martina Lindorfer, Stamatis Volanis, Alessandro Sisto, Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, and Sotiris Ioannidis. AndRadar: Fast Discovery of Android Applications in Alternative Markets. In *Proceedings of the 11th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2014.

[126] Robert Lipovsky. ESET Analyzes First Android File-Encrypting, TOR-enabled Ransomware. `http://www.welivesecurity.com/2014/06/04/simplocker`, June 2014.

[127] Flora Liu. Windows Malware Attempts to Infect Android Devices. `http://www.symantec.com/connect/blogs/windows-malware-attempts-infect-android-devices`, January 2014.

[128] Hiroshi Lockheimer. Android and Security. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`, February 2012.

[129] Lookout Mobile Security. Update: RuFraud: European Premium SMS Toll Fraud on the Rise. https://blog.lookout.com/blog/2011/12/11/european-premium-sms-fraud, December 2011.

[130] Lookout Mobile Security. State of Mobile Security 2012. https://www.lookout.com/_downloads/lookout-state-of-mobile-security-2012.pdf, September 2012.

[131] Adrian Ludwig, Eric Davis, and Jon Larimer. Android - Practical Security From the Ground Up. In *Virus Bulletin Conference*, 2013.

[132] Federico Maggi, Andrea Bellini, Guido Salvaneschi, and Stefano Zanero. Finding Non-trivial Malware Naming Inconsistencies. In *Proceedings of the 7th International Conference on Information Systems Security (ICISS)*, 2011.

[133] Federico Maggi, Andrea Valdi, and Stefano Zanero. AndroTotal: A Flexible, Scalable Toolbox and Service for Testing Mobile Malware Detectors. In *Proceedings of the 3rd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2013.

[134] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. Analysis of the Communication Between Colluding Applications on Modern Smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, 2012.

[135] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[136] Lorenzo Martignoni, Elizabeth Stinson, Matt Fredrikson, Somesh Jha, and John C. Mitchell. A Layered Architecture for Detecting Malicious Behaviors. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[137] Felix Matenaar and Patrick Schulz. Detecting Android Sandboxes. http://www.dexlabs.org/blog/btdetect, August 2012.

[138] McAfee Labs. McAfee Threats Report: Second Quarter 2013. http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q2-2013.pdf, August 2013.

[139] Paolo Milani Comparetti, Guido Salvaneschi, Engin Kirda, Clemens Kolbitsch, Christopher Kruegel, and Stefano Zanero. Identifying Dormant Functionality in Malware Programs. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P)*, 2010.

[140] Jiang Ming, Meng Pan, and Debin Gao. iBinHunt: Binary Hunting with Inter-procedural Control Flow. In *Proceedings of the 15th Annual International Conference on Information Security and Cryptology (ICISC)*, 2012.

[141] Jiang Ming, Dongpeng Xu, and Dinghao Wu. Memoized Semantics-Based Binary Diffing with Application to Malware Lineage Inference. In *Proceedings of the 30th IFIP SEC International Information Security and Privacy Conference (IFIP SEC)*, 2015.

[142] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of Static Analysis for Malware Detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[143] Simone Mutti, Yanick Fratantonio, Antonio Bianchi, Luca Invernizzi, Jacopo Corbetta, Dhilung Kirat, Christopher Kruegel, and Giovanni Vigna. BareDroid: Large-Scale Analysis of Android Apps on Real Devices. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, 2015.

[144] Matthias Neugschwandtner, Martina Lindorfer, and Christian Platzer. A View To A Kill: WebView Exploitation. In *Proceedings of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2013.

[145] Sebastian Neuner, Victor van der Veen, Martina Lindorfer, Markus Huber, Georg Merzdovnik, Martin Mulazzani, and Edgar Weippl. Enter Sandbox: Android Sandbox Comparison. In *Proceedings of the 3rd IEEE Mobile Security Technologies Workshop (MoST)*, 2014.

[146] Andrew Y. Ng. Feature selection, L1 vs. L2 regularization, and rotational invariance. In *Proceedings of the 21st International Conference on Machine Learning (ICML)*, 2004.

[147] Yi Ying Ng, Hucheng Zhou, Zhiyuan Ji, Huan Luo, and Yuan Dong. Which Android App Store Can Be Trusted in China? In *Proceedings of the 38th Annual International Computers, Software & Applications Conference (COMPSAC)*, 2014.

[148] John Oberheide and Charlie Miller. Dissecting the Android Bouncer. In *SummerCon*, 2012.

[149] Jon Oberheide, Kaushik Veeraraghavan, Evan Cooke, Jason Flinn, and Farnam Jahanian. Virtualized In-cloud Security Services for Mobile Devices. In *Proceedings of the First Workshop on Virtualization in Mobile Computing (MobiVirt)*, 2008.

[150] PandaLabs. PandaLabs Annual Report 2011. `http://www.pandasecurity.com/mediacenter/src/uploads/2014/07/Annual-Report-PandaLabs-2011.pdf`, January 2012.

[151] PandaLabs. PandaLabs Report Q1 2015. http://www.pandasecurity.com/mediacenter/src/uploads/2015/05/PandaLabs-Report_Q1-2015.pdf, May 2015.

[152] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[153] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using Probabilistic Generative Models For Ranking Risks of Android Apps. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[154] Nicholas J. Percoco and Sean Schulte. Adventures in Bouncerland. In *Black Hat USA*, 2012.

[155] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. Classification of Packed Executables for Accurate Computer Virus Detection. *Pattern Recognition Letters*, 29(14):1941–1946, 2008.

[156] Bogdan Petrovan. Google is now manually reviewing apps that are submitted to the Play Store! http://www.androidauthority.com/google-now-manually-reviewing-apps-submitted-to-play-store-594879, March 2015.

[157] Thanasis Petsas, Antonis Papadogiannakis, Michalis Polychronakis, Evangelos P. Markatos, and Thomas Karagiannis. Rise of the Planet of the Apps: A Systematic Study of the Mobile App Ecosystem. In *Proceedings of the Internet Measurement Conference (IMC)*, 2013.

[158] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. In *Proceedings of the 7th European Workshop on System Security (EuroSec)*, 2014.

[159] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[160] Mario Polino, Andrea Scorti, Federico Maggi, and Stefano Zanero. Jackdaw: Towards Automatic Reverse Engineering of Large Datasets of Binaries. In *Proceedings of the 12th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2015.

[161] Andrey Polkovnichenko and Alon Boxiner. BrainTest – A New Level of Sophistication in Mobile Malware. http://blog.checkpoint.com/2015/09/21/braintest-a-new-level-of-sophistication-in-mobile-malware, September 2015.

[162] 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 (ACSAC)*, 2010.

[163] Pouik and G0rfi3ld. Similarities for Fun & Profit. *Phrack Magazine*, 14(68), 2012.

[164] Emil Protalinski. A first: Hacked sites with android drive-by download malware. `http://www.zdnet.com/blog/security/a-first-hacked-sites-with-android-drive-by-download-malware/11810`, May 2012.

[165] Siegfried Rasthofer, Steven Arzt, Stephan Huber, Max Kohlhagen, Brian Pfretschner, Eric Bodden, and Philipp Richter. DroidSearch: A Tool for Scaling Android App Triage to Real-World App Stores. In *Proceedings of the IEEE Technically Co-Sponsored Science and Information Conference (SAI)*, 2015.

[166] Vaibhav Rastogi, Yan Chen, and William Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[167] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2013.

[168] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors. In *Proceedings of the 6th European Workshop on System Security (EuroSec)*, 2013.

[169] Raymond Roberts. Malware Development Life Cycle. In *Virus Bulletin Conference*, 2008.

[170] Sanae Rosen, Zhiyun Qian, and Z. Morley Mao. AppProfiler: A Flexible Method of Exposing Privacy-Related Behavior in Android Applications to End Users. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[171] Christian Rossow, Christian J. Dietrich, and Herbert Bos. Large-Scale Analysis of Malware Downloaders. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2012.

[172] Sankardas Roy, Jordan DeLoach, Yuping Li, Nic Herndon, Doina Caragea, Xinming Ou, Venkatesh Prasad Ranganath, Hongmin Li, and Nicolais Guevara. Experimental Study with Real-world Data for Android App Security Analysis using Machine Learning. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, 2015.

[173] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[174] David Ruddock. Google Pushes Major Update To Play Developer Content Policy, Kills Notification Bar Ads For Real This Time, And A Lot More. `http://www.androidpolice.com/2013/08/23/teardown-google-pushes-major-update-to-play-developer-content-policy-kills-notification-bar-ads-for-real-this-time-and-a-lot-more`, September 2013.

[175] Brian Ruttenberg, Craig Miles, Lee Kellogg, Vivek Notani, Michael Howard, Charles LeDoux, Arun Lakhotia, and Avi Pfeffer. Identifying Shared Software Components to Support Malware Forensics. In *Proceedings of the 11th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 2014.

[176] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting Code Clones in Binary Executables. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[177] Justin Sahs and Latifur Khan. A Machine Learning Approach to Android Malware Detection. In *Proceedings of the European Intelligence and Security Informatics Conference (EISIC)*, 2012.

[178] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo Garcia Bringas, and Gonzalo Álvarez. PUMA: Permission Usage to Detect Malware in Android. In *Proceedings of the International Joint Conference CISIS'12-ICEUTE'12-SOCO'12 Special Sessions*, 2012.

[179] Mathew J. Schwartz. Zeus Banking Trojan Hits Android Phones. `http://www.informationweek.com/security/mobile/zeus-banking-trojan-hits-android-phones/231001685`, July 2011.

[180] Asaf Shabtai, Uri Kananov, Yuval Elovici, Chanan Glezer, and Yael Weiss. "Andromaly": A Behavioral Malware Detection Framework for Android Devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.

[181] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic Reverse Engineering of Malware Emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P)*, 2009.

[182] Signals and Systems Telecom. The Mobile Device & Network Security Bible: 2013–2020. Technical report, September 2013. `http://www.reportsnreports.com/reports/267722-the-mobile-device-network-security-bible-2013-2020.html`.

[183] Zarah Simon. Adwares. Are they viruses or not? http://androidmalwareresearch.blogspot.gr/2012/07/adwares-are-they-viruses-or-not.html, July 2012.

[184] Chris Smith. Android 4.2 'Verify apps' security feature explained by Google. http://www.androidauthority.com/android-4-2-verify-apps-security-feature-explained-by-google-131514, November 2012.

[185] Liam Spradlin. Google Updates Play Store Developer Policy, Puts The Smack Down On Intrusive Advertising. http://www.androidpolice.com/2012/07/31/google-updates-play-store-developer-policy-puts-the-smack-down-on-intrusive-advertising-say-goodbye-to-airpush-and-its-cohorts, July 2012.

[186] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: Having a Deeper Look into Android Applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC)*, 2013.

[187] StackExchange. How does BinDiff work? http://reverseengineering.stackexchange.com/questions/1475/how-does-bindiff-work, April 2013.

[188] Lukas Stefanko. Android trojan drops in, despite Google's Bouncer. http://www.welivesecurity.com/2015/09/22/android-trojan-drops-in-despite-googles-bouncer, September 2015.

[189] Brett Stone-Gross, Ryan Abman, Richard Kemmerer, Christopher Kruegel, Douglas Steigerwald, and Giovanni Vigna. The Underground Economy of Fake Antivirus Software. In *Proceedings of the 10th Workshop on Economics of Information Security (WEIS)*, 2011.

[190] Brett Stone-Gross, Thorsten Holz, Gianluca Stringhini, and Giovanni Vigna. The Underground Economy of Spam: A Botmaster's Perspective of Coordinating Large-Scale Spam Campaigns. In *Proceedings of the 4th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2011.

[191] Tim Strazzere. Downloading market applications without the vending app. http://www.strazzere.com/blog/2009/09/downloading-market-applications-without-the-vending-app, September 2009.

[192] Vanja Svajcer. Sophos Mobile Security Threat Report. http://www.sophos.com/en-us/medialibrary/PDFs/other/sophos-mobile-security-threat-report.pdf, February 2014.

[193] Symantec. Symantec Report on Rogue Security Software. http://eval.symantec.com/mktginfo/enterprise/white_papers/b-

symc_report_on_rogue_security_software_WP_20100385.en-us.pdf, October 2009.

[194] Kurt Thomas and David M. Nicol. The Koobface Botnet and the Rise of Social Malware. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE)*, 2010.

[195] Cody Toombs. External Blues: Google Has Brought Big Changes To SD Cards In KitKat, And Even Samsung Is Implementing Them. http://www.androidpolice.com/2014/02/17/external-blues-google-has-brought-big-changes-to-sd-cards-in-kitkat-and-even-samsung-may-be-implementing-them, February 2014.

[196] Cody Toombs. Updates To AOSP Confirm Dalvik Runtime Will Be Removed From Android, ART Officially Takes Its Place. http://www.androidpolice.com/2014/06/19/updates-aosp-confirm-dalvik-runtime-will-removed-android-art-officially-takes-place, June 2014.

[197] Trend Micro. TrendLabs 2Q 2013 Security Roundup. http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt-2q-2013-trendlabs-security-roundup.pdf, August 2013.

[198] Hien Thi Thu Truong, Eemil Lagerspetz, Petteri Nurmi, Adam J. Oliner, Sasu Tarkoma, N. Asokan, and Sourav Bhattacharya. The Company You Keep: Mobile Malware Infection Rates and Inexpensive Risk Indicators. In *Proceedings of the 23rd International Conference on World Wide Web (WWW)*, 2014.

[199] Xabier Ugarte Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[200] Bartlomiej Uscilowski. Mobile Adware and Malware Analysis. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/madware_and_malware_analysis.pdf, October 2013.

[201] Victor van der Veen. Dynamic Analysis of Android Malware. Master's thesis, VU University Amsterdam, 2013.

[202] Leigh van Valen. A New Evolutionary Law. *Evolutionary Theory*, 1:1–30, 1973.

[203] Timothy Vidas and Nicolas Christin. Sweetening Android Lemon Markets: Measuring and Combating Malware in Application Marketplaces. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[204] 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 (ASIACCS)*, 2014.

[205] Timothy Vidas, Jiaqi Tan, Jay Nahata, Chaur Lih Tan, Nicolas Christin, and Patrick Tague. A5: Automated Analysis of Adversarial Android Applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM)*, 2014.

[206] Nicolas Viennot, Edward Garcia, and Jason Nieh. A Measurement Study of Google Play. In *Proceedings of the 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2014.

[207] Nedim Šrndić and Pavel Laskov. Practical Evasion of a Learning-Based Classifier: A Case Study. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[208] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. WuKong: A Scalable and Accurate Two-phase Approach to Android App Clone Detection. In *Proceedings of the 24th International Symposium on Software Testing and Analysis (ISSTA)*, 2015.

[209] Yi-Min Wang, Roussi Roussev, Chad Verbowski, Aaron Johnson, Ming-Wei Wu, Yennun Huang, and Sy-Yen Kuo. Gatekeeper: Monitoring Auto-Start Extensibility Points (ASEPs) for Spyware Management. In *Proceedings of the 18th USENIX Conference on System Administration (LISA)*, 2004.

[210] Lukas Weichselbaum, Matthias Neugschwandtner, Martina Lindorfer, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis: Android Malware Under The Magnifying Glass. Technical Report TR-ISECLAB-0414-001, Vienna University of Technology, 2014.

[211] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.

[212] 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 *Proceedings of the 7th Asia Joint Conference on Information Security (Asia JCIS)*, 2012.

[213] Wen-Chieh Wu and Shih-Hao Hung. DroidDolphin: A Dynamic Android Malware Detection Framework Using Big Data and Machine Learning. In *Proceeding of the Conference on Research in Adaptive and Convergent Systems (RACS)*, 2014.

[214] Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Security Symposium (SEC)*, 2012.

[215] Eva Yoo. The Top Ten Android App Stores In China 2015. http://technode.com/2015/09/22/ten-best-android-app-stores-china, September 2015.

[216] Kim Zetter. A Google Site Meant to Protect You Is Helping Hackers Attack You. http://www.wired.com/2014/09/how-hackers-use-virustotal, September 2014.

[217] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. ViewDroid: Towards Obfuscation-Resilient Mobile Application Repackaging Detection. In *Proceedings of the 7th ACM Conference on Security and Privacy in Wireless &Mobile Networks (WiSec)*, 2014.

[218] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, 2014.

[219] Yulong Zhang, Hui Xue, and Tao Wei. Occupy Your Icons Silently on Android. http://www.fireeye.com/blog/technical/2014/04/occupy_your_icons_silently_on_android.html, April 2014.

[220] Yulong Zhang, Hui Xue, Tao Wei, and Dawn Song. Ad Vulna: A Vulnaggressive (Vulnerable & Aggressive) Adware Threatening Millions. http://www.fireeye.com/blog/technical/2013/10/ad-vulna-a-vulnaggressive-vulnerable-aggressive-adware-threatening-millions.html, October 2013.

[221] Yulong Zhang, Hui Xue, Tao Wei, and Dawn Song. Monitoring Vulnaggressive Apps on Google Play. http://www.fireeye.com/blog/technical/2013/11/monitoring-vulnaggressive-apps-on-google-play.html, November 2013.

[222] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, and Wei Zou. SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2012.

[223] Min Zheng, Patrick P.C. Lee, and John C.S. Lui. ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-Virus Systems. In *Proceedings of the 10th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2013.

[224] Min Zheng, Mingshen Sun, and John C.S. Lui. DroidRay: A Security Evaluation System for Customized Android Firmwares. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014.

[225] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, Scalable Detection of "Piggybacked" Mobile Applications. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[226] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2012.

[227] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, 2012.

[228] 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 *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, 2012.

[229] Hengshu Zhu, Hui Xiong, Yong Ge, and Enhong Chen. Mobile App Recommendations with Security and Privacy Awareness. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2014.

[230] Jianwei Zhuge, Thorsten Holz, Chengyu Song, Jinpeng Guo, Xinhui Han, and Wei Zou. Studying Malicious Websites and the Underground Economy on the Chinese Web. In *Proceedings of the 7th Workshop on the Economics of Information Security (WEIS)*, 2008.

Curriculum Vitae

Education

Doctor of Technical Sciences (Dr. techn.) July 2011 – November 2015
Vienna University of Technology, Austria
Secure Systems Lab

Master of Science (Dipl.-Ing.) October 2006 – May 2011
Vienna University of Technology, Austria
Major: Software Engineering and Internet Computing

Bachelor of Science (BSc) October 2003 – July 2006
University of Applied Sciences Hagenberg, Austria
Major: Computer and Media Security

Commercial High School October 1998 – June 2003
Bundeshandelsakademie Linz-Auhof, Austria

Work Experience

Researcher since January 2013
SBA Research, Vienna, Austria

Visiting Researcher October 2014 – March 2015
Systems Security Lab, Northeastern University, Boston, USA

Research Assistant July 2011 – September 2014
Secure Systems Lab, Vienna University of Technology, Vienna, Austria

Research Intern November 2013
Foundation for Research and Technology – Hellas (FORTH-ICS), Heraklion, Greece

Research Intern January – May 2013
Fukuda Laboratory, National Institute of Informatics (NII), Tokyo, Japan

Research Intern September – November 2009
Multimedia Security Lab, Korea University, Seoul, South Korea

Software Engineer & Project Controller July 2006 – December 2008
Underground_8 Secure Computing GmbH, Linz, Austria

Software Engineering Intern March – June 2006
Utimaco Safeware AG, Linz, Austria

Program Committee Service

Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)	2016
USENIX Workshop on Offensive Technologies (WOOT)	2015
International Workshop on Emerging Cyberthreats and Countermeasures (ECTCM)	2013–2015

List of Publications

Conferences

CuriousDroid: Automated User Interface Interaction for Android Application Analysis Sandboxes
Patrick Carter, Collin Mulliner, Martina Lindorfer, William Robertson, Engin Kirda
International Conference on Financial Cryptography and Data Security (FC)
Christ Church, Barbados, February 2016 (to appear)

Marvin: Efficient and Comprehensive Mobile App Classification Through Static and Dynamic Analysis
Martina Lindorfer, Matthias Neugschwandtner, Christian Platzer
Annual International Computers, Software & Applications Conference (COMPSAC)
Taichung, Taiwan, July 2015

AndRadar: Fast Discovery of Android Applications in Alternative Markets
Martina Lindorfer, Stamatis Volanis, Alessandro Sisto, Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, Sotiris Ioannidis
Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)
London, UK, July 2014

Take a Bite – Finding the Worm in the Apple
Martina Lindorfer, Bernhard Miller, Matthias Neugschwandtner, Christian Platzer
International Conference on Information, Communications and Signal Processing (ICICS)
Tainan, Taiwan, December 2013

Lines of Malicious Code: Insights Into the Malicious Software Industry
Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, Stefano Zanero
Annual Computer Security Applications Conference (ACSAC)
Orlando, Florida, USA, December 2012

Detecting Environment-Sensitive Malware
Martina Lindorfer, Clemens Kolbitsch, Paolo Milani Comparetti
International Symposium on Recent Advances in Intrusion Detection (RAID)
Menlo Park, California, USA, September 2011

Workshops

Andrubis – 1,000,000 Apps Later: A View on Current Android Malware Behaviors
Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, Christian Platzer
International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)
Wroclaw, Poland, September 2014

Skin Sheriff: A Machine Learning Solution for Detecting Explicit Images
Christian Platzer, Martin Stuetz, Martina Lindorfer
International Workshop on Security and Forensics in Communication Systems (SFCS)
Kyoto, Japan, June 2014

Enter Sandbox: Android Sandbox Comparison
Sebastian Neuner, Victor van der Veen, Martina Lindorfer, Markus Huber, Georg Merzdovnik, Martin Mulazzani, Edgar Weippl
IEEE Mobile Security Technologies Workshop (MoST)
San Jose, California, USA, May 2014

A View to a Kill: WebView Exploitation
Matthias Neugschwandtner, Martina Lindorfer, Christian Platzer
USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)
Washington, D.C., USA, August 2013

Miscellaneous

ReCon: Revealing and Controlling Privacy Leaks in Mobile Network Traffic
Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, David Choffnes
Technical Report, Northeastern University
October 2015

Andrubis: Android Malware Under the Magnifying Glass
Lukas Weichselbaum, Matthias Neugschwandtner, Martina Lindorfer, Yanick Fratantonio, Victor van der Veen, Christian Platzer
Technical Report, Vienna University of Technology
April 2014

POSTER: Cross-Platform Malware: Write Once, Infect Everywhere
Martina Lindorfer, Matthias Neumayr, Juan Caballero, Christian Platzer
ACM Conference on Computer and Communications Security (CCS)
Berlin, Germany, November 2013

The Red Book: A Roadmap for Systems Security Research
The SysSec Consortium, Evangelos Markatos and Davide Balzarotti (editors)
August 2013