TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

DISSERTATION

Protecting Web Clients from Internet Threats

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften

unter der Leitung von

Privatdozent Dipl.-Ing. Dr.techn. Christopher Krügel
und
Privatdozent Dipl.-Ing. Dr.techn. Engin Kirda

Institut für Rechnergestützte Automation
Arbeitsgruppe Automatisierungssysteme (E183-1)

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl.-Ing. Manuel Egele
Kernstockstraße 26
A-6850 Dornbirn
0025546

Wien, am 13. Dezember 2010

Kurzfassung (German Abstract)

Das Internet hat sich in den letzten Jahren immer mehr zu einem Netzwerk entwickelt in dem die Nutzer die Möglichkeit haben unterschiedliche Dienste in Anspruch zu nehmen. Beispielsweise ist es für Nutzer heute sehr einfach möglich soziale Kontakte zu pflegen oder Produkte zu vermarkten und zu kaufen. Online-Banking ist nur ein weiterer Aspekt der die Kommerzialisierung des Internet verdeutlicht.

Wie auch im der realen Welt, zieht das Vorhandensein von Geld immer auch Personen mit bösartigen Intentionen an, die versuchen sich auf Kosten anderer zu bereichern. Bösartige Software (Malware) ist das Mittel der Wahl dem sich solche Personen bedienen. Mittels sogenannter *drive-by Downloads* ist es Cyber-Kriminellen möglich beim Besuch einer Webseite solche Malware auf dem Rechner eines Nutzers unbemerkt zu installieren. Desweiteren machen sich die Angreifer eine Technik namens *web-spamming* zunutze das es ihnen erlaubt, die Suchergebnisse von Internet Suchmaschinen in ihrem Interesse zu verfälschen, um damit mehr Besucher auf ihre Webseiten zu locken. Auf Grund dieses Vorgehens, widmet sich diese Dissertation zu Beginn den beiden Techniken (drive-by Downloads und web-spam) und stellt geeignete Schutzmechanismen vor.

Sollte die installierte Malware bisweilen unbekannt sein, haben Anti-Virus Hersteller und Forscher vielfältige Möglichkeiten die Malware auf ihre Schadwirkung zu untersuchen. Dynamische Analyse Techniken führen dazu die Malware in einer kontrollierten Umgebung aus und beobachten gleichzeitig das Verhalten der Malware. Statische Techniken hingegen erledigen die Analyse ohne den bösartigen Code auszuführen. Im Bereich der statischen Analyse stellt diese Dissertation daher ein neuartiges System vor das in der Lage ist iOS Programme auf Code zu untersuchen der die Privatsphäre des Nutzers untergräbt.

Abstract

The Internet has evolved from a basic communication network to an interconnected set of services enabling, among other things, new forms of (social) interactions and market places for the sale of products and services. Furthermore, online banking or advertising are mere examples of the commercial aspects of the Internet.

Just as in the physical world, whenever money is involved, there are people on the Internet with malevolent intentions that strive to enrich themselves by taking advantage of legitimate users. Malware (i.e., software of malicious intent) helps these people to accomplish their goals. Commonly, the business model of such entities begins with a successful attack on an Internet-connected personal computer, where the attacker installs a piece of malware. A *drive-by download* attack refers to a technique where malware is installed on a victims' computer as a result of the mere visit to an attackers' web site. Such attacks are insidious, because commonly the user does not notice that malware gets installed. Additionally, to increase the number of visitors to their malicious sites, attackers can make use of *web-spamming* to manipulate search engine results in their favor. Thus, this thesis initially focuses on *web-spamming* and *drive-by downloads* and presents mitigation approaches for these threats.

In case the installed malware was previously unknown, anti-virus companies and researchers have two possibilities to analyze the potential threat. Dynamic analysis techniques execute the captured sample in a controlled environment and observe the behavior of the program during execution. Static techniques, on the other hand, perform their analysis task without executing the malicious program. This thesis contributes to the field of malware analysis techniques as we present our novel static analysis system for iOS applications. The evaluation of this sys-

tem shows that the employed techniques are suitable to precisely identify code paths that pose a threat to the privacy of the user.

Acknowledgments

Finishing my PhD studies culminating in this thesis would not have been possible without the help and support of a multitude of people. Therefore, I think it is only fair to give my thanks to these people here.

First, my gratitude goes to my advisors, Engin Kirda and Christopher Krügel. They guided me through my studies, and provided me with the necessary feedback that was required to improve my skills.

Furthermore, I would like to thank all my colleagues that I had the honor to work with at the International Secure Systems Lab. Be it in Vienna, Sophia-Antipolis, or Santa Barbara, it was always a great time in terms of work and fun.

Last but not least, I would like to say thank you to my *awesome* family. They provided me with all the support, love, and understanding I could wish for.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today, Internet-connected devices face a plethora of attacks that threaten the integrity of the data they store, and the privacy of their users. Historically, such attacks focus on Internet-connected personal computers. More recently, however, with the growing processing power of mobile devices, such devices also became the target of attacks. Once an attack is successful, the compromised machine is considered to be under the control of the attacker. Commonly, this entails that the machine is made to join a botnet where the machine's resources are misused to distribute spam messages or to contribute to denial of service attacks. At the same time, the attacker can install spyware on the compromised target to surreptitiously steal private information from the user, such as online banking credentials or credit card numbers. Although mobile devices face similar threats, one characteristic distinguishes such devices from regular personal computers. Mobile devices advance more and more to become communication hubs for their users, and store and access a multitude of sensitive information, such as address books, email accounts, text messages, or positional data. Mobile devices not yet face the variety of malware as seen on PCs. However, third party applications with questionable behavior threaten the integrity of the often sensitive data that is stored on these devices.

1.1 Contributions

In this thesis, we present the research we conducted towards the mitigation of the threats that Internet-connected devices and their users face. Our contributions in this endeavor are as follows:

- First, we demonstrate how miscreants perform *web-spamming* attacks to influence search engine ranking algorithms. Being able to manipulate search engine results in their favor, attackers can attract web traffic (i.e., visitors) to their sites and thus increase the chances of successfully attacking innocuous web surfers. Of course, we also present our findings in trying to defend against such web-spam attacks. This work was first published in the Journal of Computer Virology.

- Web-spam can be used to promote web sites that perform *drive-by download* attacks. That is, the mere visit of such a site leads to an infection with malware. This infection occurs without additional interaction from the user, and thus commonly goes unnoticed. Therefore, we present a novel system that is able to prevent such attacks and incurs only minimal performance overheads. We published this work at the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA).

- Besides drive-by downloads, a user can also be tricked with social engineering techniques to install malicious applications voluntarily. This is the approach that privacy-threatening applications use to make their way to mobile devices. Regardless of how a malicious application is installed, it is important to security vendors and researchers to have adequate tools and techniques to analyze such malware samples. To this end, we present our novel approach to statically analyze iOS applications for possible privacy leaks that threaten the integrity of the information stored on mobile devices. This work has been accepted at the Network and Distributed System Security Symposium (NDSS), and will be presented there.

1.2 Thesis Overview

As illustrated in Figure 1.1, this thesis is structured in three parts and is based on some of our previously published [26, 29] and not yet published [27, 28] work.

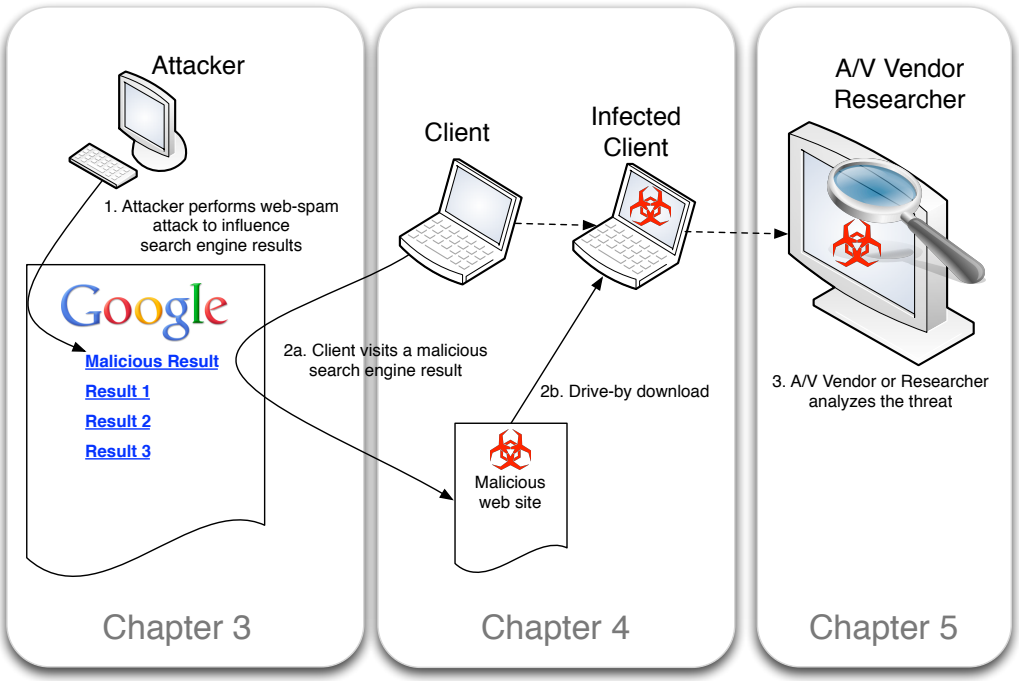

Figure 1.1: Life Cycle of Malware

Figure 1.1 depicts a possible life cycle of a malware sample. An attacker first performs *web-spamming* to attract visitors to his malicious site. Second, this malicious site will attack visitors with a so-called *drive-by download*. Finally, in case of a successful attack, security vendors and researchers will analyze previously unknown threats and provide protection mechanisms to prevent similar attacks from happening in the future.

Following this outline, Chapter 2 strives to make the reader familiar with the different kinds of threats that lurk on the Internet. Furthermore this chapter discusses what techniques attackers apply to infect their targets with malware. In Chapter 3, we present our findings regarding the web-spam problem. Our approach to detect and prevent drive-by downloads is presented in Chapter 4. In

Chapter 5, we present our novel static analysis technique for iOS applications. Chapter 6 discusses related work before Chapter 7 concludes this thesis.

Chapter 2

Threats to Internet-Connected Devices

In this chapter, we will elaborate on the different classes of malware that lurk all over the Internet and threaten its users and their privacy. Furthermore, we summarize the infection vectors used by attackers to install malware on Internet connected devices. We also give an overview of the different techniques that can be applied to analyze malicious software.

2.1 What is Malware?

Software that "*deliberately fulfills the harmful intent of an attacker*" is commonly referred to as malicious software or malware [64]. Terms such as "worm", "virus", or "Trojan horse" are used to classify malware samples that exhibit similar malicious behavior. The first instances of malicious software were viruses. The motivation for the creators of such early malware was usually to highlight some security vulnerabilities or simply to show off technical ability. For example, the cleaning of Bagle worm infected hosts by instances of the Netsky worm could be considered as rivalry between different authors [95]. As time passed, the motivations changed. Today, there is a flourishing underground economy based on malware [105]. It is no longer the fun factor that drives the development in these circles, but the perspective of the money that can be made.

Consider the following scenario which illustrates the distribution of malware and its effects. A bot is a remotely-controlled piece of malware that has infected an Internet-connected computer system. This bot allows an external entity, the so-called bot master, to remotely control this system. The pool of machines that are under control of the bot master is called a botnet. The bot master might rent this botnet to a spammer who misuses the bots to send spam emails containing links to a manipulated web page. This page, in turn, might surreptitiously install a spyware component on a visitors system which collects personal information, such as credit card numbers and online banking credentials. This information is sent back to the attacker who is now able to misuse the stolen information by purchasing goods online. All involved criminals make money at the expense of the infected user, or her bank respectively. With the rise of the Internet and the number of attached hosts, it is now possible for a sophisticated attacker to infect thousands of hosts within hours after releasing the malware into the wild [62]. Recently, a study by Stone-Gross et al. [92] revealed that the Torpig botnet consists of more than 180,000 infected computer systems.

The risk described above motivates the need to create tools that support the detection and mitigation of malicious software. Nowadays, the weapon of choice in combat against malicious software are signature-based anti-virus scanners that match a pre-generated set of signatures against the files of a user. These signatures are created in a way so that they only match malicious software. This approach has at least two major drawbacks. First, the signatures are commonly created by human analysts. This, often, is a tedious and error-prone task. Second, the usage of signatures inherently prevents the detection of unknown threats for which no signatures exist. Thus, whenever a new threat is detected, it needs to be analyzed, and signatures need to be created for this threat. After the central signature database has been updated, the new information needs to be deployed to all clients that rely on that database. Because the signatures are created by human analysts, unfortunately, there is room for error. Multiple AV vendors released signature updates that mistakenly identified legitimate executables as being malware [39, 51], thus, rendering the operating system they were designed to protect, inoperative.

Closely related to the second drawback (i.e., not being able to detect unknown threats), is the inability to detect specifically tailored malware. Besides

the mass phenomenon of Internet worms and malicious browser plug-ins, one can observe the existence of specifically tailored malware that is created for targeted attacks [5]. Spyware programs might be sent via email to the executive board of a company with the specific intent to capture sensitive information regarding the company. Because these malware samples usually do not occur in the wild, it is unlikely that an anti-virus vendor receives a sample in time to analyze it and produce signatures. This means that the spyware could be operational in the company for a long time before it is detected and removed, even if anti-virus software is in place.

The inability to detect unknown threats is an inherent problem of signature-based detection techniques. This is overcome by techniques that base their decision of identifying a piece of code as being malicious or not, on the observation of the software's behavior. Although these techniques allow for the detection of previously unknown threats to a certain extent, they commonly suffer from false positives. That is, legitimate samples are falsely classified by the detection system as being malicious due to the detector's inability to distinguish legitimate from malicious behavior under all circumstances.

2.1.1 Types of Malware

This section gives a brief overview of the different classes of malware programs that have been observed in the wild. The following paragraphs are solely intended to familiarize the reader with the terminology that we will be using in the remainder of this work. Furthermore, these classes are not mutually exclusive. That is, specific malware instances may exhibit the characteristics of multiple classes at the same time. A more detailed discussion of malicious code in general can be found for example in Skoudis et al. [86], or Szor [94].

Worm Prevalent in networked environments, such as the Internet, Spafford [89] defines a worm as "*a program that can run independently and can propagate a fully working version of itself to other machines*." This reproduction is the characteristic behavior of a worm. The Morris Worm [89] is the first publicly known instance of a program that exposes worm-like behavior on the Internet. More recently, in July 2001, the Code Red worm infected

thousands (359,000) of hosts on the Internet during the first day after its release [62]. Today, the Storm worm and others are used to create botnets that are rented out by the bot masters to send spam emails or perform distributed denial of service attacks (DDOS) [54], where multiple worm infected computers try to exhaust the system resources or the available network bandwidth of a target in a coordinated manner.

Virus "*A virus is a piece of code that adds itself to other programs, including operating systems. It cannot run independently - it requires that its "host" program be run to activate it.*" [89] As with worms, viruses usually propagate themselves by infecting every vulnerable host they can find. By infecting not only local files but also files on a shared file server, viruses can spread to other computers as well.

Trojan Horse Software that pretends to be useful, but performs malicious actions in the background, is called a Trojan horse. While a Trojan horse can disguise itself as any legitimate program, frequently, they pretend to be useful screen-savers, browser plug-ins, or downloadable games. Once installed, their malicious part might download additional malware, modify system settings, or infect other files on the system.

Spyware Software that retrieves sensitive information from a victims' system and transfers this information to the attacker is denoted as *spyware*. Information that might be interesting for the attacker include accounts for computer systems or bank account credentials, a history of visited web pages, and contents of documents and emails.

Bot A bot is a piece of malware that allows its author (i.e., the *bot master*) to remotely control the infected system. The set of bots collectively controlled by one bot master is denoted a *botnet*. Bots are commonly instructed to send spam emails or perform spyware activities as described above.

Rootkit The main characteristic of a rootkit is its ability to hide certain information (i.e., its presence) from a user of a computer system. Rootkit techniques can be applied at different system levels, for example, by instrumenting API calls in user-mode or tampering with operating system structures

if implemented as a kernel module or device driver. Manipulating the respective information allows a rootkit to hide processes, files, or network connections on an infected system. Moreover, virtual machine based rootkits [56, 81, 106] conceal their presence by migrating an infected operating system into a virtual machine. The hiding techniques of rootkits are not bad per se, but the fact that many malware samples apply rootkit techniques to hide their presence in the system, justifies mentioning them here.

2.1.2 Infection Vectors

This section gives an overview of the infection vectors that are commonly used by attackers to infect a victim's system. Brief examples are used to illustrate how these infections work and how malware used them in the past.

Exploiting Vulnerable Services over the Network

Network services running on a server provide shared resources and services to clients in a network. For example, a DNS service provides the capabilities of resolving host names to IP addresses, a file server provides shared storage on the network. Many commodity off the shelf operating systems come with a variety of network services that are already installed and running. Vulnerabilities in such services might allow an attacker to execute her code on the machine that is providing the service. Large installation bases of services that share the same vulnerability (e.g., [61]) pave the way for automatic exploitation. Thus, such conditions allow malicious software to infect accessible systems automatically. This characteristic makes network service exploitation the preferred method for infection by worms. Moreover, services that provide system access to remote users, and authenticate these users with passwords (e.g., ssh, administrative web interfaces, etc.), are frequently exposed to so-called *dictionary attacks*. Such an attack iteratively tries to log into a system using passwords stored in a dictionary.

Drive-by downloads

Drive-by downloads usually target a victim's web browser. By exploiting a vulnerability in the web browser application, a drive-by download is able to fetch malicious code from the web and subsequently execute it on the victim's machine. This usually happens without further interaction with the user. In contrast to exploiting vulnerabilities in network services in which push-based infection schemes are dominant, drive-by downloads follow a pull-based scheme [75]. That is, the connections are initiated by the client as it is actively requesting the malicious contents. Therefore, firewalls that protect network services from unauthorized access cannot mitigate the threat of drive-by attacks. Currently, two different techniques are observed in the wild that might lead to a drive-by infection:

- API misuse: If a certain API allows for downloading an arbitrary file from the Internet, and another API provides the functionality of executing a random file on the local machine, the combination of these two APIs can lead to a drive-by infection [60]. The widespread usage of browser plug-ins usually gives attackers a huge portfolio of APIs that they might use and combine for their nefarious purposes in unintended ways.

- Exploiting web browser vulnerabilities: This attack vector is identical to the case of exploitable network services. Moreover, as described in Sotirov [87] and Daniel et al. [23] the availability of client-side scripting languages, such as Javascript or VBScript, provide the attacker with additional means to successfully launch an attack. A detailed discussion of such threats is presented in Chapter 4.

Before a drive-by download can take place, a user is first required to visit the malicious site. In order to lure the user into visiting the malicious site, attackers perform social engineering and send spam emails that contain links to these sites or infect existing web pages with the malicious code. For example, the infamous Storm worm makes use of its own botnet resources to send spam emails containing links to such attack pages [54].

To maximize the number of sites that host such drive-by attacks, attackers exploit vulnerabilities in web applications that allow them to manipulate these

web sites (e.g., [22]). This is an example in which attackers use the infection vector of vulnerable network services to launch drive-by download attacks on clients of that service (e.g., a web site). Another technique for attackers to lure users to their web sites is by trying to cheat the ranking algorithms that web search engines use, to sort result pages. An attacker may create a page that is specifically instrumented to rank high for common search query terms. If the page is listed on a top position for these query terms, it will result in a large number of visitors [9, 48]. Chapter 3 elaborates on these so-called *web-spam* attacks in detail. Provos et al. [75] discovered that more than 1.3% of results to Google search queries include at least one page that tries to install malicious software on a visitor's machine. Provos et al. [76] also analyzed the techniques malware authors apply to lure a user to open a connection to a host that performs drive-by download attacks. The most prevalent of these actions are circumventing web-server security measures, providing user generated content, advertising schemes, and malicious widgets.

Social Engineering

All techniques that lure a user into deliberately executing malicious code on her device, possibly under false pretenses, are subsumed as social engineering attacks. There are virtually no limits to the creativity of attackers when social engineering is involved. Asking the user to install a provided "codec" to view the movie that is hosted on the current web site, clicking and opening an image that is attached to a spam email, or speculating that the user plugs a "found" USB key into her computer eventually [91] are just a few examples of social engineering. At the time of writing, virtually all malware for mobile devices is distributed using social engineering. That is, malicious software or ill-behaved software is seeded in the respective market places and the user deliberately downloads and installs the offending programs from there. Commonly, the user is not aware of the programs' malicious behavior. Therefore, to better assess the privacy threats such applications pose for their users we present our novel static analysis system for iOS applications in Chapter 5.

2.2 Understanding the Threats

Regardless of what what infection vector a malware sample used to compromise an Internet-connected device, it is important for security vendors and researchers to have appropriate tools and techniques to analyze these samples. The available analysis methods are divided into two categories. *Dynamic analysis* subsumes the techniques that execute the malware in a controlled environment and monitor the malware's execution. *Static analysis*, on the other hand, refers to those techniques that perform the analysis by inspecting the sample without actually running its code.

2.2.1 Static Analysis

Static analysis techniques can be applied on different representations of a program. If the source code is available, static analysis tools can help finding memory corruption flaws [14, 15, 34] and prove the correctness of models for a given system.

Note that when dealing with potentially malicious code, the source code is usually not available. However, static analysis techniques can also be applied to the binary representation of a program. Such approaches commonly start by disassembling the code of the binary under analysis. That is, the binary code is translated to assembly instructions of the respective instruction set. Based on this disassembly, further analysis steps can be performed. By extracting the *call graph*, for example, an analyst can understand which functions invoke what other functions in the program. This requires that the analysis is able to distinguish function boundaries in the binary successfully. A *control flow graph*, on the other hand, illustrates the possible execution paths on a finer grained level (e.g., within a function body). When compiling the source code of a program into a binary executable, some information (e.g., the size of data structures or variables) is lost. This loss of information complicates the task of statically analyzing binary programs.

Limitations of Static Malware Analysis Approaches: Generally, the source code of malware samples is not readily available. This reduces the applicable

static analysis techniques for malware analysis to those that retrieve the information from the binary representation of the malware. Analyzing binaries brings along intricate challenges. Consider, for example, that most malware attacks devices executing instructions in the IA32 instruction set (e.g., PCs). The disassembly of such programs might result in ambiguous results if the binary employs self modifying code techniques, such as packer programs. Additionally, malware relying on values that cannot be statically determined (e.g., current system date, indirect jump instructions) exacerbate the application of static analysis techniques. However, most mobile devices feature RISC CPUs that have a different, much simpler instruction set, such as ARMv7. Therefore, some of the limitations for static analysis techniques on IA32 binaries are not fully applicable to applications compiled to run on such RISC CPUs.

2.2.2 Dynamic Analysis

Monitoring the actions performed by a program, and the effects these actions have on the system, while the program is executed is called *dynamic analysis*.

Function Call Monitoring: It is relatively easy for dynamic malware analysis tools to monitor the invocation of functions during program execution. Regardless of whether the called function is implemented in the binary itself, or in a dynamically loaded library, calling a function is commonly performed by a `call` instruction. Thus, intercepting such calls enables a dynamic malware analysis tool to create the function call graph for the observed execution paths. System calls are a special kind of function, as they provide the only means for a program to communicate or influence its environment (e.g., file system, network, etc.). This system call interface is commonly well documented and thus, many dynamic analysis techniques also monitor system call invocations.

Of course, many dynamic analysis tools keep track of parameter and return values between function calls. For example, opening and writing to a file requires at least two separate function or system calls. It is important for a dynamic analysis tool to keep track of whether these two function calls refer to the same file object or not.

Apart from focusing on the execution of a single program, a dynamic analysis tool might be designed to monitor a complete system. Such whole system analysis techniques allow the analyst to gain insight into the interaction of multiple programs.

Information Flow Tracking: An orthogonal approach to the monitoring of function calls during the execution of a program is the analysis on *how* the program processes data. The goal of information flow tracking is to shed light on the propagation of "interesting" data throughout the system while a program manipulating this data is executed. In general, the data that should be monitored is specifically marked (tainted) with a corresponding label. Whenever the data is processed by the application, its taint-label is propagated. Assignment statements, for example, usually propagate the taint-label of the source operand to the target. Besides the obvious cases, policies have to be implemented that describe how taint-labels are propagated in more complex scenarios. Such scenarios include the usage of a tainted pointer as the base address when indexing to an array or conditional expressions that are evaluated on tainted values.

Limitations of Dynamic Malware Analysis Approaches: Although dynamic malware analysis techniques are a powerful instrument to gain insight into the operation of malware, these techniques are also limited by a number of factors. For example, a malware sample could perform a series of operations to determine whether it is currently being analyzed. In case the sample detects the analysis environment it could cease to exhibit any malicious behavior and therefore escape analysis. Furthermore, dynamic analysis suffers from the problem of incomplete path coverage. That is, information is only collected while the program is running and only along the code paths that are actually executed. Therefore, logic bombs or startup delays are often successful countermeasures for malware samples to evade dynamic analysis.

2.3 Summary

This chapter made the reader familiar with the different infection vectors that attackers make use of to infect target devices with all sorts of malware. Further-

more, we briefly elaborated on what analysis techniques can be used to scrutinize malware once it successfully infected a system.

Web-spam and *drive-by downloads* attacks often go hand in hand, and are among the most prevalent of the attack scenarios on the Internet. Thus, the following two chapters will elaborate on these attacks. Of course, we also propose mitigation approaches to these threats.

Chapter 3

Influencing Search Engine Results with Web-Spam

Search engines are designed to help users find relevant information on the Internet. Typically, a user submits a query (i.e., a set of keywords) to a search engine, which then returns a list of links to pages that are most relevant to this query. To determine the most-relevant pages, a search engine selects a set of candidate pages that contain some or all of the query terms and calculates a *page score* for each page. Finally, a list of pages, sorted by their score, is returned to the user.

This score is calculated from properties of the candidate pages, so-called features. Unfortunately, details on the exact algorithms that calculate these ranking values are kept secret by search engine companies, since this information directly influences the quality of the search results. Only general information is made available. For example, in 2007, Google claimed to take more than 200 features into account for the ranking value [42].

The way in which pages are ranked directly influences the set of pages that are visited frequently by the search engine users. The higher a page is ranked, the more likely it is to be visited [9]. This makes search engines an attractive target for everybody who aims to attract a large number of visitors to her site. There are three categories of web sites that benefit directly from high rankings in search engine results. First, sites that sell products or services. In their context, more visitors imply more potential customers. The second category contains sites that are

financed through advertisement. These sites aim to rank high for any query. The reason is that they can display their advertisements to each visitor, and, in turn, charge the advertiser. The third, and most dangerous, category of sites that aim to attract many visitors by ranking high in search results are sites that distribute malicious software. Such sites typically contain code that exploits web browser vulnerabilities to silently install malicious software on the visitor's computer. Once infected, the attacker can steal sensitive information (such as passwords, financial information, or web-banking credentials), misuse the user's bandwidth to join a denial of service attack, or send spam. The threat of drive-by downloads (i.e., automatically downloading and installing software without the user's consent as the result of a mere visit to a web page) and distribution of malicious software via web sites has become a significant security problem. Web sites that host drive-by downloads are either created solely for the purpose of distributing malicious software or existing pages that are hacked and modified (for example, by inserting an `iframe` tag into the page that loads malicious content). Provos et al. [75, 77] observe that such attacks can quickly reach a large number of potential victims, as at least 1.3% of all search queries directed to the Google search engine contain results that link to malicious pages. Moreover, the pull-based infection scheme circumvents barriers (such as web proxies or NAT devices) that protect from push-based malware infection schemes (such as traditional, exploit-based worms). As a result, the manipulation of search engine results is an attractive technique for attackers that aim to attract victims to their malicious sites and spread malware via drive-by attacks [79].

Search engine optimization (SEO) companies offer their expertise to help clients improve the rank for a given site through a mixture of techniques, which can be classified as being acceptable or malicious. Acceptable techniques refer to approaches that improve the content or the presentation of a page to the benefit of users. Malicious techniques, on the other hand, do not benefit the user but aim to mislead the search engine's ranking algorithm. The fact that bad sites can be pushed into undeserved, higher ranks via malicious SEO techniques leads to the problem of *web spam*.

Gyöngyi and Garcia-Molina [44] define web spam as every deliberate human action that is meant to improve a site's ranking without changing the site's true

value. Search engines need to adapt their ranking algorithms continuously to mitigate the effect of spamming techniques on their results. For example, when the Google search engine was launched, it strongly relied on the PageRank [8] algorithm to determine the ranking of a page where the rank is proportional to the number of incoming links. Unfortunately, this led to the problem of link farms and "Google Bombs," where enormous numbers of automatically created forum posts and blog comments were used to promote an attacker's target page by linking to it.

Clearly, web spam is undesirable, because it degrades the quality of search results and draws users to malicious sites. Although search engines invest a significant amount of money and effort into fighting this problem, checking the results of search engines for popular search terms demonstrates that the problem still exists. In this work, we aim to post-process results returned by a search engine to identify entries that link to spam pages. To this end, we first study the importance of different features for the ranking of a page. In some sense, we attempt to reverse-engineer the "secret" ranking algorithm of a search engine to identify the most important features. Based on this analysis, we attempt to build a classifier that inspects these features to identify indications that a page is web spam. When such a page is identified, we can remove it from the search results.
The two main contributions presented in this chapter are the following:

- We conducted comprehensive experiments to understand the effects of different features on search engine rankings.

- We developed a system that allows us to reduce spam entries from search engine results by post-processing them. This protects users from visiting either spam pages or, more importantly, malicious sites that attempt to distribute malware.

3.1 Overview

In this section, we first provide an overview of our approach to determine the features that are important for the ranking algorithm. Then, we describe how we

use this information to develop a technique that allows us to identify web spam pages in search engine results.

3.1.1 Inferring Important Features

Unfortunately, search engine companies keep their ranking algorithms and the features that are used to determine the relevance of a page secret. However, to be able to understand which features might be abused by spammers and malware authors to push their pages, a more detailed understanding of the page ranking techniques is necessary. Thus, the goal of the first step of our work is to determine the features of a web page that have the most-pronounced influence on the ranking of this page.

A feature is a property of a web page, such as the number of links pointing to other pages, the number of words in the text, or the presence of keywords in the title tag. To infer the importance of the individual features, we perform "black-box testing" of search engines. More precisely, we create a set of different test pages with different combinations of features and observe their rankings. This allows us to deduce which features have a positive effect on the ranking and which contribute only a little.

3.1.2 Removing Spam from Search Engine Results

Based on the results of the previous step, we developed a system that aims to remove spam entries from search engine results. To this end, we examine the results that are returned by a search engine and attempt to detect links that point to web spam pages. This is a classification problem; every page in the result set needs to be classified as either spam or nospam.

To perform this classification, we have to determine those features that are indicators of spam. For this, we leverage the findings from the first step and a labeled training set to construct a C4.5 decision tree. A decision tree is useful because of its intuitive insight into which features are important to the classification. Using this classifier, we can then check the results from the search engine and remove those links that point to spam pages. The result is an improvement of search quality and fewer visits to malicious pages.

3.2 Feature Inference

In this section, we give a detailed introduction to our inference techniques for important features. First, we discuss which features we selected. Then, we describe how these features are used to prepare a set of (related, but different) pages. Finally, we report on the rankings that major search engines produced for these pages and the conclusions that we could draw about the importance of each feature.

3.2.1 Feature Selection

As mentioned previously, we first aim to "reverse engineer" the ranking algorithm of a search engine to determine those features that are relevant for ranking. Based

1	Keyword(s) in title tag
2	Keyword(s) in body section
3	Keyword(s) in H1 tag
4	External links to high quality sites
5	External links to low quality sites
6	Number of inbound links
7	Anchor text of inbound links contains keyword(s)
8	Amount of indexable text
9	Keyword(s) in URL file path
10	Keyword(s) in URL domain name

Table 3.1: Feature set used for inferring important features.

on reports from different SEO vendors [83] and study of related work [7, 32], we chose ten presumably important page features (see Table 3.1). We focused on features that can be directly influenced by us. The rationale is that only from the exact knowledge of the values of each feature, one can determine their importance. Additionally, the feature value should remain unchanged during the whole experiment. This can only be ensured for features under direct control.

When considering features, we first examined different locations on the page where a search term can be stored. Content-based features, such as body-, title-, or headings-tags are considered since these typically provide a good indicator for the

information that can be found on that page. Additionally, we also take link-based features into account (since search engines are known to rely on linking information). Usually, the number of incoming links pointing to a page (i.e., the *in-link* feature) cannot be influenced directly. However, by recruiting 19 volunteers willing to host pages linking to our experiments, we were able to fully control this feature as well.

Together with features that are not directly related to the page's content (e.g., keyword in domain name), we believe to have covered a wide selection of features from which search engines can draw information to calculate the rankings.

We are aware of the fact that search engines also take temporal an location-based aspects into account when computing their rankings (e.g., how does a page or its link count evolve over time). However, we decided against adding time- and location-dependent features to our feature set because this would have made the experiment significantly more complex and does not necessarily add value. We aim to eliminate the influence of these aspects by always publishing and modifying pages from the same location and at the same time.

3.2.2 Preparation of Pages

Once the features were selected, the next step was to create a large set of test pages, each with a different combination and different values of these features. For these test pages, we had to select a combination of search terms (a query) for which no search engine would produce any search results prior to our experiment (i.e., only pages that are part of our experiment are part of the results). We arbitrarily chose "gerridae plasmatron" as the key phrase to optimize the pages for.[1] Remember, the goal is to estimate the influence of page features to the ranking algorithms and not to determine whether our experiment pages outperform (in terms of search engine response position) existing legitimate sites.

Using this search phrase, we prepared the test pages for our experiment. To this end, we first created a reference page consisting of information about gerridae and plasmatrons compiled from different sources. In a second step, this reference page was copied 90 times. To evade duplicate detection by search en-

[1] Gerridae is the Latin expression for water strider, plasmatron is a special form of an ion source.

gines (where duplicate pages are omitted from the results), each of these 90 pages was obfuscated by substituting many words in a manner similar to [55]. Subsequent duplicate detection by the search engines (presumably based on title and headline tag) required a more aggressive obfuscation scheme where title texts and headlines where randomized as well.

For features whose possible values exceed the boolean values (i.e., present or absent), such as keyword frequencies, we selected representative values that correspond to one of the following four classes.

- The feature is not present at all.
- The feature is present in *normal* quantities.
- The feature is present in *elevated* quantities.
- The feature is present in *spam* quantities.

That is, a feature with a large domain (i.e., set of possible values) can assume four different values in our experiment. Of course, there is no general rule to define a precise frequency for which a feature can be considered to be normal, elevated, or spam. Thus, we manually examined legitimate and spam pages and extracted average, empirical frequencies for the different values. For example, for the frequencies of the keyword in the body text, a 1% keyword frequency is used as a baseline, 4% is regarded elevated, and 10% is considered to be spam.

Since only 90 domains were available, we had to select a representative subset of the 16,392 possible feature combinations. Moreover, to mitigate any measurement inaccuracies, we decided to do all experiments triple-redundant. That is, we chose a subset of 30 feature combinations, where each combination forms an experiment group that consists of three identical instances that share the same feature values. For these 30 experiment groups, we decided to select the feature values in a way to represent different, common cases. The regular case is a legitimate site, which is represented by the reference page. For this page, all feature values belong to the *normal* class. Other cases include keyword stuffing in different page locations (e.g., body, title, headlines), or differing amounts of incoming and outgoing links. The full list of the created experiments is listed in Table 3.2.

No.	Feature Combination	Description
1	1,2,3,4,7,9	Baseline
2	1,2,3,7,$9	Baseline with much text
3	1,2,3,$6,7,$9	Baseline with much text, links to low quality sites
4	1,+2,3,7,9	Elevated use of keywords in `BODY`
5	1,$2,3,7,9	Keyword spamming of `BODY`
6	+1,2,3,7,9	Elevated use of keywords in the `TITLE`
7	$1,2,3,7,9	Keyword spamming of `TITLE`
8	1,2,3,$4,7,9,10	Keyword spamming of the URL
9	$1,$2,$3,$4,$5,7,9	Spam all on site
10	$1,$2,$3,$4,$5,$7,9	Spam all
11	$1,$2,$3,$4,$5,$7,$9	Spam all with much text
12	1,2,3,4,5,7,9	Include links to high quality pages
13	1,2,3,4,+5,7,9	Include more links to high quality pages
14	1,2,3,4,$5,7,9	Include many links to high quality pages
15	1,2,3,4,6,7,9	Include links to low quality pages
16	1,2,3,4,+6,7,9	Include more links to low quality pages
17	1,2,3,4,$6,7,9	Include many links to low quality pages
18	1,2,3,4,7,8,9	In-links with keywords in anchor text
19	1,2,3,4,7,9	In-links without keywords in anchor text
20	1,2,3,4,+7,8,9	Elevated amount of in-links w/ kw in anchor text
21	1,2,3,4,+7,9	Elevated amount of in-links w/o kw in anchor text
22	1,2,3,4,$7,8,9	Spam amount of in-links w/ kw in anchor text
23	1,2,3,4,$7,9	Spam amount of in-links w/o kw in anchor text
24	1,2,3,$4,7,9	URL keyword spam without domain name
25	1,2,3,4,7,9,10	Baseline with keyword in domain name
26	$1,$2,$3,$4,$5,$7,$9,10	Spam all w/ kw in domain name
27	1,2,3,4,7,8,9	In-links with keywords and kw in file name
28	1,2,3,4,7,9	In-links without keywords and kw in file name
29	1,2,3,4,7,8,9,10	In-links with keywords and kw in domain name
30	1,2,3,4,7,9,10	In-links without keywords and kw in domain name

Table 3.2: List of experiment groups.
Column 2 references the features in Table 3.1 and captures the list of applied features for this experiment group. The lack of a feature in the description denotes that the feature is not used for this experiment, the prefix (+) indicates that a feature is applied in elevated quantities, where ($) means the feature is present in spam quantities. The third column is a description of the case that this experiment group reflects.

3.2.3 Execution of Experiments and Results

Once the 30 experiment groups (i.e., 90 pages) were created, they were deployed to 90 freshly registered domains, served by four different hosting providers. Additionally, some domains were hosted on our department web server. This was done to prevent any previous reputation of a long-lived domain to influence the rankings, and hence, our results.

Once the sites were deployed, we began to take hourly snapshots of the search engine results for the query "gerridae plasmatron." To keep the results comparable we queried the search engines for results of the English web (i.e., turning off any language detection support). In addition, we also took snapshots of results to queries consisting of the individual terms of the key phrase. Since all major search engines had results for the single query terms (gerridae/plasmatron) before our experiment started, we gained valuable insights into how our sites perform in comparison to already existing, mostly legitimate sites.

Our experiment was carried out between December 2007 and March 2008. During 86 days, we submitted 2,312 queries to Google and 1,700 queries to the Yahoo! search engine. Interestingly, we observed that rankings usually do not remain stable over a longer period of time. In fact, the longest period of a stable ranking for all test pages was only 68 hours for Google and 143 hours for Yahoo!. Also, we observed that Google refuses to index pages whose path (in the URL) contained more than five directories. This excluded some of our test pages from being indexed for the first couple of weeks.

One would expect that instances within the same experiment group occupy very close positions in the search engine results. Unfortunately, this is not always the case. While there were identical instances that ranked at successive or close positions, there were also some experiment groups whose instances were significantly apart. We suspect that most of these cases are due to duplicate detection (where search engines still recognized too many similarities among these instances).

At the time of writing, querying Google for "gerridae plasmatron" resulted in 92 hits. Including omitted results, 330 hits are returned. Yahoo! returns 82 hits without and 297 hits including the omitted results. Microsoft Live search returns

only 28 pages. Since Microsoft Live search seemed slower in indexing our test pages, we report our results only for Google and Yahoo!.

Note that the Google and Yahoo! results consist of more than 90 elements. The reason for this is that the result sets also contain some sites of the volunteers, which frequently contain the query terms in anchor texts pointing to the test sites.

For Google, searching for "gerridae" yields approximately 55,000 results. Our test pages constantly managed to occupy five of the top ten slots with the highest ranking page at position three. Six was the highest position observed for the "plasmatron" query.

For Yahoo!, we observed that for both keywords pages of our experiments managed to rank at position one and stay there for about two weeks.

3.2.4 Extraction of Important Features

Because of the varying rankings, we determined a page's position by averaging its positions over the last six weeks of the experiment. We decided for the last six weeks, since the initial phase of our experiment contains the inaccuracies that were introduced due to duplicate detection. Also, it took some time before most pages were included in the index. We observed that when we issued the same query to Google and Yahoo!, they produced different rankings. This indicates that the employed algorithms weight features apparently differently. Thus, we extracted different feature weights for Google and Yahoo! as described below.

Knowing the combinations of all feature values for a page k and observing its position $pos(k)$ in the rankings, our goal is now to assign an (optimal) weight to each feature that best captures this feature's importance to the ranking algorithm. As a first step, we define a function $score$. This function takes as input a set of weights and feature values and computes a score $score(k)$ for a page k.

$$score(k) = \sum_{i=1}^{n} f_i^k \cdot w_i$$

$$n \ldots \text{number of features}$$
$$w_i \in [-1, 1] \ldots \text{weight of feature } i$$
$$f_i^k \in 0, 1 \ldots \text{presence of feature } i \text{ in test page } k$$

This calculation is repeated for all test pages using the same weights. Once all scores are calculated, the set of test pages is sorted by their score. This allows us to assign a predicted ranking $rank(k)$ to each page. Subsequently, distances between the predicted ranking and the real position are calculated for all test pages. When the sum of these distances reaches the minimum, the weights are optimal. This translates to the following objective function of a linear programming problem (LP):

$$min : \sum_{k=1}^{m} \alpha_k |pos(k) - rank(k)|$$

Note that we added the factor $\alpha(k) = m - pos(k)$ to the LP, which allows higher-ranking test pages to exert a larger influence on the feature weights (m is the number of test pages). This is to reflect that the exact positions of lower-ranking pages often fluctuated significantly. Thus, we required a way to reduce these "random" influence on the calculation of the weights. Solving this LP with the Simplex algorithm results in weights for all features that, over all pages, minimize the distance between the predicted rank and the actual position.

For Google, we found that the number of search terms in the title and the text body of the document had the strongest, positive influence on the ranking. Also, the number of outgoing links was important. On the other hand, the fact that the keywords are part of the file path had only a small influence. This is also true for the anchor text of inbound links.

For Yahoo!, the features were quite different. For example, the fact that a keyword appears in the title has less influence and even decreases with an increase of the frequency. Yahoo! also (and somewhat surprisingly) puts significantly more weight on both the number of incoming and outgoing links than Google. On the other hand, the number of times keywords appear in the text have no noticeable, positive effect.

As a last step, we examine the quality of our predicted rankings. To this end, we calculate the distance between the predicted position and the actual position for each experiment group. More precisely, Figure 3.1 shows, for each experiment

group, the distance between the actual and predicted positions, taking the closest match for all three pages in each group.

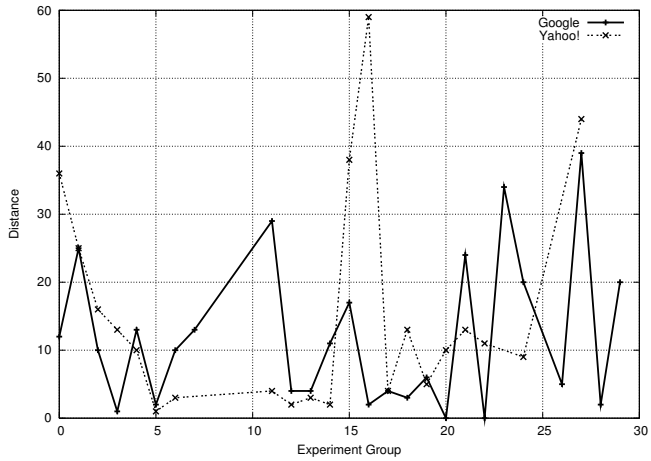

Figure 3.1: Differences when comparing predicted values with actual ranking positions.

Considering the Google results, 78 experiment pages of 26 experiment groups were listed in the rankings. The missing experiment groups are those whose pages have a directory hierarchy level of five, and thus, were not indexed by the search engine spiders. Looking at the distance, we observe that we can predict the position for six groups (23%) within a distance of two, and for eleven groups (42%) with a distance of five or less (over a range of 78 positions). For Yahoo!, when comparing the experiment groups with the rankings, 21 groups appear in the results. Three (14%) of these groups are predicted within a distance of two, while eight (38%) are within a distance of five or less positions to the observed rank (over a range of 63 positions).

At a first glance, our predictions do not appear very precise. However, especially for Yahoo!, almost all predictions are reasonably close to the actual results. Also, even though our predictions are not perfectly accurate, they typically reflect the general trend. Thus, we can conclude that our general assessment of the importance of a feature is correct, although the precise weight value might be dif-

ferent. Also, we only consider a linear ranking function, while the actual ranking algorithms are likely more sophisticated.

3.3 Reducing Spam from Search Engine Results

In this section, we present the details of our prototype system to detect web spam entries in search engine results. The general idea behind this system is to use machine learning techniques to generate a classification model (a classifier) that is able to distinguish between legitimate and spam sites by examining a page's features. The following section first presents the details on how the system operates. Then, the evaluation section describes our spam detection effectiveness.

3.3.1 Detecting Web Spam in Search Engine Results

During the previous feature inference step, we determined the features that are most important to search engine ranking algorithms. Assuming an attacker can also learn this information, this suggests that the attacker will focus on those features that have the most pronounced influence on the rankings. This motivates our approach in developing a classifier that distinguishes spam and non-spam pages according to these features.

The classifier presented in this section is developed for the Google search engine. Thus, we include those features that are most relevant for Google, as discussed in the previous section. These are the number of keywords in the title, body, and domain name. In addition, we consider linking information. While counting the outgoing links of a page is trivial, the number of incoming links is not easily determinable. The information of how many in-links point to a page is not made available by search engines. This is the reason why we have to estimate the corresponding features with the help of `link:` queries. Google and Yahoo! support queries in the form of `link:www.example.com` resulting in a list of pages that link to `www.example.com`. The drawback is that neither the Google nor the Yahoo! results contain all pages that link to the queried page. Thus, these numbers are only an approximation of the real number of links pointing to a site.

On the other hand, we can introduce additional information sources that were not available to us before. For example, the PageRank value (as reported by the Google toolbar) was added to the feature set. This value could not be used for the experiment because of the infrequent updates (roughly every three months) and its violation of the requirement that we can control each feature directly.

Classifier. To build a classifier for web pages, we first require a labeled training set. Another set of data is required to verify the resulting model and evaluate its performance. To create these sets, 12 queries were submitted to the Google search engine (asking for popular search terms, extracted from Google's list of popular queries, called Zeitgeist [41]). For every query, the first 50 results were manually classified as legitimate or spam/malicious. Discarding links to non-HTML content (e.g., PDF or PPT files) resulted in a training data set consisting of 295 sites (194 legitimate, 101 spam). The test data set had 252 pages (193 legitimate, 59 spam).

All result pages were downloaded and fed into feature extractors that parse the HTML source code and return the value (i.e., the frequency) of the feature under consideration. If the query consists of multiple terms, query dependent feature extractors report higher values if the full query matches the analyzed feature. The rationale behind this is that a single heading tag that contains the whole query indicates a better match than multiple, individual heading tags, each containing one of the query terms. Feature extractors that follow this approach are marked with an (X) in the following list, which enumerates all the features that we consider:

- **Title:** the number of query terms from HTML `title` tag (X)
- **H1-Tag:** the number of query terms in HTML `H1`-tags (X)
- **Body:** the number of query terms in the HTML `body` section (X)
- **Domain name:** the number of query terms in the domain name part of the URL
  (e.g, www.*gerridae-plasmatron*.com/index.php)
- **Filepath:** the number of query terms in the path of the URL
  (e.g., www.example.org/*gerridae-plasmatron*/index.php)
- **Out-links:** the total number of outbound links
- **Out-links-keywords:** number of outbound links containing keywords as anchor texts (X)

- **In-links - Google:** the number of inbound links reported by Google `link:` query
- **In-links - Yahoo!:** the number of inbound links reported by Yahoo! `link:` query
- **PageRank site:** the Google PageRank value for the URL as reported by the Google toolbar
- **PageRank domain:** the Google PageRank value for the domain as reported by the Google toolbar
- **Word count:** total number of words in the document
- **Tfreq:** the frequency of query terms appearing on the page (number of query terms / number of words on page)

Using the labeled training data as a basis, we ran a series of algorithms to train different classification models. Each classifier was evaluated against the test data set. To this end, we leveraged the Weka [102] toolkit that provides support for a multitude of classification models.

3.3.2 Evaluated Classification Models

We evaluated a total of eight distinct classification models from the *Weka* toolkit to assess their applicability for our purpose. Each model comes with a unique set of properties and settings, which we briefly discuss here. A comparative evaluation of the presented classification models is illustrated in Table 3.3.

Naive Bayesian Classifier Heckerman [47] describes Bayesian networks as a graphical model that encodes probabilistic relationships among variables of interest. Such a network can be learned from training data or a-priori knowledge. In our case, the model is inferred from our training data set. The applied naive Bayesian classifier is described by John et al. [49] as a Bayesian network that relies on two simplifying assumptions. First, the predictive attributes for a given class are conditionally independent. Furthermore, no hidden or latent attributes influence the prediction process.

Support Vector Machines with Sequential Minimal Optimization Support vector machines [46] (SVMs) can classify objects by projecting them into a n-dimensional space. The dimensional size is determined by the number of characteristics of the training- or query-vector. In our case, the feature vector is of size 13, resulting in an equally sized vector space. The actual classification is done by filling the vector space with labeled elements from the training set and creating a hyperplane that separates the points according to their labels. A query can then be categorized by simply projecting it into the same space and determining on which side of the plane it resides.

Platt [70] introduced sequential minimal optimization (SMO) as an alternative approach to train support vector machines. The original SVM's training step requires that a large quadratic programming problem is solved. SMO speeds up this training by breaking the QP problem into a series of smallest possible QP problems which can be solved analytically. Once the SVM is trained, classification is performed like in a normal SVM.

Locally weighted learning As explained in [4], the locally weighted learning approach can be used to classify an unknown sample by the following steps. Given the sample point to classify, the system calculates those points in the training data set that are close to this sample. The class of each of those points influences the classification decision inversely proportional to the measured distance. That is, the closer a point in the training set is to the sample to classify, the more influence this point's class has on the final classification decision.

Fuzzy Lattice Reasoning Classifier (FLR) During training, this classification model [53] generates hyperboxes based on the points contained in the training data set. A hyperbox corresponds to a rule which indicates that a point located within this box is a member of the respective class. Training is performed iteratively and each point from the training data set induces a new rule. For each new rule the model calculates a fuzzy degree of inclusion with the existing rules. The maximum value of these degrees suggests, how existing and new rules must be combined. For classifying an unknown data point, the system calculates the in-

clusion degrees for the rule induced by the unknown point and assigns this point to the class of the rule with the highest value.

ConjunctiveRule This classifier aims to make decisions based on a single logic proposition of the form `if` A`, then` B. The proposition's `antecedent` (that is, the A part of the proposition) is the conjunction of selected feature rules, whereas the `consequent` is the mere statement whether a site is to be labeled as spam or not.

The proposition is generated by iteratively selecting single features from the data set. The algorithm greedily adds feature rules based on their *information gain* calculated on the instances of the training set that are not covered by the current rule. To avoid over-fitting of possibly irrelevant features, the Weka toolkit uses *reduced error pruning* in its default configuration.

An exemplary logic proposition, generated from our training set, can be seen below:

$$(title < 0.166667) \wedge (domainname \leq 1.5) \wedge (tfreq \leq 3.5) \rightarrow \textit{no-spam}$$

J48 The `J48` classifier implements the $C4.5$ algorithm [78] for generating decision trees. Given a set of features, the heuristic selects an item that best splits the training set into distinct groups. To this end, the *normalized information gain* is leveraged to compare all available features. The selected feature (i.e., the feature with highest gain) is inserted as a rule into a decision tree, dividing the data set into distinct subsets.

This classification is repeated on the generated subsets until all features have been mapped as rules into the decision tree or none of the remaining features can be used to correctly sub-classify the data. Terminal nodes are inserted into the tree holding the information on branch classification (such as *spam* or *no-spam*).

A more detailed description of this method can be found in Section 3.3.3. Furthermore, Figure 3.2 shows a $C4.5$ decision tree generated from the training set.

Best-first decision tree This decision tree heuristics differs from the `J48` classifier by means of selecting the *ordering* of features used for making a decision.

The *best-first* heuristics [85] seeks to maximally reduce impurity of subsets after each decision.

That is, the features are selected in such a way that the branch-paths are as short as possible, ending in *pure* terminal nodes (all elements of that node have been equally classified) as soon as possible.

Clustering Weka offers a multitude of algorithms for computing clusters. The most widely used method is the k*-means clustering* [59]. Here, the training set is organized in d-dimensional vectors. k randomly selected vectors make up the mean values for *initial* clustering. The remaining vectors are assigned to the closest cluster based on the *Euclidean distance* to the cluster's selected mean value.

These initial clusters are refined by updating the chosen means with their actual values, possibly reassigning elements to clusters with closer distance. This refinement is repeated until no further modifications in the partitioning/clustering of the vectors is observed.

Thus, the training set is always divided into 2 clusters ($k = 2$) containing 13-dimensional vectors ($d = 13$), one dimension per feature.

Model	TP	FN	FP	TN	FPR	Precision	Recall
bayes.NaiveBayes	22	37	29	164	0.15	0.43	0.37
function.SMO	19	40	27	166	0.13	0.41	0.32
lazy.LWL	28	31	33	160	0.17	0.45	0.47
misc.FLR	2	57	9	184	0.04	0.18	0.03
rules.ConjunctiveRule	37	22	68	125	0.35	0.35	0.62
trees.J48	28	31	46	147	0.23	0.37	0.47
trees.BFTree	24	35	35	158	0.18	0.40	0.40
meta.Clustering	30	29	61	132	0.31	0.32	0.50

Table 3.3: Comparison of different classification models

Table 3.3 gives a numerical representation of the classification approaches presented in this section. The first four columns refer to the number of *true/false positives* (TP/FP) and *true/false negatives* (TN/FN), followed by the *false positive rate*. The numbers show, that most of the approaches perform comparably good

in terms of precision and recall, with Fuzzy Lattice Reasoning (misc.FLR) as the only exception.

3.3.3 Evaluation of the J48 decision tree

We chose a decision tree as the classifier because it intuitively presents the importance of the involved features (i.e., the closer to the root a feature appears in the tree, the more important it is). The J48 implementation included in the Weka package, offers various possibilities to tweak the final result. The most interesting parameter is the confidence factor, which allows to tweak the degree of pruning and, therefore, classification accuracy. We found that a value of 0.1 leads to the best results for our data set. The generated tree is shown in Figure 3.2. This tree consists of 21 nodes, 11 of which are leafs. Five features were selected by the algorithm to be useful as distinction criteria between spam and legitimate sites. Additionally, Weka calculates a confidence factor for every leaf, indicating how accurate this classification is.

The most important feature is related to the presence of the search terms on the page (i.e., the query term frequency > 0). Other important features are the domain name, the file path, the number of in-links as reported by Yahoo!, and the PageRank value of the given site as reported by the Google toolbar.

The fact that we want to improve the results by removing spam sites demands a low false positive rate. False positives are legitimate sites that are removed from the results because they are misclassified as spam. It is clearly desirable to have a low number of these misclassifications, since false positives influence the quality of the search results in a negative way. False negatives on the other hand, do not have an immediate negative effect on the search results. If a spam site is misclassified as legitimate, it ends up as part of the search results. Since we are only post-processing search engine results, the site was there in the first place. Thus, false negatives indicate inaccuracies in our classification model, but do not influence the quality of the original search results.

Evaluating the J48 decision tree with our test data set results in the confusion matrix as shown in Table 3.4. The classifier has a false positive rate of 10.8% and a false negative rate of 64.4%. The detection rate (true positives) is 35.6%.

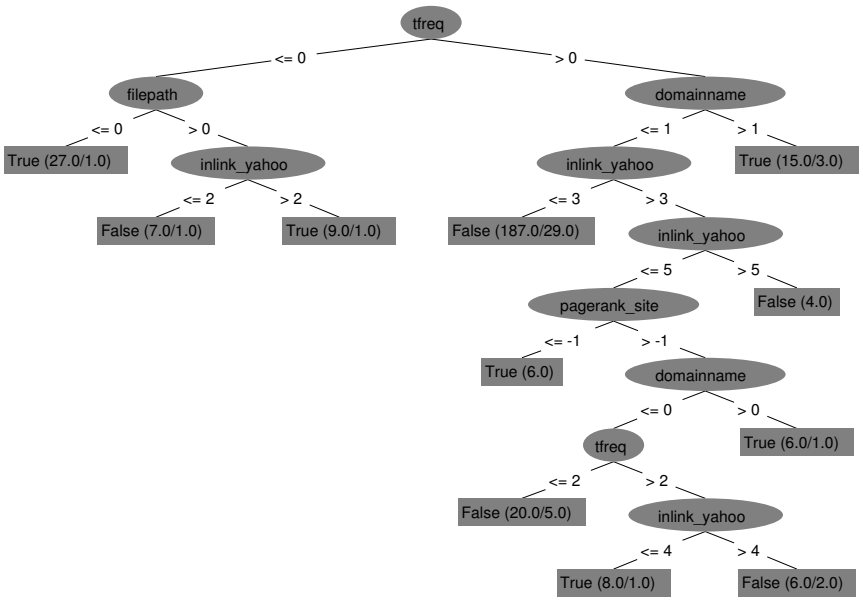

Figure 3.2: Generated J48 decision tree.
The node labels correspond to the feature extractors listed in Section 3.3.1

	Classified as Spam	Classified as Legitimate
Spam	21	38
Legitimate	20	173

Table 3.4: Confusion matrix of the J48 decision tree

Detecting 35% of the unwanted sites is good, but the false positive rate of 11% is unacceptable. To lower the false positive rate, we decided to incorporate the confidence factor that is provided for each leaf in the decision tree. By using this confidence factor as a threshold (i.e., a site is only classified as spam when the confidence factor is above the chosen threshold), we can tune the system in a way that it produces less false positives, at the cost of more false negatives. For example, by using a confidence value of 0.88, the classifier has a false negative rate of 81.4%. However, it produces no false positives for our test set. The true

positive rate with this threshold value is 18.6%, indicating that the system still detects about every fifth spam/malicious page in the search results.

While a detection rate of 18% is not perfect and allows for improvement, it clearly lowers the amount of unwanted pages in the results. Taking into consideration that most users only pay attention to the top 10 or top 20 results of a search query, these 18% create up to two empty slots in the top 10 rankings that can accommodate potentially interesting pages instead.

3.4 Summary

Search engines are a target for attackers that aim to distribute malicious content on their websites or earn undeserved (advertising) revenue. This observation motivated our work to create a classifier that is able to identify and remove unwanted entries from search results. First, we need to understand which features are important for the rank of a page. The reason is that these features are most likely the ones that an attacker will tamper with. To infer important features, we conducted an experiment in which we monitored, for almost three months, the ranking of pages with 30 different combinations of feature values. Then, we computed the weights for the features that would best predict the actual, observed rankings. Those features with the highest weights are considered to be the most important for the search engine ranking algorithm. Based on the features determined in the first step and a labeled training set, we generated a classifier (a J48 decision tree). This decision tree was then evaluated on a test data set. The initial evaluation resulted in 35% detection rate and 11% false positives. By taking into account the confidence values of the decision tree and introducing a cutoff value, the false positives could be lowered to zero. At this rate, almost one out of five spam pages can be detected, improving the results of search engines without removing any valid results. Furthermore, we also compared the performance of the J48 decision tree with seven other machine learning algorithms.

In this chapter we have shown how attackers can influence the ranking of their web sites in search engine results. Many malicious pages that are pushed in this way perform so-called *drive-by download* attacks. That is, the mere visit of such

a page can lead to an infection with malware. Chapter 4 will elaborate in detail on this threat and a countermeasure technique we developed.

Chapter 4

Drive-by Download Attacks

A *drive-by download* attack denotes a download and subsequent execution of software that happens without the knowledge and consent of the user. Unfortunately, drive-by downloads present a major threat to the Internet and its users [75]. In a typical attack, the mere visit of a web site that contains the malicious content can lead to the infection of a user's computer with malware. The malicious code that is installed as part of the attack then has typically full control over the victim's machine. Often, keystrokes are recorded, passwords are stolen, and sensitive information is leaked out. Also, infected computers may join a botnet [21], a large collection of compromised hosts controlled by the attacker. The computational power of compromised hosts are valuable for attackers as these hosts can be misused for spam campaigns [54] or denial of service attacks [63].

The typical steps of a drive-by download attack are shown in Figure 4.1. It can be seen how the attacker first prepares a web site with malicious content. When this site is later visited by a victim, hostile script code is downloaded and executed by the victim's browser. This script exploits a vulnerability in the browser or an installed browser-plugin. Once successful, the payload (shellcode) of the exploit downloads malware that provides full control to the attacker.

Most current drive-by downloads target browser plug-ins that are developed and distributed by third parties [75, 76]. The reason is that these plug-ins are less tested than the core browser, and thus, more likely to contain security relevant vulnerabilities. Also, plug-ins are typically distributed as binary executables (at least

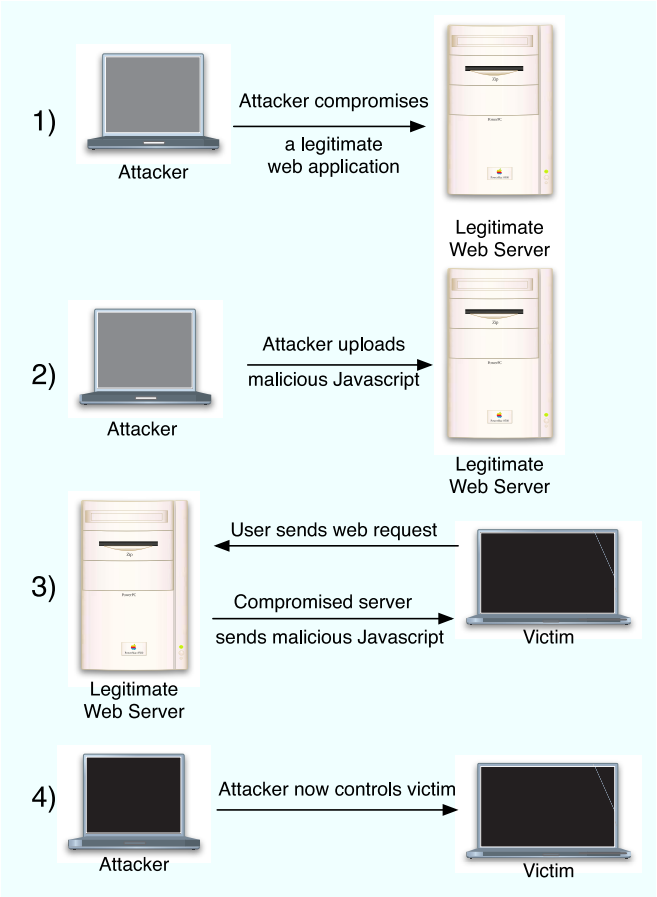

Figure 4.1: The typical steps of a drive-by download attack.

in the case of Microsoft's Internet Explorer). As many plug-ins are written in non-safe languages such as C, they are susceptible to a wide range of vulnerabilities that are common for applications written in such languages. These vulnerabilities include buffer overflows, memory corruption issues, and pointer overwrites. Finally, plug-ins are often executed in the context of the browser, and as a result, can get full access to the browser and the underlying operating system.

As mentioned previously, as part of a drive-by download, attackers use client-side scripting code to load the shellcode (payload) into memory and execute the exploit against a vulnerable component. More precisely, JavaScript [36] is typically used to assign the binary representation of shellcode to a variable that is stored in the address space of the browser. To make their exploits more reliable, attackers resort to a technique called *heap spraying* [23, 88]. Heap spraying cre-

ates multiple instances of the shellcode, combined with a NOP sledge [96][1]. By leveraging the knowledge of how a script engine manages its heap memory, an attacker can, to a certain extent, influence where variables are stored in memory. As a result, the area of heap memory that needs to be sprayed for an attack to succeed is reduced. Once the heap memory has been "prepared," the actual exploit is launched. To this end, the hostile script typically invokes a vulnerable method (of a plug-in) with malicious arguments.

When the attacker has prepared a malicious script that can launch a drive-by download, it can be placed on a web site. Then, the attacker has to ensure that potential victims visit this site. One way is to create a new site and manipulate search engines so that they list this site high in their rankings. The higher a page is ranked, the higher the chance is that an unsuspecting Internet user will visit it. Another approach is to embed malicious content in advertisements that are placed on legitimate web pages. Here, the site embedding the advertisement becomes an unknowing accomplice for distributing the attack. Moreover, an attacker can also take advantage of vulnerabilities found in popular web applications. By exploiting these applications, they are able to place their content directly on the vulnerable web site. Automated SQL injection attacks [22, 50], for example, modify the database back-ends of web applications in order to include `iframe` tags that load the malicious pages.

Drive-by attacks belong to the most common methods for spreading malware today [76]. Thus, it is important to find solutions that mitigate the problem and protect users. In this chapter, we present a proof-of-concept implementation of a system that detects shellcode-based drive-by download attacks. Our basic idea is to check the variables (strings) that are allocated by the browser (the script engine) when executing client-side scripts. When such a variable contains shellcode, we assume that the script is hostile, attempting to setup the environment for an exploit. Thus, the script is terminated, before any vulnerable function is invoked. We implemented our system in the Mozilla Firefox browser. However, our conceptual solution is general and works for arbitrary browsers.
The main contributions presented in this chapter are as follows:

[1] A NOP sledge consists of a sequence of NOP instructions that increase the chance of successfully hitting the shellcode when hijacking the control flow of the vulnerable application.

- We propose a technique that uses emulation to automatically identify shellcode-based drive-by download attacks in a browser.

- We describe a proof-of-concept implementation of our approach that is integrated into the Mozilla Firefox browser.

- We present experimental results that show the feasibility of our approach. We have evaluated our prototype on more than one thousand malicious and several thousand benign sites. Our experimental results demonstrate that the system is able to accurately detect drive-by downloads with no false positives.

4.1 Anatomy of a drive-by attack

In this section, we first provide a short overview of JavaScript to enable the reader to understand script-based drive-by downloads. Then, we present and discuss a real-world attack to illustrate the problem that we aim to defend against.

4.1.1 JavaScript basics

JavaScript is an implementation of the ECMA-262 standard that defines an object-oriented scripting language [36]. The JavaScript specification defines a set of core components, such as data types (e.g., `String`, `Integer`, `Object`), special objects (e.g., `Date`, `Math`), and operators. The most prominent use of JavaScript is for supporting dynamic content on the client-side (in web browsers). However, JavaScript is also often embedded in other software, such as Adobe's Acrobat PDF reader. Systems that use JavaScript typically provide *environments* that allow a script to interact and communicate with other components. The document object model (DOM), for example, is part of the environment provided by the web browser. It allows scripts to manipulate the web pages that are displayed and to react to user actions and inputs.

The JavaScript interpreter of the Mozilla foundation is called SpiderMonkey [36]. Microsoft's implementation of ECMA-262 is called JScript [66]. This

implementation adds facilities to the environment that allow a script to instantiate and communicate with ActiveX components [65]. These components are the preferred way of providing plug-ins for the Internet Explorer. On request, the libraries implementing the components are loaded into the address space of the Internet Explorer process, and the necessary objects are instantiated. ActiveX plug-ins, thus, have the same privileges that the browser has, often including full access to the file system and the network.

Among the data types, `strings` deserves special attention. ECMA-262 defines strings as sequences of 16-bit integers, commonly interpreted as UTF-16 characters. Popular JavaScript engines, such as SpiderMonkey, implement strings as immutable. That is, once a string variable is initialized, the value does not change for the rest of its lifetime. String operations, such as substituting characters (i.e., `replace` method of the string object), do not modify the original value. Instead, a new additional string variable is instantiated with the modified content. We will see that this fact has important ramifications for the implementation of our defense technique.

4.1.2 An example of a real-world drive-by download

In this section, we describe a typical drive-by download attack. We actually encountered this specific attack during our experiments. On September 2, 2008, our high-interaction client honeypot visited `http://www.thewebleaders.com`. This page contained an `iframe` that loaded the script presented in Listing 4.1.

The most noticeable property of the script is that it uses obfuscated variable and function names to make it difficult for a human analyst to understand the script's purpose. Manual analysis reveals that the function defined in Line 1 serves as a decryption routine. The two values that make up the key for decryption are the location currently visited by the browser (`location.href`, Line 2), and the source code of the decryption function itself (`arguments.callee`, Lines 3,4). Using the current location as part of the key to the decryption function allows the attacker to prevent the analysis of the script when it is loaded from a different location. That is, when the script is captured, and during a later analysis served

```
1       function XfNLVA421(IaP1EoKdg) {
2               var I833Nad64 = location.href;
3               var hOtmWAGmO = arguments.callee;
4               hOtmWAGmO = hOtmWAGmO.toString()
5               ...
6               try {
7                       eval(jiiIUpFi3);
8               } catch(e)
9               ...
10      }
11      XfNLVA421('a7A7a7A7ac9bB5b261...');
```

Listing 4.1: Excerpt of an obfuscated, real-world malicious script.

```
1   function IxQUTJ9S() {
2     if (!Iw6mS7sE) {
3       var YlsElYlW = 0x0c0c0c0c;
4       var hpgfpT9z = unescape("%u00e8%u0000% ...");
        ...
5       for (var CCEzrp0s=0;CCEzrp0s<Wh_74Nkm;CCEzrp0s++) {
6         je9rIXgu[CCEzrp0s] = QdV7IGyr + hpgfpT9z;
7       }
        ...
9   }
    ...
11  var Kp1uYOjP = new ActiveXObject('Sb.SuperBuddy');
12  if (Kp1uYOjP) {
13        IxQUTJ9S();
14        oH9mUjOd(9);
15        Kp1uYOjP.LinkSBIcons(0x0c0c0c0c);
```

Listing 4.2: Excerpt of a real-world, decrypted malicious script.

locally, the decryption will fail. The last step of the function uses the decrypted content in an `eval`[2] statement (Line 7). Nesting the `eval` in a try-catch block suppresses the errors that would be seen by the analyst if the `eval` would fail. This failure would happen, for example, in case the key is wrong.

[2]ECMA-262 specifies that an implementation must provide an `eval` function. This function takes an argument of type string and interprets its argument as an ECMAScript program. That is, the `eval` function executes the argument it receives as a script.

After decryption, the string passed to `eval` contains the code excerpt presented in Listing 4.2. Line 4 loads x86 shellcode into variable `hpgfpT9z`. Subsequently, the heap is sprayed by filling the memory with a large number of strings that contain a NOP sledge and a copy of the shellcode (Lines 5-7). In Line 11, the SuperBuddy ActiveX component is instantiated. If a valid object can be created, then the vulnerable method `LinkSBIcons` is invoked (Line 15). The `LinkSBIcons` vulnerability is known as CVE-2006-5820 [20]; the argument of `LinkSBIcons` is used as a function pointer, thus diverting control flow to the sprayed heap.

4.2 Automatically detecting drive-by attacks

As described in the previous section, drive-by downloads that target memory corruption vulnerabilities have to prepare the environment before they can successfully launch their exploits. To this end, they use client-side script code to allocate (often large numbers of) strings that are filled with x86 (shell)code. The key idea of our detection approach targets precisely this behavior. More specifically, to detect drive-by downloads that exploit memory corruption vulnerabilities, we monitor all strings that are allocated by the JavaScript interpreter. These strings are checked for the presence of shellcode. Of course, all checks occur *before* a vulnerability can be abused to redirect control flow to the shellcode. When our system detects that a script creates a string that contains shellcode, the execution of the script is stopped.

The prototype implementation of our detection technique was implemented and integrated into the Mozilla Firefox browser and SpiderMonkey, its JavaScript engine. We chose Firefox as our prototype platform as this is an open source browser. Obviously, we would have liked to have integrated our solution into the Internet Explorer. Unfortunately, we did not have access to the source code. Nevertheless, we note that our solution is conceptually generic, and is not dependent on a specific browser. We have chosen to target JavaScript because it is by far the most common language for writing scripts on the web. Of course, an attacker could make use of a different language than JavaScript to deliver an exploit (and

some indeed use Visual Basic Script). However, it would be straightforward to include our technique also into other script engines.

In the following sections, we describe our technique in more detail. In particular, we discuss how we keep track of the strings that are allocated, and how we detect the shellcode that an attacker may attempt to inject. Then, we discuss two optimizations that are applied to make the proposed approach fast enough to be used in practice.

4.2.1 Tracking object (string) allocations

For a drive-by attack to succeed, it is important that the bytes constituting the shellcode are stored at successive addresses in memory. Otherwise, these bytes would not be interpreted as valid x86 instructions. Of course, one could consider to split a sequence of instructions over multiple segments and connect these segments with jumps, but at least the bytes of each segment need to be consecutive to be valid. In JavaScript, the only way to guarantee that bytes are stored in a consecutive manner is by using a string variable. Here, consecutive characters of the string are allocated in adjacent memory locations.

To detect the shellcode that a malicious script might construct on the heap, we have to keep track of all string variables that the program allocates. To this end, we modified the SpiderMonkey JavaScript interpreter that is embedded in Firefox. More precisely, we added code to all points in the interpreter where string variables are created. These points were found at three locations: one for the allocation of global string variables, one for local string variables, and one for strings that are properties (members) of objects. The code that we added simply keeps track of the start address of a new string variable and its length. Here, it is important to recall that strings in JavaScript are immutable. As a result, whenever a character in a string is modified, or when two strings are concatenated, the resulting string is created in a new memory area. Thus, string manipulation is automatically handled by the code introduced for creating a new string variable.

In addition to the start address and the length of new string variables, we also keep track of the two sub-strings that are used in a string concatenation operation. That is, whenever a new string is created as a result of a concatenation operation,

we keep pointers to the sub-strings. This is needed for an optimization that is discussed later.

An attacker might consider to use integer arrays to store the shellcode in successive memory addresses. However, JavaScript supports arrays of integers that follow this (packed) memory layout only for 31-bit values, where the remaining bit is always set to indicate that the value is an integer. The fact that one bit is set in each four-byte integer makes it more difficult for the attacker to craft his shellcode. Also, support for packed integer arrays can be easily disabled. For integer values that are larger than 31-bit, and for all other data types, JavaScript handles arrays differently. More precisely, such arrays only store identifiers (pointers) that reference objects that are allocated elsewhere. Since these objects contain additional management information and are allocated from a pool of memory, it is very difficult for an attacker to reliably predict where these objects will end up. As a result, our system focuses on the content of string variables. Of course, when attackers develop techniques to store shellcode in objects that are allocated in the object pool, we could easily add checks for these objects as well.

4.2.2 Checking strings for shellcode

Given information about the addresses and lengths of the strings in memory, the next question that needs to be answered is how shellcode can be automatically detected within these strings. More precisely, we have to discuss how shellcode can be recognized, and the points in time when this analysis is launched.

For the detection of shellcode, we are leveraging *libemu* [6]. libemu is a small library written in C that offers basic x86 emulation and shellcode detection. It is efficient in detecting shellcode and being used in projects such as Nepenthes and Honeytrap. To recognize shellcode in a string (character buffer), libemu checks, starting from each character, whether there is a sequence of valid instructions of sufficient length. When such an instruction sequence is found, libemu reports shellcode. Since most bytes can be disassembled to valid x86 instructions, libemu also uses a number of heuristics to discriminate random instructions from actual shellcode. We currently use a value of 32 bytes as the threshold for the minimal length of a shellcode sequence. We found that this value works well in our experi-

ments, and it is also significantly shorter than all Windows shellcode encountered in the wild [73].

Note that an attacker might try to evade detection by distributing shellcode fragments over multiple strings. In this case, to be successful, each fragment must end in a jump instruction to the next fragment. Moreover, since the total length of each fragment must not exceed 32 bytes, there is almost no space for a NOP sledge. As a result, the attacker must guess the jump offset quite precisely. While modern heap manipulation techniques allow for a certain control over the heap layout, we believe that such an attack is very difficult to launch in practice. Moreover, randomizing the allocation of individual objects in the heap would be easy to do and render this hypothetical evasion vector infeasible. Note that randomizing object allocations does not help against current drive-by attacks that store the complete shellcode in one string. The reason is that the location of a particular string might not be know precisely, but the attacker can allocate thousands of such self-contained, malicious strings (sometimes worth tens or hundreds of megabyte). Then, hitting a single string is sufficient to successfully run the shellcode.

The goal of our detection approach is to ensure that the attacker *cannot* execute shellcode *before* we analyze all (string) objects that he has created. The straightforward approach to do this is to invoke the emulator whenever a new string object is created. Of course, every string object is only checked once. Nevertheless, this naive approach incurs a significant performance penalty.

4.2.3 Performance optimizations

To reduce the performance penalty that is incurred when checking every string that is allocated, two approaches are possible. First, one can reduce the total number of invocations of the emulation engine. Second, one can reduce the amount of data that the emulator needs to inspect. Our prototype supports techniques to leverage speedups from both of these approaches.

Since vulnerabilities exploited by drive-by download attacks are almost always found in the browser or its plug-in components, we consider the JavaScript interpreter as safe. As a result, while executing JavaScript core functionality, a

script is allowed to create string objects without checks, even ones that contain shellcode. To transfer control flow to such a string buffer with shellcode, the malicious script must exploit a vulnerability in an "external" component, leaving the JavaScript core part. Thus, to detect any shellcode before it can be executed, it is not required to perform emulation immediately after creating a new string object. Instead, it is possible to only record information on all created string objects, and postpone emulation to the time at which control flow leaves the interpreter, entering an external component or the browser.

The delayed checking allows us to collect information about the involved string objects and leverage this knowledge to decrease the overall amount of data that has to be checked. First, we use information about string concatenation, a frequent operation. Consider that we observe the fact that a given string `a` consists of the concatenation of strings `x` and `y`. This allows us to skip the analysis (emulation) of `x` and `y` when `a` was already scanned and found to be clean. A second venue for optimization is the JavaScript garbage collector. By invoking garbage collection on every transition from the interpreter to the environment, we are able to discard all objects from the emulation that are freed by the garbage collector. We have modified the garbage collector routines to remove the freed contents from the list of objects to emulate (after zeroing their content).

Note that although the detection is delayed, it is still complete in the sense that no machine instructions residing in the memory space of a JavaScript object can potentially be executed before being checked by our shellcode detector.

4.3 Evaluation

This section discusses how we evaluated our prototype as well as the experimental results. The evaluation was carried out in three parts. First, we evaluated our system for false positives by accessing a large number of popular benign web pages. Second, we used our system on pages that launch drive-by downloads and evaluated the detection effectiveness. Third, we examined the performance overhead of our system.

4.3.1 False positive evaluation

In the context of our system, a false positive is a page that is detected as malicious without actually loading shellcode to memory. To evaluate the likelihood of false positives, we extended our prototype system to visit a list of $k = 4,502$ known, benign pages. These pages were taken from the Alexa ranking of global top-sites, and simply consisted of the top k pages. We consider this to be a realistic test set that reflects a wide range of web applications and categories of content.

For the batch evaluation of URLs, we implemented a Firefox extension that visits all URLs provided in a file. After a timeout, the extension automatically visits the next URL in the list. More precisely, the extension moves to the next URL two seconds after the page finished loading, or ten seconds after page loading started. The hard limit of ten seconds was necessary to handle scripts that continuously issued page reloads.

Our prototype did not produce any false positives for this dataset. This might look suspicious at a first glance: The x86 instruction set is known to be densely packed, thus, almost any sequence of bytes makes up valid instructions. However, one has to consider the fact that JavaScript uses 16-bit Unicode characters to store text. That is, even if a given sequence of ASCII characters results in a valid x86 instruction most of the time, the JavaScript representation of the same characters most likely does not, since every other byte would contain the value 0x0. Of course, an attacker can encode the shellcode appropriately. However, benign pages typically do not contain strings that map to valid instruction sequences.

4.3.2 Detection effectiveness

In a next step, we evaluated the capabilities of our technique to identify drive-by attacks that rely on shellcode to perform their malicious actions. To this end, we evaluated our system on the traces of 1,187 web browsing sessions that are known to contain drive-by attacks. These traces were collected by visiting URLs that are advertised in spam emails. We retrieved a list of such URLs from the Spamcop [90] web service, as well as from mails collected in the spam trap of a medium-size security company.

To filter those URLs that actually host drive-by attacks, we used the Capture Honeypot Client (HPC) [11]. Capture visits the URLs with a browser on a virtual machine (VM). After a site is loaded, the state of the VM is inspected, and all modifications to the file system and registry as well as new processes are logged. In addition to the logged information, the system records a trace of all network communication that was taking place. Capture simply visits a URL in a browser and performs no additional actions. Thus, by filtering URLs that caused a new process to be launched, we were able to identify those sites that perform drive-by attacks. The system running in the VM was a Windows XP Professional (Service Pack 2) installation. No additional security patches were applied, and automatic updates were turned off. Additionally, the Flash and QuickTime plug-ins were installed.

Once a URL was identified to host a drive-by attack, we used Chaosreader [43] to extract application level data from the network traces. Chaosreader is able to recognize a variety of application data from network traces. Among others, Chaosreader identifies HTML documents, binary images, or gzip compressed data, saving each response to an HTTP request in a distinct file. Files that were found to be compressed were decompressed before continuing.

Extracting a single file for every response to an HTTP request made further post-processing necessary. For example, if an HTML page references a JavaScript URI via an `src` attribute of a `script` tag, this results in another request in which the browser fetches the JavaScript. The response contains only JavaScript code without surrounding HTML tags. Visiting such a file in a web browser results in its contents being interpreted as text, and thus, no interpretation of the code takes place. We used a simple heuristics to add the necessary HTML and `script` tags to such files. More precisely, whenever a file does not already contain valid HTML, and it does contain any of the most used JavaScript reserved words (e.g., `function`, `var`), it is wrapped in appropriate tags.

Once the HTML and JavaScript files were restored, they were uploaded on a local web server. In total, 11,910 URLs (files) were associated with the 1,187 traces. Note that a trace can contain multiple resources that are accessed by the browser. For example, redirection chains or embedded content. Our prototype system was instructed to visit each of these URLs. The modifications required to

process encrypted attack scripts are detailed in Section 4.4. One might ask why we did not simply use our system to visit malicious pages that are live on the Internet, but, instead, replicate malicious sites and their scripts locally. The reason is that malicious sites on the Internet are frequently taken down. Additionally, many malicious sites only perform attacks on the first visit of a client. Thus, changing the prototype and revisiting the same location could not detect attacks hosted on such pages. In our setup, we have created a corpus that allows us to replicate our experiments and better debug and understand cases in which the detection fails initially.

When running our prototype detection system on the resources associated with 1,187 traces, we detected 956 instances of shellcode. This yields an initial detection effectiveness of 81%. We then examined the remaining 231 traces to understand why our system did not detect shellcode while the Capture honeypot client indicated an attack.

Manual analysis revealed four main causes that result in our prototype failing to detect a threat. One group (with 62 traces) contains drive-by downloads that do not make use of memory exploits. In particular, a popular attack against the Sina Downloader ActiveX component exploits insecure component behavior. More precisely, this component contains functions that allow a script to download a file and to start a program. This makes it trivial for an attacker to download malware and start it, without ever corrupting memory. However, note that this attack targets an old vulnerability (from 2006) that is very specific to a particular component. Thus, it is not a general class of vulnerabilities that our approach misses, but a specific problem in a component that basically offers all the functionality required by the attacker.

The second group of attacks (with four traces) that were missed by our system are due to exploits that use Visual Basic (VB) script code to prepare the environment and launch the exploit. As mentioned previously, our current prototype only instruments the JavaScript engine. However, similar techniques could easily be added to the VB script engine.

The third group of missed attacks (with 127 traces) are due to the way our experiments are carried out. Recall that we do not visit live pages on the Internet, but invoke individual resources (files) that we extracted from network traces. In

some cases, the malicious code is distributed over several scripts that are in different files. In these cases, the browser does not see and analyze the complete, malicious script at once. This typically leads to JavaScript errors, and failure to inject shellcode into the heap. This, however, does not reflect a deficiency in our approach. If these sites were visited with a browser protected by our proposed technique, all scripts would be fetched and executed by the web browser in the same context, thus, allowing to detect the threat.

Finally, a forth group (with 38 traces) was not recognized as malicious because it contains traces that were false positives of the Capture honeypot client. More specifically, they were `.cab` archive files. Whenever a `.cab` file is downloaded, Windows automatically starts the Windows Management Instrumentation to handle this resource. While this activity results in a new process being launched, it is not because of a malicious drive-by download but due to legitimate activity. However, Capture considers the start of a new process as an indication of a successful attack.

Given the discussion of the four cases above, we argue that only the traces associated with attacks against the Sina Downloader ActiveX and similar components should be considered false negatives for our system. As a result, we can compute a detection rate of $\frac{956}{956+62} = 93.9\%$. Also, we observe that we detected all drive-by attacks that exploited a memory corruption vulnerability, which is by far the most common type of exploit found in the wild.

After evaluating the detection capabilities of our system, we also performed further analysis of the ActiveX components created by the malicious scripts. Our results show that most malicious sites perform their attacks through only a handful of vulnerable components. Figure 4.2 depicts a breakdown of the distribution of the involved components. It is interesting to observe that the two most prominent components (SuperBuddy and QuickTime viewer) account for almost 50% of the targets of the attacks. Note that the figure lists the 1,688 ActiveX components that were created during our evaluation. Nonetheless, not every created component lead to a successful exploit.

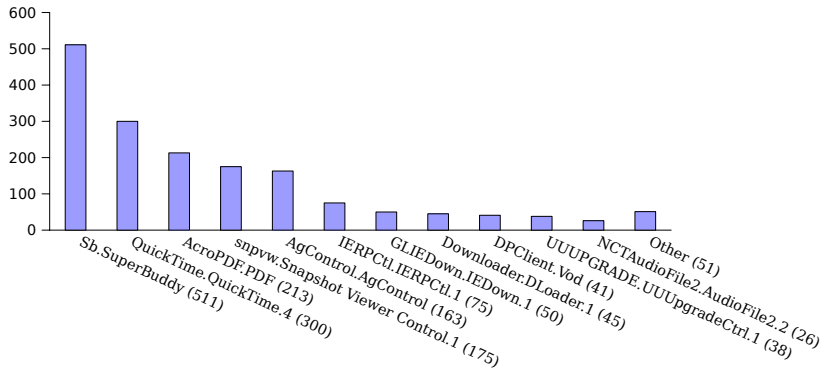

Figure 4.2: ActiveX components involved in drive-by downloads.

4.3.3 Performance

Our approach uses x86 instruction emulation to detect shellcode within JavaScript strings. This happens online; that is, the analysis must be performed at the time the browser loads a page. Since emulation is a resource intensive task, careless invocations of the emulator may lead to a significant performance overhead. We have pursued several strategies to minimize the overhead, as explained in Section 4.2.3. In this section, we describe the results of our performance evaluation.

Our experiment measures the wall-clock time required to load a set of web pages. We have chosen the 150 most popular web sites (according to Alexa). The same set of pages was processed three times. First, we ran an off-the-shelf Mozilla Firefox browser without performing any additional tasks. Second, we used our modified version of the browser that provides protection against drive-by download attacks, without any performance optimizations. Third, we used the browser with protection and performance optimizations.

All measurements have been carried out on a machine with an Intel Core 2 Duo processor running at 2.66 GHz and 4 GB of main memory. Internet connectivity was established using an ADSL line with a bandwidth of 1 MBit/s.

The results of our performance evaluation are presented in Table 4.1. On average, an unmodified Firefox browser took 3.51 seconds to load one web page from our testing set. This time includes the download of the content over the Internet, parsing and rendering of the page, and execution of all JavaScript code. In comparison, a modified version of the browser, which provides protection against

	Total time[s]	Time/page[s]	Overhead/page	Factor
Off-the-shelf browser	527	3.51		
Protected browser				
w/o optimizations	1,237	8.25	4.74	2.35
w/ optimizations	876	5.84	**2.33**	**1.66**

Table 4.1: Page load times (sec) with and without drive-by download protection.

drive-by download attacks, takes additional time. The overhead can be attributed to the effort spent on tracing the allocated string objects, and more importantly, emulation of their content when executing functionality from the JavaScript environment. A basic implementation of our system, without application of performance optimization measures, took 8.25 seconds per page. This is a significant performance penalty. Our final implementation, including all optimizations took, 5.84 seconds per page. That is, the overhead of the naive version could be reduced in half.

Browsing the Web is an interactive occupation, and it is desirable for the user to experience as little latency as possible when loading a new page. Obviously, the decrease in performance introduced by our approach seems significant. However, note that the time users typically spend on consuming the downloaded content (e.g., reading an article) by far outweighs the time that is spent on waiting for new content to be loaded. Thus, we believe that the benefit of a secure browsing experience, without the risk of falling prey to a drive-by download attack, well compensates the inflicted performance penalty.

4.4 Implementation details

As mentioned previously, our system has been implemented by extending Mozilla Firefox and SpiderMonkey. However, all drive-by download attacks in our dataset target the Internet Explorer (IE). The astute reader might wonder how our system can actually detect such attacks, since they are not supposed to work with Firefox. In the following, we provide some (what we believe) interesting details on how we implemented our system to detect IE attacks with a modified Firefox browser.

Of course, when our technique would be integrated with Internet Explorer, such extensions would not be necessary. Also, the system as introduced can readily detect drive-by downloads that target Firefox. Moreover, we discuss some additional issues that needed to be addressed because of our experimental setup.

Simulating ActiveX components.

Attacks that aim to exploit a vulnerability in a specific plug-in often perform a check for the availability of this plug-in. That is, such attacks only reveal their malicious behavior when the vulnerable component is present. In the case of ActiveX plug-ins, this is done by trying to instantiate the vulnerable component. If the plug-in object is instantiated successfully, it usually means that the component is present.

Unfortunately, Mozilla Firefox does not support ActiveX plug-ins. However, as most drive-by attacks rely on ActiveX to be present, we had to modify the browser appropriately. More precisely, we extended Firefox such that it creates dummy objects for instantiation requests to ActiveX components. Thus, whenever a malicious script attempts to instantiate an ActiveX component, the call succeeds and the corresponding dummy object is created.

These objects accept all method invocations, and also log method calls together with their respective arguments. Note that although it is not the main focus of our work, this information can be used to identify the vulnerability that is used to divert the control flow.

Browser fingerprinting.

Browser fingerprinting is a technique applied by attackers to serve only exploits that match the specific browser of the sites' visitors. To this end, instead of bluntly trying a series of attacks, a script is executed to determine the browser, its version, and installed plug-ins. Based on the knowledge gathered by this script, it fetches only those exploit scripts that match this setup (e.g., if no QuickTime plug-in is detected, no QuickTime related exploits are tried). Even when no fingerprinting is performed as described above, the malicious script most likely verifies that it is executed in a browser that it intends to exploit. Therefore, the script queries

```
1       try {
2               ...
3       } catch (e) {};
4       finally {
5               ...
6       }
```

Listing 4.3: Illustration of different parsing behavior.

the properties of the `navigator` object and only continues if the information matches its authors' intentions. Since our prototype is implemented in Mozilla Firefox, this would have prevented all scripts that perform such techniques from executing. However, the recorded traces hold proof of a successful drive-by attack. Thus, we modified our prototype to pretend to be the same browser and version[3] that was used when the traces were recorded.

To assure that the script is executing in Microsoft's Internet Explorer, attackers rely on inaccuracies of the JScript parser. More precisely, the JScript parser is more tolerant with regards to semicolons than SpiderMonkey.

Listing 4.3, for example, illustrates this with a try-catch-finally construct. While the JScript parser gladly accepts this syntax (notice the semicolon after the catch block in Line 3), the SpiderMonkey engine terminates the script with an error (i.e., "finally without try") at Line 4. These different parsing behaviors introduce further means for an attacker to make sure the script is interpreted by the Internet Explorer. As we could observe such attacks in the wild, we had to modify the parser of our prototype to reflect the behavior of the JScript parser.

Dynamic encryption keys.

Most malicious scripts are encrypted in some way. The attackers' motivation to disguise malicious scripts is obviously the intention to encumber the analysis of such scripts. Encryption is a straightforward approach to do so.

An encrypted script contains a decryption routine and a cipher text. During execution, the cipher text is decrypted by the routine, and the result is executed via

[3]Corresponding to the user-agent string: `Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)`

JavaScript's `eval` function. Two possibilities exist where the decryption routine derives the correct key from. (1) the key might be part of the script itself (e.g., stored in a variable), or (2) the key is dependent on the environment of the script. While in the first case, decryption is automatically handled by the interpreter, the second case requires that the environment presents the right information for the queried value. In our evaluation dataset, many decryption keys were partly derived from the current URL of the browser. Since the scripts were hosted at a local web server, the URLs were different, thus leading to wrong decryption keys. For wrong key values, the decryption routines produce only garbage and, as a result, no malicious behavior can be observed. Since, on the other hand, the values were correct when the network traces were recorded, we modified our prototype to report the URL that was visited during the recording of the trace as the current location. This allowed the scripts to decrypt the cipher text correctly, and we were able to analyze and detect their malicious behavior.

Batch processing time-outs.

Some malicious scripts use the `setTimeout` function of JavaScript to delay their actions. During our batch processing of URLs, we use a time-out of ten seconds before moving to the next page. As a result, the usage of such timers could prevent detection. To mitigate this problem, we had to assure that these timeouts expire before the batch processing extension moves to the next URL. To this end, we modified Firefox to replace all delays of `setTimeout` calls with a delay of 50ms.

Interestingly, during our evaluation, we encountered a malicious script that implemented a custom version of a setTimeout-equivalent function. More precisely, the script looped and measured the expired time between the initial run of the loop and the current time. Once the desired delay was reached, execution continued. This sample did not use the `setTimeout` function and thus, the extension switched to the next URL before the malicious content was executed. Notice, however, that not detecting the malicious script in this sample is an artifact of the batch processing and does not indicate a weakness in our proposed approach. In

fact, after removing the sleep function, the system did detect the malicious script, the shellcode it used, and the involved ActiveX components.

4.5 Summary

Drive-by downloads belong to the most threatening vectors of attack that are currently used by cyber-criminals to illegitimately gain control of victims' computers. In this chapter, we present a novel approach that helps protect a user against drive-by attacks that rely on shellcode.

Our system is integrated into the web browser where it monitors JavaScript code that is downloaded and executed. More precisely, our system traces all string objects that are created during run-time, and it uses x86 instruction emulation to determine whether a string buffer contains executable shellcode. The detection of the shellcode takes place before a vulnerability can be exploited (and control flow redirected). Hence, an attack can be mitigated before the security of the browser is compromised.

Our approach includes optimizations to assure a reasonable performance overhead while delivering excellent detection results for drive-by attacks that exploit binary vulnerabilities in browser plug-in software. We have built a prototype implementation with which we have verified the capability of our approach to successfully detect real-world drive-by download attacks. Our evaluation shows that our approach is feasible in practice.

Although drive-by download attacks are one of the most common infection vectors for malware, they are not the only possibility. Social engineering, for example, can also be used to trick users into voluntarily installing malicious programs on their devices. No matter how a malware program was installed on a device, anti-virus companies and researchers alike need access to the appropriate tools and techniques to analyze such malware samples. This analysis allows them to assess the threat the malware poses to the user and her privacy, and is a necessary precursor to develop effective countermeasures. Therefore, the following chapter presents our novel contribution in the field of static analysis of privacy threatening iOS applications.

Chapter 5

Analyzing iOS Applications for Privacy Threats

Mobile phones have rapidly evolved over the last years. The latest generations of smartphones are basically miniature versions of personal computers; they offer not only the possibility to make phone calls and to send messages, but they are a communication and entertainment platform for users to surf the web, send emails, and play games. Mobile phones are also ubiquitous, and allow anywhere, anytime access to information. In the second quarter of 2010 alone, more than 300 million devices were sold worldwide [40].

Given the wide range of applications for mobile phones and their popularity, it is not surprising that these devices store an increasing amount of sensitive information about their users. For example, the address book contains information about the people that a user interacts with. The GPS receiver reveals the exact location of the device. Photos, emails, and the browsing history can all contain private information.

Since the introduction of Apple's iOS[1] and the Android operating systems, smartphone sales have significantly increased. Moreover, the introduction of market places for apps (such as Apple's App Store) has provided a strong economic driving force, and tens of thousands of applications have been developed for iOS and Android. Of course, the ability to run third-party code on a mobile device

[1] Apple iOS, formally known as iPhone OS, is the operating system that is running on Apples' iPhone, iPod Touch, and iPad products.

is a potential security risk. Thus, mechanisms are required to properly protect sensitive data against malicious applications.

Android has a well-defined mediation process that makes the data needs and information accesses transparent to users. With Apple iOS, the situation is different. In principle, there are no technical mechanisms that limit the access that an application has. Instead, users are protected by Apple's developer license agreement [3]. This document defines the acceptable terms for access to sensitive data. An important rule is that an application is prohibited from transmitting any data unless the user expresses her explicit consent. Moreover, an application can ask for permission only when the data is directly required to implement a certain functionality of the application. To enforce the restrictions set out in the license agreement, Apple has introduced a vetting process.

During the vetting process, Apple scrutinizes all applications submitted by third-party developers. If an application is determined to be in compliance with the licencing agreement, it is accepted, digitally signed, and made available on the iTunes App Store. It is important to observe that accessing the App Store is the only way for users with unmodified iOS devices to install applications. This ensures that only Apple-approved programs can run on iPhones (and other Apple products). To be able to install and execute other applications, it is necessary to "jailbreak" the device and disable the check that ensures that only properly signed programs can run.

Unfortunately, the exact details of the vetting process are not known publicly. This makes it difficult to fully trust third-party applications, and it raises doubts about the proper protection of users' data. Moreover, there are known instances (e.g., [101]) in which a malicious application has passed the vetting process, only to be removed from the App Store later when Apple became aware of its offending behavior. For example, in 2009, when Apple realized that the applications created by `Storm8` harvested users phone numbers and other personal information, all applications from this developer were removed from the App Store.

The goal of the work described in this chapter is to automatically analyze iOS applications and to study the threat they pose to user data. As a side effect, this also shines some light on the (almost mysterious) vetting process, as we obtain

a better understanding of the kinds of information that iOS applications access without asking the user. To analyze iOS applications, we developed PiOS, an automated tool that can identify possible privacy breaches.

PiOS uses static analysis to check applications for the presence of code paths where an application first accesses sensitive information and subsequently transmits this information over the network. Since no source code is available, PiOS has to perform its analysis directly on the binaries. While static, binary analysis is already challenging, the work is further complicated by the fact that most iOS applications are developed in Objective-C.

Objective-C is a superset of the C programming language that extends it with object-oriented features. Typical applications make heavy use of objects, and most function calls are actually object method invocations. Moreover, these method invocations are all funneled through a single dispatch (send message) routine. This makes it difficult to obtain a meaningful program control flow graph (CFG) for a program. However, a CFG is the starting point required for most other interesting program analysis. Thus, we had to develop novel techniques to reconstruct meaningful CFGs for iOS applications. Based on the control flow graphs, we could then perform data flow analysis to identify flows where sensitive data might be leaked without asking for user permission.

Using PiOS, we analyzed 825 free applications available on the iTunes App Store. Moreover, we also examined 582 applications offered through the Cydia repository. The Cydia repository is similar to the App Store in that it offers a collection of iOS applications. However, it is not associated with Apple, and hence, can only be used by jailbroken devices. By checking applications both from the official Apple App Store and Cydia, we can examine whether the risk of privacy leaks increases if unvetted applications are installed.

The contributions of this chapter are as follows:

- We present a novel approach that is able to automatically create comprehensive CFGs from binaries compiled from Objective-C code. We can then perform reachability analysis on these CFGs to identify possible leaks of sensitive information from a mobile device to third parties.

- We describe the prototype implementation of our approach, PiOS, that is able to analyze large bodies of iPhone applications, and automatically determines if these applications leak out any private information.

- To show the feasibility of our approach, we have analyzed more than 1,400 iPhone applications. Our results demonstrate that a majority of applications leak the device ID. However, with a few notable exceptions, applications do respect personal identifiable information. This is even true for applications that are not vetted by Apple.

5.1 System Overview

The goal of PiOS is to detect privacy leaks in applications written for iOS. This makes is necessary to first concretize our notion of a privacy leak. We define as a privacy leak any event in which an iOS application reads sensitive data from the device and sends this data to a third party without the user's consent. To request the user's consent, the application displays a message (via the device's UI) that specifies the data item that should be accessed. Moreover, the user is given the choice of either granting or denying the access. When an application does not ask for user permission, it is in direct violation of the *iPhone developer program license agreement* [3], which mandates that no sensitive data may be transmitted unless the user has expressed her explicit consent.

The license agreement also states that an application may ask for access permissions only when the proper functionality of the application depends on the availability of the data. Unfortunately, this requirement makes it necessary to understand the semantics of the application and its intended use. Thus, in this work, we do not consider privacy violations where the user is explicitly asked to grant access to data, but this data is not essential to the program's functionality.

In a next step, we have to decide the types of information that constitute sensitive user data. Turning to the Apple license agreement is of little help. Unfortunately, the text does neither precisely define user data nor enumerate functions that should be considered sensitive. Since the focus of this work is to detect leaks in general, we take a loose approach and consider a wide variety of data that can

be accessed through the iOS API as being potentially sensitive. In particular, we used the open-source iOS application Spyphone [84] as inspiration. The purpose of Spyphone is to demonstrate that a significant number of interesting data elements (user and device information) is accessible to programs. Since this is exactly the type of information that we are interested in tracking, we consider these data elements as sensitive. A more detailed overview of sensitive data elements is presented in Section 5.4.

Data flow analysis. The problem of finding privacy leaks in applications can be framed as a data flow problem. That is, we can find privacy leaks by identifying data flows from input functions that access sensitive data (called *sources*) to functions that transmit this data to third parties (called *sinks*). We also need to check that the user is not asked for permission. Of course, it would be relatively easy to find the location of functions that interact with the user, for example, by displaying a message box. However, it is more challenging to automatically determine whether this interaction actually has the intent of warning the user about the access to sensitive data. In our approach, we use the following heuristic: Whenever there is any user interaction between the point where sensitive information is accessed and the point where this information could be transferred to a third party, we optimistically assume that the purpose of this interaction is to properly warn the user.

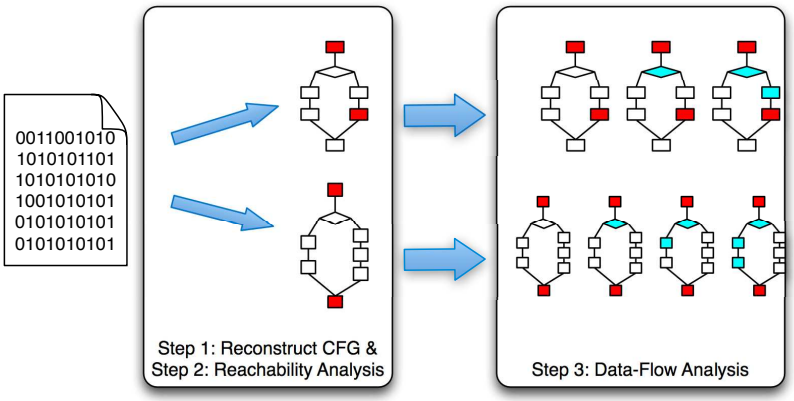

Figure 5.1: The PiOS system.

As shown in Figure 5.1, PiOS performs three steps when checking an iOS application for privacy leaks. First, PiOS reconstructs the control flow graph (CFG) of the application. The CFG is the underlying data structure (graph) that is used to find code paths from sensitive sources to sinks. Normally, a CFG is relatively straightforward to extract, even when only the binary code is available. Unfortunately, the situation is different for iOS applications. This is because almost all iOS programs are developed in Objective-C.

Objective-C programs typically make heavy use of objects. As a result, most function calls are actually invocations of instance methods. To make matters worse, these method invocations are all performed through an indirect call of a single dispatch function. Hence, we require novel binary analysis techniques to resolve method invocations, and to determine which piece of code is eventually invoked by the dispatch routine. For this analysis, we first attempt to reconstruct the class hierarchy and inheritance relationships between Objective-C classes. Then, we use backward slicing to identify both the arguments and types of the input parameters to the dispatch routine. This allows us to resolve the actual target of function calls with good accuracy. Based on this information, the control flow graph can be built.

In the second step, PiOS checks the CFG for the presence of paths that connect nodes accessing sensitive information (sources) to nodes interacting with the network (sinks). For this, the system performs a standard reachability analysis.

In the third and final step, PiOS performs data flow analysis along the paths to verify whether sensitive information is indeed flowing from the source to the sink. This requires some special handling for library functions that are not present in the binary, especially those with a variable number of arguments. After the data flow analysis has finished, PiOS reports the source/sink pairs for which it could confirm a data flow. These cases constitute privacy leaks. Moreover, the system also outputs the remaining paths for which no data flow was found. This information is useful to be able to focus manual analysis on a few code paths for which the static analysis might have missed an actual data flow.

5.2 Background Information

The goal of this section is to provide the reader with the relevant background information about iOS applications, their Mach-O binary format, and the problems that compiled Objective-C code causes for static binary analysis. The details of the PiOS system are then presented in later sections.

5.2.1 Objective-C

Objective-C is a strict superset of the C programming language that adds object-oriented features to the basic language. Originally developed at NextStep, Apple and its line of operating systems is now the driving force behind the development of the Objective-C language.

The foundation for the object-oriented aspects in the language is the notion of a class. Objective-C supports single inheritance, where every class has a single superclass. The class hierarchy is rooted at the *NSObject* class. This is the most basic class. Similar to other object-oriented languages, (static) class variables are shared between all instances of the same class. Instance variables, on the other hand, are specific to a single instance. The same holds for class and instance methods.

Protocols and categories. In addition to the features commonly found in object-oriented languages, Objective-C also defines *protocols* and *categories*. Protocols resemble interfaces, and they define sets of optional or mandatory methods. A class is said to adopt a protocol if it implements at least all mandatory methods of the protocol. Protocols themselves do not provide implementations.

Categories resemble aspects, and they are used to extend the capabilities of existing classes by providing the implementations of additional methods. That is, a category allows a developer to extend an existing class with additional functionality, even without access to the source code of the original class.

Message passing. The major difference between Objective-C binaries and binaries compiled from other programming languages (such as C or C++) is that, in Objective-C, objects do not call methods of other objects directly or through

virtual method tables (*vtables*). Instead, the interaction between objects is accomplished by sending messages. The delivery of these messages is implemented through a dynamic dispatch function in the Objective-C runtime.

To send a message to a receiver object, a pointer to the *receiver*, the name of the method (the so-called *selector*; a null-terminated string), and the necessary parameters are passed to the `objc_msgSend` runtime function. This function is responsible for dynamically resolving and invoking the method that corresponds to the given selector. To this end, the `objc_msgSend` function traverses the class hierarchy, starting at the receiver object, trying to locate the method that corresponds to the selector. This method can be implemented in either the class itself, or in one of its superclasses. Alternatively, the method can also be part of a category that was previously applied to either the class, or one of its superclasses. If no appropriate method can be found, the runtime returns an "object does not respond to selector" error.

Clearly, finding the proper method to invoke is a non-trivial, dynamic process. This makes it challenging to resolve method calls statically. The process is further complicated by the fact that calls are handled by a dispatch function.

5.2.2 Mach-O Binary File Format

iOS executables use the Mach-O binary file format, similar to MacOS X. Since many applications for these platforms are developed in Objective-C, the Mach-O format supports specific sections, organized in so-called *commands*, to store additional meta-data about Objective-C programs. The `__objc_classlist` section, for example, contains a list of all classes for which there is an implementation in the binary. These are either classes that the developer has implemented or classes that the static linker has included. The `__objc_classref` section, on the other hand, contains references to all classes that are used by the application. The implementations of these classes need not be contained in the binary itself, but may be provided by the runtime framework (the equivalent of dynamically-linked libraries). It is the responsibility of the dynamic linker to resolve the references in this section when loading the corresponding library. Further sections include

information about categories, selectors, or protocols used or referenced by the application.

Apple has been developing the Objective-C runtime as an open-source project. Thus, the specific memory layout of the involved data structures can be found in the header files of the Objective-C runtime. By traversing these structures in the binary (according to the header files), one can reconstruct basic information about the implemented classes. In Section 5.3.1, we show how we can leverage this information to build a class hierarchy of the analyzed application.

Signatures and encryption. In addition to dedicated sections containing metadata for Objective-C files, the Mach-O binary file format also supports cryptographic signatures and encrypted binaries. The `LC_SIGNATURE_INFO` command[2] stores the cryptographic signature information. Upon invoking a signed application, the operating system's loader verifies that the binary has not been modified. This is done by recalculating the signature and matching it against the information stored in the section. If the signatures do not match, the application is terminated.

The `LC_ENCYPTION_INFO` command consists of three fields that indicate whether a binary is encrypted and store the offset and the size of the encrypted content. When the field `cryptid` is set, this means that the program is encrypted. In this case, the two remaining fields (`cryptoffset` and `cryptsize`) identify the encrypted region within the binary. When a program is encrypted, the loader tries to retrieve the decryption key from the system's secure key chain. If a key is found, the binary is loaded to memory, and the encrypted region is replaced in memory with an unencrypted version thereof. If no key is found, the application cannot be executed.

5.2.3 iOS Applications

The mandatory way to install applications on iOS is through Apple's App Store. This store is typically accessed via iTunes. Using iTunes, the requested application bundle is downloaded and stored in a zip archive (with an `.ipa` file exten-

[2] A command is part of a section.

sion). This bundle contains the application itself (the binary), data files, such as images, audio tracks, or databases, and meta-data related to the purchase.

All binaries that are available via the App Store are encrypted and digitally signed by Apple. When an application is synchronized onto the mobile device (iPhone, iPad, or iPod), iTunes extracts the application folder from the archive (bundle) and stores it on the device. Furthermore, the decryption key for the application is added to the device's secure key chain. This is required because the application binaries are also stored in encrypted form.

As PiOS requires access to the unencrypted binary code for its analysis, we need to find a way to obtain the decrypted version of a program. Unfortunately, it is not straightforward to extract the application's decryption key from the device (and the operating system's secure key chain). Furthermore, to use these keys, one would also have to implement the proper decryption routines. Thus, we use an alternative method to obtain the decrypted binary code.

Decrypting iOS applications. Apple designed the iPhone platform with the intent to control all software that is executed on the devices. Thus, the design does not intend to give full system (or root) access to a user. Moreover, only signed binaries can be executed. In particular, the loader will not execute a signed binary without a valid signature from Apple. This ensures that only unmodified, Apple-approved applications are executed on the device.

The first step to obtain a decrypted version of an application binary is to lift the restriction that only Apple-approved software can be executed. To this end, one needs to jailbreak the device[3]. The term *jailbreaking* refers to a technique where a flaw in the iOS operating system is exploited to unlock the device, thereby obtaining system-level (root) access. With such elevated privileges, it is possible to modify the system loader so that it accepts any signed binary, even if the signature is not from Apple. That is, the loader will accept any binary as being valid even if it is equipped with a self-signed certificate. Note that jailbroken devices still have access to the iTunes App Store and can download and run Apple-approved applications.

[3]In July 2010 the Library of Congress which runs the US Copyright Office found that jailbreaking an iPhone is fair use [19].

One of the benefits of jailbreaking is that the user obtains immediate access to many development tools ready to be installed on iOS, such as a debugger, a disassembler, and even an SSH server. This makes the second step quite straightforward: The application is launched in the debugger, and a breakpoint is set to the program entry point. Once this breakpoint triggers, we know that the system loader has verified the signature and performed the decryption. Thus, one can dump the memory region that contains the now decrypted code from the address space of the binary.

5.3 Extracting CFGs from Objective-C Binaries

Using the decrypted version of an application binary as input, PiOS first needs to extract the program's inter-procedural control flow graph (CFG). Nodes in the CFG are basic blocks. Two nodes connected through an edge indicate a possible flow of control. Basic blocks are continuous instructions with linear control flow. Thus, a basic block is terminated by either a conditional branch, a jump, a call, or the end of a function body.

Disassembly and initial CFG. In an initial step, we need to disassemble the binary. For this, we chose IDA Pro, arguably the most popular disassembler. IDA Pro already has built-in support for the Mach-O binary format, and we implemented our analysis components as plug-ins for the IDA-python interface. Note that while IDA Pro supports the Mach-O binary format, it provides only limited additional support to analyze Objective-C binaries: For example, method names are prepended with the name of the class that implements the method. Similarly, if load or store instructions operate on instance variables, the memory references are annotated accordingly. Unfortunately, IDA Pro does not resolve the actual targets of calls to the `objc_msgSend` dispatch function. It only recognizes the call to the dynamic dispatch function itself. Hence, the resulting CFG is of limited value. The reason is that, to be able to perform a meaningful analysis, it is mandatory to understand which method in which class is invoked whenever a message is sent. That is, PiOS needs to resolve, for every call to the `objc_msgSend` function,

what method in what class would be invoked by the dynamic dispatch function during program execution.

Section 5.3.2 describes how PiOS is able to resolve the targets of calls to the dispatch function. As this process relies on the class hierarchy of a given application, we first discuss how this class hierarchy can be retrieved from an application's binary.

5.3.1 Building a Class Hierarchy

To reconstruct the class hierarchy of a program, PiOS parses the sections in the Mach-O file that store basic information about the structure of the classes implemented by the binary. The code of Apple's Objective-C runtime is open source, and thus, the exact layout of the involved structures can be retrieved from the corresponding header files. This makes the parsing of the binaries easy.

To start the analysis, the `__objc_classlist` section contains a list of all classes whose implementation is present in the analyzed binary (that is, all classes implemented by the developer or included by the static linker). For each of these classes, we can extract its type and the type of its superclass. Moreover, the entry for each class contains structures that provide additional information, such as the list of implemented methods and the list of class and instance variables. Similarly, the Mach-O binary format mandates sections that describe protocols used in the application, and categories with their implementation details.

In principle, the pointers to the superclasses would be sufficient to recreate the class hierarchy. However, it is important for subsequent analysis steps to also have information about the available methods for each class, as well as the instance and class variables. This information is necessary to answer questions such as "does a class `C`, or any of its superclasses, implement a given method `M`?"

Obviously, not all classes and types used by an application need to be implemented in the binary itself. That is, additional code could be dynamically linked into an application's address space at runtime. Fortunately, as the iOS SDK contains the header files describing the APIs (e.g., classes, methods, protocols, ...) accessible to iOS applications, PiOS can parse these header files and extend the class hierarchy with the additional required information.

5.3.2 Resolving Method Calls

As mentioned previously, method calls in Objective-C are performed through the dispatch function `objc_msgSend`. This function takes a variable number of arguments (it has a *vararg* prototype). However, the first argument always points to the object that receives the message (that is, the called object), while the second argument holds the selector, a pointer to the name of the method. On the ARM architecture, currently the only architecture supported by iOS, the first two method parameters are passed in the registers R0 and R1, respectively. Additional parameters to the dispatch function, which represent the actual parameters to the method that is invoked, are passed via registers R2, R3, and the stack.

Listing 5.1 shows a snippet of Objective-C code that initializes a variable of type `NSMutableString` to the string "Hello." This snippet leads to two method invocations (messages). First, a string object is allocated by the `alloc` method of the `NSMutableString` class. Second, this string object is initialized with the static string "Hello." This is done through the `initWithString` method.

The disassembly in Listing 5.2 shows that CPU register R0 is initialized with a pointer to the `NSMutableString` class. This is done by first loading the (fixed) address `off_31A0` (instruction: 0x266A) and then dereferencing it (0x266E). Similarly, a pointer to the selector (`alloc`, referenced by address `off_3154`) is loaded into register R1. The addresses of the `NSMutableString` class and the selector refer to elements in the `__objc_classrefs` and `__objc_selrefs` sections, respectively. That is, the dynamic linker will patch in the final addresses at runtime. However, since these addresses are fixed (constant) values, they can be directly resolved during static analysis and associated with the proper classes and methods. Once R0 and R1 are set up, the BLX (branch with link exchange) instruction calls the `objc_msgSend` function in the Objective-C runtime. The result of the `alloc` method (which is the address of the newly-created string instance) is saved in register R0.

In the next step, the `initWithString` method is called. This time, the method is not calling a static class function, but an instance method instead. Thus, the address of the receiver of the message is not a static address. In contrast, it is the address that the previous `alloc` function has returned, and that is already

conveniently stored in the correct register (R0). The only thing that is left to do is to load R1 with the proper selector (`initWithString`) and R2 with a pointer to the static string "Hello" (`cfstr_Hello`). Again, the BLX instruction calls the `objc_msgSend` function.

As the example shows, to analyze an Objective-C application, it is necessary to resolve the contents of the involved registers and memory locations when the dispatch function is invoked. To this end, PiOS employs backward slicing to calculate the contents of these registers at every call site to the `objc_msgSend` function in an application binary. If PiOS is able to determine the type of the receiver (R0) and the value of the selector (R1), it annotates the call site with the specific class and method that will be invoked when the program is executed.

```
NSMutableString *v;
v = [[NSMutableString alloc] initWithString : @''Hello'']
```

Listing 5.1: Simple Objective-C expression

```
__text:00002668 30 49       LDR     R1, =off_3154
__text:0000266A 31 48       LDR     R0, =off_31A0
__text:0000266C 0C 68       LDR     R4, [R1]
__text:0000266E 00 68       LDR     R0, [R0]
__text:00002670 21 46       MOV     R1, R4
__text:00002672 00 F0 32 E9 BLX     _objc_msgSend ;
                ; NSMutableString alloc
__text:00002676 2F 49       LDR     R1, =off_3190
__text:00002678 2F 4A       LDR     R2, =cfstr_Hello
__text:0000267A 09 68       LDR     R1, [R1]
__text:0000267C 00 F0 2C E9 BLX     _objc_msgSend ;
                ; NSMutableString  initWithString:
```

Listing 5.2: Disassembly of Listing 5.1

Backward Slicing

To determine the contents of the registers R0 and R1 at each call site to the `objc_msgSend` function, PiOS performs backward slicing [100], starting from

those registers. That is, PiOS traverses the binary backwards, recording all instructions that influence or define the values in the target registers. Operands that are referenced in such instructions are resolved recursively. The slicing algorithm terminates if it reaches the start of the function or if all values can be determined statically (i.e., they are statically defined). A value is statically defined if it is a constant operand of an instruction or a static memory location (address).

In Listing 5.2, for example, the slice for the call to `objc_msgSend` at address 0x2672 (the `alloc` call) stops at 0x2668. At this point, the values for both R0 and R1 are statically defined (as the two offsets `off_3154` and `off_31A0`). The slice for the call site at 0x267c (the string initialization) contains the instructions up to 0x2672. The slicing algorithm terminates there because function calls and message send operations store their return values in R0. Thus, R0 is defined to be the result of the message send operation at 0x2668.

Once the slice of instructions influencing the values of R0 and R1 is determined, PiOS performs forward constant propagation. That is, constant values are propagated along the slice according to the semantics of the instructions. For example, `MOV` operations copy a value from one register to another,[4] and `LDR` and `STR` instructions access memory locations.

Tracking Type Information

PiOS does not track (the addresses of) individual instances of classes allocated during runtime. Thus, the question in the previous example is how to handle the return value of the `alloc` function, which returns a dynamic (and hence, unknown pointer) to a string object. Our key insight is that, for our purposes, the actual address of the string object is not important. Instead, it is only important to know that R0 points to an object of type `NSMutableString`. Thus, we do not only propagate constants along a slice, but also type information.

In our example, PiOS can determine the return type of the `alloc` method call at address 0x2672 (the `alloc` method always returns the same type as its receiver; `NSMutableString` in this case). This type information is then prop-

[4] GCC seems to frequently implement such register transfers as `SUB Rd, Rs, #0`, or `ADD Rd, Rs, #0`.

agated along the slice. As a result, at address 0x267c, we have at our disposal the crucial information that R0 contains an object of type `NSMutableString`.

To determine the types of function arguments and return values, our system uses two sources of information. First, for all external methods, the header files specify the precise argument and return types. Unfortunately, there is no such information for the methods implemented in the application binary. More precisely, although the data structure that describes class and instance methods does contain a field that lists the parameter types, the stored information is limited to basic types such as integer, Boolean, or character. All object arguments are defined as a single type *id* and, hence, cannot be distinguished easily.

Therefore, as a second source for type information, PiOS attempts to resolve the precise types of all arguments marked as *id*. To this end, the system examines, for each method, all call sites that invoke this method. For the identified call sites, the system tries to resolve the parameter types by performing the above-mentioned backward slicing and constant propagation steps. Once a parameter type is identified, the meta-data for the method can be updated accordingly. That is, we are building up a database as we learn additional type information for method call arguments.

Frequently, messages are sent to objects that are returned as results of previous method calls. As with method input arguments, precise return type information is only available for functions whose prototypes are defined in header files. However, on the ARM architecture, the return value of a method is always returned in register R0. Thus, for methods that have an implementation in the binary and whose return type is not a basic type, PiOS can derive the return type by determining the type of the value stored in R0 at the end of the called method's body. For this, we again use backward slicing and forward constant propagation. Starting with the last instruction of the method whose return type should be determined, PiOS calculates the slice that defines the type of register R0 at this program location.

5.3.3 Generating the Control Flow Graph

Once PiOS has determined the type of R0 and the content of R1 at a given call site to `objc_msgSend`, the system checks whether these values are "reasonable." To this end, PiOS verifies that the class hierarchy contains a class that matches the type of R0, and that this class, or any of its superclasses or categories, really implements the method whose name is stored as the selector in R1. Of course, statically determining the necessary values is not always possible. However, note that in cases where only the selector can be determined, PiOS can still reason about the type of the value in R0 if there is exactly one class in the application that implements the selector in question.

When PiOS can resolve the target of a function call through the dispatch method, this information is leveraged to build the control flow graph of the application. More precisely, when the target of a method call (the recipient of the message) is known, and the implementation of this method is present in the binary under analysis (and not in a dynamic library), PiOS adds an edge from the call site to the target method.

5.4 Finding Potential Privacy Leaks

The output of the process described in the previous section is an inter-procedural control flow graph of the application under analysis. Based on this graph, we perform reachability analysis to detect privacy leaks. More precisely, we check the graph for the presence of paths from sources (functions that access sensitive data) to sinks (functions that transmit data over the network). In the current implementation of PiOS, we limited the maximum path length to 100 basic blocks.

Interestingly, the way in which iOS implements and handles user interactions implicitly disrupts control flow in the CFG. More precisely, user interface events are reported to the application by sending messages to delegate objects that contain the code to react to these events. These messages are not generated from code the developer wrote, and thus, there is no corresponding edge in our CFG. As a result, when there is a user interaction between the point where a source is accessed, and data is transmitted via a sink, there will never be a path in our CFG. Thus, all

paths from sensitive sources to sinks represent potential privacy leaks. Of course, a path from a source to a sink does not necessarily mean that there is an actual data flow. Hence, we perform additional data flow analysis along an interesting path and attempt to confirm that sensitive information is actually leaked.

5.4.1 Sources and Sinks

In this section, we discuss in more detail how we identify sources of sensitive data and sinks that could leak this data.

Sources. Sources of sensitive information cover many aspects of the iOS environment. Table 5.1 enumerates the resources that we consider sensitive. As mentioned previously, this list is based on [84], where Seriot presents a comprehensive list of potentially sensitive information that can be accessed by iOS applications.

Access to the address book
Current GPS coordinates of the device
Unique Device ID
Photo Gallery
Email account information
WiFi connection information
Phone related information (Phone# , last called, etc.)
Youtube application (watched videos and recent search)
MobileSafari settings and history
Keyboard cache

Table 5.1: Sensitive information sources.

Any iOS application has full read and write access to the address book stored on the device. Access is provided through the `ABAddressBook` API. Thus, whenever an application performs the initial `ABAddressBookCreate` call, we mark this call instruction a source.

An application can only access current GPS coordinates if the user has explicitly granted the application permission to do so. This is enforced by the API, which displays a dialog to the user the first time an application attempts to access the `CoreLocation` functionality. If access is granted, the application

can install a delegate with the `CoreLocation` framework that is notified whenever the location is updated by the system. More precisely, every time the location is updated, the `CoreLocation` framework will invoke the handler method (i.e., `locationManager:didUpdateToLocation:fromLocation`) of the delegate which was passed to the `CLLocationManager:setDelegate` method during initialization.

A unique identifier for the iOS device executing the application is available to all applications through the `UIDevice uniqueIdentifier` method. This ID is represented as a string of 40 hexadecimal characters that uniquely identifies the device.

The keyboard cache is a local file accessible to all applications. This file contains all words that have been typed on the device. The only exception are characters typed into text fields marked to contain passwords.

Furthermore, there exist various property files that provide access to different pieces of sensitive information. The `commcenter` property file contains SIM card serial numbers and IMSI identifiers. The user's phone number can be accessed by querying the `standardUserDefaults` properties. Email account settings are accessible through the `acountsettings` properties file. Similar files exist that contain the history of the Youtube and MobileSafari applications, as well as recent search terms used in these applications. The `wifi` properties file contains the name of wireless networks the device was connected to. Also, a time stamp is stored, and the last time when each connection was active is logged. Accesses related to these properties are all considered sensitive sources by PiOS.

Sinks. We consider sinks as operations that can transmit information over the network, in particular, methods of the `NSURLConnection` class. However, there are also methods in other classes that might result in network requests, and hence, could be used to leak data. The `initWithContentsOfURL` method of the `NSString` class, for example, accepts a URL as parameter, fetches the content at that URL, and initializes the string object with this data. To find functions that could leak information, we carefully went through the API documentation. In total, we included 14 sinks.

5.4.2 Dataflow Analysis

Reachability analysis can only determine that there exists a path in the CFG that connects a source of sensitive information to a sink that performs networking operations. However, these two operations might be unrelated. Thus, to enhance the precision of PiOS, we perform an additional data flow analysis on the paths that the reachability analysis reports. That is, for every path that connects a source and a sink in the CFG, we track the propagation of the information accessed at the source node. If this data reaches one or more method parameters at the sink node, we can confirm a leak of sensitive information, and an alert is raised.

We use a standard data flow analysis that uses forward propagation along the instructions in each path that we have identified. For methods whose implementation (body) is not available in the binary (e.g., external methods such as `initWithString` of the `NSMutableString` class), we conservatively assume that the return value of this function is tainted when one or more one of the arguments is tainted.

Methods with variable number of arguments. To determine whether the output of an external function should be tainted, we need to inspect all input arguments. This makes functions with a variable number of arguments a little more intricate to handle. The two major types of such functions are string manipulation functions (e.g., `NSMutableString appendStringWithFormat`) that use a format string, and initialization functions for aggregate types that fetch the objects to be placed in the aggregate from the stack (e.g., `NSDictionary initWithObjects:andKeys`). Ignoring these functions is not a good option – especially because string manipulation routines are frequently used for processing sensitive data.

For string methods that use format strings, PiOS attempts to determine the concrete value (content) of the format string. If the value can be resolved statically, the number of arguments for this call is determined by counting the number of formatting characters. Hence, PiOS can, during the data flow analysis, taint the output of such a function if any of its arguments is tainted.

The initialization functions fetch the contents for the aggregate from the stack until the value NULL is encountered. Thus, PiOS iteratively tries to statically resolve the values on the stack. If a value statically resolves to NULL, the number of arguments for this call can be determined. However, since it is not guaranteed that the NULL value can be determined statically, we set the upper bound for the number of parameters to 20.

5.5 Evaluation

We evaluated PiOS on a body of 1,407 applications. 825 are free applications that we obtained from Apple's iTunes store. We downloaded the remaining 582 applications from the popular BigBoss [1] repository which is installed by default with Cydia [37] during jailbreaking. Applications originating from the Cydia repositories are not encrypted. Therefore, these applications can be directly analyzed by PiOS. Applications purchased from the iTunes store, however, need to be decrypted before any binary analysis can be started. Thus, we automated the decryption approach described in Section 5.2.3.

Since iTunes does not support direct searches for free applications, we rely on apptrakr.com [2] to provide a continuously updated list of popular, free iOS applications. Once a new application is added to their listings, our system automatically downloads the application via iTunes and decrypts it. Subsequently, the application is analyzed with PiOS.

5.5.1 Resolving Calls to `objc_msgSend`

As part of the static analysis process, PiOS attempts to resolve all calls to the `objc_msgSend` dispatch function. More precisely, for each call site referring to `objc_msgSend`, the system reasons about the target method (and class) that would be invoked during runtime (described in Section 5.3.2) by the dispatch routine. This is necessary to build the program's control flow graph.

During the course of evaluating PiOS on 1,407 applications, we identified 4,156,612 calls to the message dispatch function. PiOS was able to identify the corresponding class and method for 3,408,421 call sites (82%). Note that PiOS

reports success only if the inferred class exists in the class hierarchy, and the selector denotes a method that is implemented by the class, or its ancestors in the hierarchy. These results indicate that a significant portion of the CFGs can be successfully reconstructed, despite the binary analysis challenges.

5.5.2 Advertisement and Tracking Libraries

PiOS resolves all calls to the `objc_msgSend` function regardless of whether the target method in the binary was written by the application developer herself, or whether it is part of a third-party library that was statically linked against the application. In an early stage of our experiments, we realized that many applications contained one (or even multiple instances) of a few popular libraries. Moreover, all these libraries triggered PiOS' privacy leak detection because the system detected paths over which the unique device ID was transmitted to third parties.

A closer examination revealed that most of these libraries are used to display advertisement to users. As many iOS applications include advertisements to create a stream of revenue for the developer, their popularity was not surprising. However, the fact that all these libraries also leak the device IDs of users that install their applications was less expected. Moreover, we also found tracking libraries, whose sole purpose is to collect and compile statistics on application users and usage. Clearly, these libraries send the device ID as a part of their functionality.

Applications that leak device IDs are indeed pervasive, and we found that 656 (or 55% of all applications) in our evaluation data set include either advertisement or tracking libraries. Some applications even include multiple different libraries at once. In fact, these libraries were so frequent that we decided to white-list them; in the sense that it was of no use for PiOS to constantly re-analyze and reconfirm their data flows. More precisely, whenever a path starts from a sensitive sink in a white-listed library, further analysis is skipped for this path. Thus, the analysis results that we report in the subsequent sections only cover the code that was actually written by application developers. For completeness, Table 5.2 shows how frequently our white-list triggered for different applications.

While not directly written by an application developer, libraries that leak device IDs still pose a privacy risk to users. This is because the company that is

Library Name	Type	# apps using	# white-listed accesses
AdMob	Advertising	538	55,477
Pinchmedia	Statistics/Tracking	79	2,038
Flurry	Statistics/Tracking	51	386
Mobclix	Advertising	49	1,445
AdWhirl	Advertising	14	319
QWAdView	Advertising	14	219
OMApp	Statistics/Tracking	10	658
ArRoller	Advertising	8	734
AdRollo	Advertising	7	127
MMadView	Advertising	2	96
Total		772	61,499

Table 5.2: Prevalence of advertising and tracking libraries.

running the advertisement or statistics service has the possibility to aggregate detailed application usage profiles. In particular, for a popular library, the advertiser could learn precisely which subset of applications (that include this library) are installed on which devices. For example, in our evaluation data set, AdMob is the most-widely-used library to serve advertisements. That is, 82% of the applications that rely on third-party advertising libraries include AdMob. Since each request to the third-party server includes the unique device ID and the application ID, AdMob can easily aggregate which applications are used on any given device.

Obviously, the device ID cannot immediately be linked to a particular user. However, there is always the risk that such a connection can be made by leveraging additional information. For example, AdMob was recently acquired by Google. Hence, if a user happens to have an active Google account and uses her device to access Google's services (e.g., by using GMail), it now becomes possible for Google to tie this user account to a mobile phone device. As a result, the information collected through the ad service can be used to obtain a detailed overview of who is using which applications. Similar considerations apply to many other services (such as social networks like Facebook) that have the potential to link a device ID to a user profile (assuming the user has installed the social networking application).

The aforementioned privacy risk could be mitigated by Apple if an identifier would be used that is unique for the *combination* of application and device. That is, the device ID returned to a program should be different for each application.

5.5.3 Reachability Analysis

Excluding white-listed accesses to sensitive data, PiOS checked the CFGs of the analyzed applications for the presence of paths that connect sensitive sources to sinks. This analysis resulted in a set of 205 applications that contain at least one path from a source to a sink, and hence, a potential privacy leak. Interestingly, 96 of the 656 applications that triggered the white-list also contain paths in their core application code (i.e., outside of ad or tracking libraries).

The overwhelming majority (i.e., 3,877) of the accessed sources corresponds to the unique device identifier. These accesses originate from 195 distinct applications. 36 applications access the GPS location data at 104 different program locations. Furthermore, PiOS identified 18 paths in 5 applications that start with an access to the address book. One application accesses both the MobileSafari history and the photo storage. An overview that summarizes the potential leaks is shown Table 5.3.

Source	# App Store	# Cydia	Total
DeviceID	170	25	195
Location	35	1	36
Address book	4	1	5
Phone number	1	0	1
MobileSafari history	0	1	1
Photos	0	1	1

Table 5.3: Applications accessing sensitive data.

An interesting conclusion that one can draw from looking at Table 5.3 is that, overall, the programs on Cydia are not more aggressive (malicious) than the applications on the App Store. This is somewhat surprising, since Cydia does not implement any vetting process.

5.5.4 Data Flow Analysis

For the 205 applications that were identified with possible information leaks, PiOS then performed additional analysis to attempt to confirm whether sensitive information is actually leaked. More precisely, the system enumerates all paths in the CFG between a pair of source and sink nodes whose length does not exceed 100 basic blocks. Data flow analysis is then performed on these paths until either a flow indicates that sensitive information is indeed transmitted over the network, or all paths have been analyzed (without result). Note that our analysis is not sound; that is, we might miss data flows due to code constructs that we cannot resolve statically. However, the analysis is precise, and every confirmed flow is indeed a privacy leak. This is useful when the majority of paths actually correspond to leaks, which we found to be true.

For 172 applications, the data flow analysis confirmed a flow of sensitive information to a sink. We manually analyzed the remaining 33 applications to asses whether there really is no data flow, or whether we encountered a false negative. In six applications, even after extensive, manual reverse engineering, we could not find an actual flow. In these cases, our data flow analysis produced the correct result. The remaining 27 cases were missed due to a variety of program constructs that are hard to analyze statically (recall that we operate directly on binary code). We discuss a few of the common problems below.

For six applications, the data flow analysis was unsuccessful because these applications make use of custom-written functions to store data in aggregate types. Also, PiOS does not support nested data structures such as dictionaries stored inside dictionaries.

In four cases, the initial step could not resolve all the necessary object types. For example, PiOS was only able to resolve that the invoked method (the sent message) was `setValue:forHTTPHeader-Field`. However, the object on which the method was called could not be determined. As a result, the analysis could not proceed.

Two applications made use of a JSON library that adds categories to many data types. For example, the `NSDictionary` class is extended with a method that returns the contents of this dictionary as a JSON string. To this end, the

method sends each object within the dictionary a `JSONRepresentation` message. The flows of sensitive information were missed because PiOS does not keep track of the object types stored within aggregate data types (e.g., dictionaries).

In other cases, flows were missed due to aliased pointers (two different pointers that refer to the same object), leaks that only occur in the applications exception handler (which PiOS does not support), or a format string that was read from a configuration file.

5.5.5 Case Studies

When examining the results of our analysis (in Table 5.3), we can see that most leaks are due to applications that transmit the device ID. This is similar to the situation of the advertising and tracking libraries discussed previously. Moreover, a number of applications transmit the user's location to a third party. These cases, however, cannot be considered real privacy leaks. The reason is that iOS itself warns users (and asks for permission) whenever an application makes use of the `CoreLocation` functionality. Unfortunately, such warnings are not provided when other sensitive data is accessed. In the following, we discuss in more detail the few cases in which the address book, the browser history, and the photo gallery is leaked.

Address book leaks. PiOS indicated a flow of sensitive information for the *Gowalla* social networking application. Closer examination of the offending path showed that the application first accesses the address book and then uses the `loadRequest` method of the `UIWebView` class to launch a web request. As part of this request, the application transmits all user names and their corresponding email addresses.

We then attempted to manually confirm the privacy leak by installing Gowalla on a iOS device and monitoring the network traffic. The names of the methods involved in the leak that was detected, `emailsAndNamesQueryString` and `emailsAndNamesFromAddressBook`, both implemented in the same class (i.e., `InviterViewController`), made it easy to find the corresponding actions on the user interface. In particular, the aforementioned class is responsible

for inviting a user's friends to also download and use the *Gowalla* application. A user can choose to send invitations to her Twitter followers, Facebook friends, or simply select a group of users from the address book. This is certainly legitimate behavior. However, the application also, and before the user makes any selection, transmits the address book in its entirety to the developer. This is the flow that PiOS detects. The resulting message[5] indicates that the developers are using this information to crosscheck with their user database whether any of the user's contacts already use the application. When we discovered this privacy breach, we informed Apple through the "Report a problem" link associated with this application on iTunes. Despite our detailed report, Apple's response indicated that we should discuss our privacy concerns directly with the developer.

PiOS found another leak of address book data in *twitterickl*. This application checks all contacts in the address book to determine whether there is a picture associated with the person. If not, the application attempts to obtain a picture of this person from Facebook. While information from the address book is used to create network requests, these requests are sent to Facebook. It is not the application developers that attempt to harvest address book data.

In other three cases, the address book is also sent without displaying a direct warning to the user before the sensitive data is transferred. However, these applications either clearly inform the user about their activity at the beginning (*Facebook*) or require the user to actively initiate the transfer by selecting contacts from the address book (*XibGameEngine*, to invite friend; *FastAddContacts* to populate the send-to field when opening a mail editor). This shows that not all leaks have the same impact on a user's privacy, although in all cases, PiOS correctly recognized a sensitive data flow.

Browser history and photo gallery. Mobile-Spy offers an application called *smartphone* on the Cydia market place. This application is advertised as a surveillance solution to monitor children or employees. Running only on jailbroken devices, the software has direct access to SMS messages, emails, GPS coordinates, browser history, and call information. The application is designed as a daemon

[5] "We couldn't find any friends from your Address Book who use Gowalla. Why don't you invite some below?"

process running in the background, where it collects all available information and transfers it information to Mobile-Spy's site. The user who installs this application can then go to the site and check the collected data.

PiOS was able to detect two flows of sensitive information in this application. The upload of the MobileSafari history, and the upload of the Photo gallery. However, PiOS was not able to identify the leaking of the address book, and the transfer of the email box, or SMS messages. The reason for all three cases is that the application calls `system` with a `cp` command to make a local copy of the local phone databases that hold this information. These copies are later opened, and their content is transferred to the Mobile-Spy service. Tracking through the invocation of the `system` library call would require PiOS to understand the semantics of the passed (shell) commands. Clearly, this is outside of the scope of this thesis.

Phone Number. In November 2009, Apple removed all applications developed by Storm8 due to privacy concerns. More precisely, these applications were found to access the user's phone number via the `SBFormattedPhoneNumber` key in the `standardUserDefaults` properties. Once retrieved, the phone number was then transmitted to Storm8's servers. Shortly after the ban of all their applications, Storm8 developers released revised versions that did not contain the offending behavior. This incident prompted Apple to change their vetting process, and now, all applications that access this key are rejected. Thus, to validate PiOS against this known malicious behavior, we obtained a version of Vampires Live (a Storm8 application) that predates this incident, and hence, contains the offending code. PiOS correctly and precisely identified that the phone number is read on program startup and then sent to Storm8.

5.5.6 Discussion

With the exception of a few bad apples, we found that a significant majority of applications respects the person user information stored on iOS devices. While this could be taken as a sign that Apple's vetting process is successful, we found similar results for the unchecked programs that are hosted on Cydia, an unofficial

repository that can only be accessed with a jailbroken phone. However, the unique device ID of the phone is treated distinctively different, and more than half of the applications leak this information (often because of advertisement and tracking libraries that are bundled with the application). While these IDs cannot be directly linked to a user's identify, they allow third parties to profile user behavior. Moreover, there is always the risk that outside information can be used to eventually make the connection between the device ID and a user.

5.6 Limitations

It is not always possible for PiOS to statically determine the receiver and selector for every call to the `objc_msgSend` function.Recall that the selector is the name of a method. Typically, this value is a string value stored in the `__objc_selref` section of the application. However, any string value can be converted to a selector, and it is possible to write programs that receive string values whose value cannot be statically determined (e.g., as a response to a networking request, or as a configuration value chosen by the user). This limitation is valid for all static analysis approaches and not specific to PiOS.

Furthermore, aggregate types in Objective-C, such as `NSArray`, `NSSet`, or `NSDictionary` are not generic. Therefore, the types of objects in such containers cannot be specified more precisely than `id` (which is of type `NSObject`). For example, the delegate method `touchesEnded:withEvent` provided by the `UIResponder` class is called whenever the user finishes a touch interaction with the graphical user interface (e.g., click an element, swipe an area, ...). This method receives as the first argument a pointer to an object of type `NSSet`. Although this set solely contains `UITouch` elements, the lack of generic support in Objective-C prohibits the type information to be stored with the aggregate instance. Similarly, any object can be added to an `NSArray`. Thus, PiOS has to treat any value that is retrieved from an aggregate as `NSObject`. Nevertheless, as described in Section 5.3.2, PiOS might still be able to reason about the type of such an object if a subsequent call to the `objc_msgSend` function uses a selector that is implemented by exactly one class.

5.7 Summary

The growing popularity and sophistication of smartphones, such as the iPhone or devices based on Android, have also increased concerns about the privacy of their users. To address these concerns, smartphone OS designers have been using different security models to protect the security and privacy of users. For example, Android applications are shipped with a manifest that shows all required permissions to the user at installation time. In contrast, Apple has decided to take the burden off its iPhone users and determine, on their behalf, if an application conforms to the predefined privacy rules. Unfortunately, Apple's vetting process is not public, and there have been cases in the past (e.g., [101]) where vetted applications have been discovered to be violating the privacy rules defined by Apple.

The goal of the work described in this chapter is to automatically analyze iOS applications and to study the threat they pose to user data. We present a novel approach that is able to automatically create comprehensive CFGs from binaries compiled from Objective-C code. We can then perform reachability analysis on the generated CFGs and identify private data leaks. We have analyzed more than 1,400 iPhone applications. Our experiments show that most applications do not secretly leak any sensitive information that can be attributed to a person. This is true both for vetted applications on the App Store and those provided by Cydia. However, a majority of applications leaks the device ID, which can provide detailed information about the habits of a user. Moreover, there is always the possibility that additional data is used to tie a device ID to a person, increasing the privacy risks.

Chapter 6

Related Work

In this chapter, we will discuss the work that is related to the topics that are covered in this dissertation. First, we will cover the related work in the research area of web-spam. Subsequently, we will discuss the works that are related to our drive-by download prevention technique. This includes works in the realm of shellcode detection and JavaScript analysis. Finally, we present the related work in the field of static analysis and mobile device security.

Web spam In recent years, considerable effort was dedicated to the detection and mitigation of web spam. In [44], the authors present different techniques to fool search engine ranking algorithms. Boosting techniques, such as link farms, are used to push pages to undeserved higher ranks in search engine results. Hiding or cloaking techniques are used to trick search engines by serving different content to the search engine spiders and human users.

One of the most prominent boosting techniques are link farms, and multiple researchers have presented techniques for detecting them. For example, Wu and Davison [104] propose an algorithm that generates a graph of a link farm from an initial seed and propagates badness values through this graph. This information can then be used with common, link-based ranking algorithms, such as PageRank or HITS. The same authors also present their findings on cloaking and redirection techniques [103]. Ntoulas et al. [68] present a technique of detecting spam pages by content analysis. This work only takes query independent features into account, while Svore et al. [93] also use query dependent information. A system

to detect cloaking pages is proposed by Chellapilla and Chickering in [13]. For this, a given URL is downloaded twice, providing different user agent strings for each download. If the pages are (significantly) different, the page uses cloaking techniques.

Wang et al. [99] follow the money in advertising schemes and propose a five-layer, double-funnel model to explain the relations that exist between advertisers and sites that employ web spam techniques. Fetterly et al. [35] present a series of measurements to evaluate the effectiveness in web spam detection. A quantitative study of forum spamming was presented by Niu et al. [67]

The work that is closest to our attempt in inferring the importance of different web page features is [7]. In that paper, Bifet et al. attempt to infer the importance of page features for the ranking algorithm by analyzing the results for different queries. They extract feature vectors for each page and try to model the ranking function by using support vector machines. Since their work is based on already existing pages, they do not have control over certain features (e.g., in-link properties). In [32], Evans performs a statistical analysis of the effect that certain factors have on the ranking of pages. While he includes factors, such as the listing of pages in web directories and a site's PageRank value, Evans only focuses on query independent values while neglecting all other factors.

Drive-by downloads While powerful, existing analysis techniques are typically too heavyweight to be used for detection on a client machine. In contrast to that, our proposed technique detects drive-by download attacks by monitoring potentially malicious scripts directly in the browser.

Previous studies have shown that drive-by download attacks pose a real threat to the Internet and its users. The mechanisms used by attackers to mount their attacks are investigated by Provos et al. in [76]. The life cycle of an infected machine is analyzed by Polychronakis in [74]. In [75], Provos et al. present a measurement study that reports that the results for 1.3% of all Google search queries contain at least one link pointing to a page that performs a drive-by attack. Also, Frei et al. [38] analyzed the vulnerability landscape of web browsers in the Internet. Apparently, only 60% of the users that navigate the Internet everyday use the latest, most secure version of their web browser. Based on a Secunia report [82],

the authors argue that many browser plug-ins commonly in use have known vulnerabilities. The fact that many users only reluctantly update their web browsers and plug-ins makes it feasible for attackers to distribute attacks that target old vulnerabilities. As many of the vulnerabilities leading to control flow hijacking are present in ActiveX components, Dormann and Plakosh [25] propose fuzzy testing as a means of detecting such flaws before distributing a component.

Network level shellcode detection Detecting shellcode in network traffic has a long standing history. Network intrusion detection systems, such as Snort [80] or Bro [69], rely on signatures to identify malicious network streams. While signature detection works well for known static threats, advanced polymorphic shellcode and engines that can automatically produce such shellcode can sometimes evade these detection techniques. Based on abstract payload execution, Toth and Kruegel have proposed a mechanism to detect buffer overflow attacks [96]. More precisely, their prototype implementation identifies long valid sequences of instructions in HTTP requests, thus detecting the NOP sledge that commonly accompanies shellcode. Continuing this work, Polychronakis et al. [71, 73] proposed to apply lightweight emulation on network data to identify polymorphic shellcode. This approach relies on the so-called GetPC heuristic. That is, a shellcode is only identified if a sequence of instructions is emulated that reads the current program counter value. The class of non-self-contained shellcode, however, contains code that reaches its goal without showing such behavior. In [72], the authors extend their detection techniques to also identify this class of attacks. While network-traffic-based techniques are useful, they typically cannot be used to detect drive-by downloads. The reason is that, although JavaScript contents of a web page are transmitted over the network, this code is often obfuscated. Furthermore, the shellcode contained in the JavaScript scripts are not transmitted in binary form. Instead, the ASCII representation of the individual bytes is transmitted. This sequence does not yield a valid instruction sequence in general.

Analyzing JavaScript Analyzing malicious JavaScript has experienced significant attention by the scientific community. Hallaraker and Vigna [45] describe an approach to audit the execution of JavaScript code. These audit logs can be

compared to high-level policies to detect potential attacks. Similarly, Feinstein and Peck introduced Caffeine Monkey [33], a tool that supports the collection and analysis of malicious JavaScript. To this end, they extended the Mozilla SpiderMonkey JavaScript engine by adding run-time logging facilities. Chenette et al. [16] aim at automatically reversing the obfuscation of malicious JavaScripts. Their approach relies on hooking techniques to monitor calls to relevant JavaScript functions, such as eval or document.write. These systems focus on auditing JavaScript activity, while our approach aims at detecting malicious drive-by downloads.

Vogt et al. propose a system that prevents cross-site scripting attacks performed by malicious JavaScript code [98]. To protect a user from JavaScript that tries to steal sensitive information, the propagation of such information through the JavaScript engine is tracked. Requests to a domain containing information originating from another domain raise an alert, and allow the user to stop further execution of the script.

Static analysis Clearly, static analysis and program slicing have been used before. Weiser [100] was the first to formalize a technique called program slicing. As outlined in Section 5.3.2, PiOS makes use of this technique to calculate program slices that define receiver and selector values at call-sites to the `objc_msgSend` dynamic dispatch function.

Also, static binary analysis was used in the past for various purposes. Kruegel et al. [57] made use of static analysis to perform mimicry attacks on advanced intrusion detection systems that monitor system call invocations. Christodorescu and Jha [17] present a static analyzer for executables that is geared towards detecting malicious patterns in binaries even if the content is obfuscated. Similarly, the work described in Christodorescu [18] et al. is also based on binary static analysis, and identifies malicious software using a semantics-aware malware detection algorithm. However, some of the obfuscation techniques available on the x86 architecture cannot be used on ARM based processors. The RISC architecture of ARM facilitates more robust disassembly of binaries, as instructions cannot be nested within other instructions. Furthermore, the strict memory alignment pro-

hibits to jump to the middle of ARM instructions. Thus, disassembling ARM binaries generally produces better results than disassembling x86 binaries.

Note that while static binary analysis is already challenging in any domain, in our work, the analysis is further complicated by the fact that most iOS applications are developed in Objective-C. It is not trivial to obtain a meaningful program control flow graph for iOS applications.

In [10], Calder and Grunwald optimize object code of C++ programs by replacing virtual function calls with direct calls if the program contains exactly one implementation that matches the signature of the virtual function. This is possible because the mangled name of a function stored in an object file, contains information on the class and parameter types. PiOS uses a similar technique to resolve the type of a receiver of a message. However, PiOS only follows this approach if the type of the receiver cannot be determined by backwards slicing and constant propagation.

In another work, Dean et al. [24] present an approach that performs class hierarchy analysis to statically resolve virtual function calls and replace them with direct function calls. In PiOS, we do not use the class hierarchy to resolve the invoked method. However, we do use this information to verify that the results of the backwards slicing and forward propagation step are consistent with the class hierarchy, and thus sensible.

PiOS is also related to existing approaches that perform static data flow analysis. Livshits and Lam [58], for example, use static taint analysis for Java byte-code to identify vulnerabilities that result from incomplete input validation (e.g., SQL injection, cross site scripting). The main focus of Tripp et al. [97] is to make static taint analysis scale to large real-world applications. To this end, the authors introduce hybrid thin-slicing and combine it with taint analysis to analyze large web applications, even if they are based on application frameworks, such as Struts or Spring. Furthermore, Pixy [52] performs inter-procedural, context-sensitive data-flow analysis on PHP web-applications, and also aims to identify such taint-style vulnerabilities.

Mobile device security There has also been some related work in the domain of mobile devices: Enck et al. [30] published TaintDroid, a system that shares a

similar goal with this work; namely, the analysis of privacy leaks in smart phone applications. Different to our system, their work targets Android applications and performs *dynamic* information-flow tracking to identify privacy leaks. Most Android applications are executed by the open source Dalvik virtual machine. The information-flow capabilities of TaintDroid were build into a modified version of this VM. iOS applications, in contrast, are compiled into native code and executed by the device's CPU directly. TaintDroid was evaluated on 30 popular Android applications. The results agree quite well with our findings. In particular, many of the advertising and statistics libraries that we identified in Section 5.5.2 also have corresponding Android versions. As a result, TaintDroid raised alerts when applications transmitted location data to AdMob, Mobclix, and Flurry back-end servers. Furthermore, Enck et al. [31] present an approach named Kirin where they automatically extract the security manifest of Android applications. Before an application is installed, this manifest is evaluated against so-called *logic invariants*. The result is that the user is only prompted for her consent to install the application if these invariants are violated. That is, only applications that violate a user's assumption of privacy and security are prompted for the user agreement during installation. The concept of a security manifest provides the user basic information on which she can base her decision on whether to install an application or not. Unfortunately, the iOS platform does not provide such amenities. To take a decision, the user can only rely on the verbal description of the application and Apple's application vetting process.

Another work that focuses on Android is the formal language presented by Chaudhuri [12]. Together with operational semantics and a type system, the author created the language with the aim of being able to describe Android applications with regard to security properties. However, the language currently only supports Android-specific constructs. That is, the general Java constructs that build the majority of an application's code cannot currently be represented.

Chapter 7

Conclusions

In this thesis we analyzed the techniques attackers make use of to attack and infect Internet-connected devices. Once a system is compromised it is the task of security vendors and researchers to analyze the threat and provide countermeasures.

Thus, this thesis covered the problem of *web-spam*, a technique malware authors frequently make use of to increase their chances to infect Internet-connected personal computers with malware. Based on the insight we gained from conducting this research we designed and presented a technique to detect such web-spam pages.

Additionally, we presented our research on *drive-by download* attacks, where the mere visit of a web page can infect a web client with malware. Based on the observation that many drive-by attacks feature executable exploit code we presented a technique to protect unsuspecting users from web sites that launch such drive-by download attacks. Since the runtime overhead of this approach is barely noticeable when compared to the delays introduced by the network, our approach could be implemented as an integral part of protecting users from malicious web sites.

Although the iOS platform, at the time of writing, does not suffer from the proliferation of malware comparable to PCs, there are still third party applications with at least questionable behavior and functionality. Thus, in this thesis we also presented a technique to automatically analyze iOS applications with respect to privacy violations. That is, our system is able to detect applications that send

sensitive data, such as address book contents or positional data, to the network, possibly breaching the user's privacy.

Bibliography

[1] http://thebigboss.org.

[2] AppTrakr, Complete App Store Ranking. http://apptrakr.com/.

[3] iPhone Developer Program License Agreement. http://www.eff.org/files/20100302_iphone_dev_agr.pdf.

[4] C. G. Atkeson, A. W. Moore, and S. Schaal. Locally weighted learning. *Artif. Intell. Rev.*, 11(1-5):11–73, 1997.

[5] Avira Press Center. Avira warns: targeted malware attacks increasingly also threatening German companies. http://www.avira.com/en/security_news/targeted_attacks_threatening_companies.html, 2007. Last accessed, May 2010.

[6] P. Baecher and M. Koetter. x86 shellcode detection and emulation. http://libemu.mwcollect.org/. Last accessed, May 2010.

[7] A. Bifet, C. Castillo, P.-A. Chirita, and I. Weber. An Analysis of Factors Used in Search Engine Ranking. In *Adversarial Information Retrieval on the Web*, 2005.

[8] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *7th International World Wide Web Conference (WWW)*, 1998.

[9] F. Cacheda and Á. Viña. Experiencies retrieving information in the World Wide Web. In *Proceedings of the Sixth IEEE Symposium on Computers*

and Communications (ISCC 2001), pages 72–79. IEEE Computer Society, 2001.

[10] B. Calder and D. Grunwald. Reducing indirect function call overhead in c++ programs. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 397–408, New York, NY, USA, 1994. ACM.

[11] Capture-HPC Client Honeypot / Honeyclient. `https://projects.honeynet.org/capture-hpc`, 2009.

[12] A. Chaudhuri. Language-based security on android. In *ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, 2009.

[13] K. Chellapilla and D. Chickering. Improving Cloaking Detection Using Search Query Popularity and Monetizability. In *Adversarial Information Retrieval on the Web*, 2006.

[14] H. Chen, D. Dean, and D. Wagner. Model Checking One Million Lines of C Code. In *11th Annual Network and Distributed System Security Symposium (NDSS04)*, 2004.

[15] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM conference on Computer and communications security (CCS)*, pages 235 – 244, 2002.

[16] S. Chenette. ToorConX - the ultimate deobfuscator. `http://www.toorcon.org/tcx/26_Chenette.pdf`, 2008.

[17] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[18] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy (Oakland)*, 2005.

[19] A. Cohen. The iPhone Jailbreak: A Win Against Copyright Creep. http://www.time.com/time/nation/article/0,8599,2006956,00.html.

[20] Superbuddy activex control vulnerability. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5820, 2006.

[21] D. Dagon, G. Gu, C. Lee, and W. Lee. A Taxonomy of Botnet Structures. In *Annual Computer Security Applications Conference (ACSAC)*, 2007.

[22] Dan Goodin (The Register). SQL injection taints BusinessWeek.com. http://www.theregister.co.uk/2008/09/16/businessweek_hacked/, 2008. Last accessed, May 2010.

[23] M. Daniel, J. Honoroff, and C. Miller. Engineering heap overflow exploits with javascript. In *2nd USENIX Workshop on Offensive Technologies (WOOT08)*, 2008.

[24] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, 1995.

[25] W. Dormann and D. Plakosh. Vulnerability detection in activex controls through automated fuzz testing. http://www.cert.org/archive/pdf/dranzer.pdf, 2008.

[26] M. Egele, C. Kolbitsch, and C. Platzer. Removing web spam links from search engine results. *Journal in Computer Virology*, pages 1–12, 2009. 10.1007/s11416-009-0132-6.

[27] M. Egele, C. Krügel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *17th Annual Network and Distributed System Security Symposium (NDSS11), to appear*, 2011.

[28] M. Egele, T. Scholte, E. Kirda, and C. Krügel. A survey on automated dynamic malware analysis techniques and tools. *ACM Comput. Surv.*, to appear.

[29] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *DIMVA*, pages 88–106, 2009.

[30] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of OSDI 2010*, October 2010.

[31] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *IEEE Security and Privacy*, 7(1):50–57, 2009.

[32] M. P. Evans. Analysing Google rankings through search engine optimization data. *Internet Research Vol. 17 No. 1*, 2007.

[33] B. Feinstein and D. Peck. Caffeine monkey: Automated collection, detection and analysis of malicious javascript. `http://www.dc414.org/download/confs/defcon15/Speakers/Feinstein_and%20_Peck/Whitepaper/dc-15-feinstein_and_peck-WP.pdf`, 2006.

[34] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy*, pages 194 – 208, 2004.

[35] D. Fetterly, M. Manasse, and M. Najork. Spam, damn spam, and statistics: Using statistical analysis to locate spam web pages. In *WebDB*, pages 1–6, 2004.

[36] M. Foundation. SpiderMonkey (JavaScript-C) Engine. `http://www.mozilla.org/js/spidermonkey/`.

[37] J. Freeman. `http://cydia.saurik.com/`.

[38] S. Frei, T. Dübendorfer, G. Ollmann, and M. May. Understanding the web browser threat. Technical Report 288, ETH Zurich, 06 2008. 2008.

[39] FRISK Software International. F-prot virus signature updates cause false alarm in Windows 98. `http://www.f-prot.com/news/vir_alert/falsepos_invictus.html`, 2003. Last accessed, May 2010.

[40] Gartner Newsroom. Competitive Landscape: Mobile Devices, Worldwide, 2Q10. `http://www.gartner.com/it/page.jsp?id=1421013`, 2010.

[41] Google. Zeitgeist: Search patterns, trends, and surprises. `http://www.google.com/press/zeitgeist.html`. Last accessed, 29.06.2009.

[42] Google Keeps Tweaking Its Search Engine. `http://www.nytimes.com/2007/06/03/business/yourmoney/03google.html?pagewanted=4&_r=1`. Last accessed, 29.06.2009.

[43] B. Gregg. fetch application data from snoop or tcpdump logs. `http://chaosreader.sourceforge.net/`.

[44] Z. Gyöngyi and H. Garcia-Molina. Web Spam Taxonomy. In *Adversarial Information Retrieval on the Web*, 2005.

[45] O. Hallaraker and G. Vigna. Detecting malicious javascript code in mozilla. In *10th International Conference on Engineering of Complex Computer Systems (ICECCS 2005)*, pages 85–94, 2005.

[46] M. A. Hearst. Support vector machines. *IEEE Intelligent Systems*, 13(4):18–28, 1998.

[47] D. Heckerman. A tutorial on learning with bayesian networks. Technical report, Microsoft Research, 1995.

[48] B. J. Jansen and A. Spink. An analysis of web searching by european AlltheWeb.com users. *Information Processing and Management*, 41(2):361–381, 2005.

[49] G. H. John and P. Langley. Estimating continuous distributions in bayesian classifiers. In *UAI '95: Proceedings of the Eleventh Annual Conference*

on Uncertainty in Artificial Intelligence, August 18-20, 1995, Montreal, Quebec, Canada, pages 338–345, 1995.

[50] John Leyden. Drive-by download attack compromises 500k websites. `http://www.channelregister.co.uk/2008/05/13/zlob_trojan_forum_compromise_attack/`. Last accessed, February 2009.

[51] John Leyden (The Register). Kaspersky false alarm quarantines Windows Explorer. `http://www.channelregister.co.uk/2007/12/20/kaspersky_false_alarm/`, 2007. Last accessed, May 2010.

[52] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*, 2006.

[53] V. G. Kaburlasos, I. N. Athanasiadis, and P. A. Mitkas. Fuzzy lattice reasoning (flr) classifier and its application for ambient ozone estimation. *Int. J. Approx. Reasoning*, 45(1):152–188, 2007.

[54] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamalytics: an empirical analysis of spam marketing conversion. In *ACM Conference on Computer and Communications Security (CCS)*, pages 3–14, 2008.

[55] C. Karlberger, G. Bayler, C. Kruegel, and E. Kirda. Exploiting Redundancy in Natural Language to Penetrate Bayesian Spam Filters. In *First USENIX Workshop on Offensive Technologies (WOOT07)*, 2007.

[56] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *IEEE Symposium on Security and Privacy*, pages 314–327, 2006.

[57] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *14th USENIX Security Symposium*, 2005.

[58] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *14th USENIX Security Symposium*, 2005.

[59] J. B. MacQueen. Some Methods for classification and Analysis of Multivariate Observations. *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[60] Microsoft Corporation. Microsoft Security Bulletin MS06-014 - Vulnerability in the Microsoft Data Access Components (MDAC) Function Could Allow Code Execution. `http://www.microsoft.com/technet/security/Bulletin/MS06-014.mspx`, 2006. Last accessed, May 2010.

[61] Microsoft Corporation. Microsoft Security Bulletin MS08-067 - Critical; Vulnerability in Server Service Could Allow Remote Code Execution. `http://www.microsoft.com/technet/security/Bulletin/MS08-067.mspx`, 2008. Last accessed, May 2010.

[62] D. Moore, C. Shannon, and K. C. Claffy. Code-Red: a case study on the spread and victims of an Internet worm. In *Internet Measurement Workshop*, pages 273–284, 2002.

[63] D. Moore, G. Voelker, and S. Savage. Inferring Internet Denial of Service Activity. In *Usenix Security Symposium*, 2001.

[64] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symposium on Security and Privacy, Oakland*, May 2007.

[65] M. D. Network. ActiveX Controls. `http://msdn.microsoft.com/en-us/library/aa751968.aspx`.

[66] M. D. Network. JScript (Windows Script Technologies. `http://msdn.microsoft.com/en-us/library/hbxc2t98.aspx`.

[67] Y. Niu, Y.-M. Wang, H. Chen, M. Ma, , and F. Hsu. A quantitative study of forum spamming using context-based analysis. In *NDSS*, 2007.

[68] A. Ntoulas, M. Najork, M. Manasse, and D. Fetterly. Detecting Spam Web Pages through Content Analysis. In *15th International World Wide Web Conference (WWW)*, 2006.

[69] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31, 1999.

[70] J. C. Platt. Fast training of support vector machines using sequential minimal optimization. pages 185–208, 1999.

[71] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level polymorphic shellcode detection using emulation. In *Detection of Intrusions and Malware & Vulnerability Assessment, Third International Conference (DIMVA)*, pages 54–73, 2006.

[72] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In *Recent Advances in Intrusion Detection, 10th International Symposium (RAID)*, pages 87–106, 2007.

[73] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level polymorphic shellcode detection using emulation. *Journal in Computer Virology*, 2(4):257–274, 2007.

[74] M. Polychronakis and N. Provos. Ghost turns zombie: Exploring the life cycle of web-based malware. In *First USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2008.

[75] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iFRAMEs point to us. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[76] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost In The Browser: Analysis of Web-based Malware. In *First Workshop on Hot Topics in Understanding Botnets (HotBots '07)*, 2007.

[77] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost In The Browser Analysis of Web-based Malware. In *First Workshop on Hot Topics in Understanding Botnets (HotBots '07)*, 2007.

[78] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[79] Rahul Mohandas (McAfee Avert Labs). Analysis of Adversarial Code: The role of Malware Kits! `http://clubhack.com/2007/files/Rahul-Analysis_of_Adversarial_Code.pdf`, December 2007. Last accessed, 29.06.2009.

[80] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *13th Systems Administration Conference (LISA)*, 1999.

[81] J. Rutkowska. Introducing Blue Pill. `http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html`, 2006. Last accessed, May 2010.

[82] Secunia PSI study: 28% of all detected applications are insecure. `http://secunia.com/blog/11/`, 2007.

[83] Google Search Engine Ranking Factors. `http://www.seomoz.org/article/search-ranking-factors`. Last accessed, 29.06.2009.

[84] N. Seriot. iPhone Privacy. `http://www.blackhat.com/presentations/bh-dc-10/Seriot_Nicolas/BlackHat-DC-2010-Seriot-iPhone-Privacy-slides.pdf`.

[85] H. Shi. Best-first decision tree learning. 2007.

[86] E. Skoudis and L. Zeltser. *Malware: Fighting Malicious Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[87] A. Sotirov. Heap feng shui in javascript. http://www.phreedom.org/research/heap-feng-shui/heap-feng-shui.html. Last accessed, May 2010.

[88] A. Sotirov. Heap Feng Shui in JavaScript. http://www.phreedom.org/research/heap-feng-shui/heap-feng-shui.html. Last accessed, November 2008.

[89] E. H. Spafford. The Internet worm incident. In *Proceedings of the 2nd European Software Engineering Conference*, pages 446–468, 1989.

[90] Spamcop - the premier service for reporting spam. http://www.spamcop.net/.

[91] S. Stasiukonis. Social engineering, the USB way. http://www.darkreading.com/security/perimeter/showArticle.jhtml?articleID=208803634, 2007. Last accessed, May 2010.

[92] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. A. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: analysis of a botnet takeover. In *ACM Conference on Computer and Communications Security (CCS)*, pages 635–647, 2009.

[93] K. Svore, Q. Wu, C. Burges, and A. Raman. Improving Web Spam Classification using Rank-time Features. In *Adversarial Information Retrieval on the Web*, 2007.

[94] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[95] S. Tanachaiwiwat and A. Helmy. Vaccine: War of the worms in wired and wireless networks. In *IEEE Infocom 2006, Poster*, 2006.

[96] T. Toth and C. Krugel. Accurate buffer overflow detection via abstract payload execution. In *RAID*, pages 274–291, 2002.

[97] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *ACM Conference on Programming Language Design and Implementation*, 2009.

[98] P. Vogt, F. Nentwich, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *14th Annual Network and Distributed System Security Symposium (NDSS 2007)*, 2007.

[99] Y.-M. Wang, M. Ma, Y. Niu, and H. Chen. Spam Double-Funnel: Connecting Web Spammers with Advertisers. In *16th International Conference on World Wide Web*, 2007.

[100] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[101] Wired. Apple Approves, Pulls Flashlight App with Hidden Tethering Mode. `http://www.wired.com/gadgetlab/2010/07/apple%2dapproves%2dpulls%2dflashlight%2dapp%2dwith%2dhidden%2dtethering%2dmode/`.

[102] I. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2nd edition edition, 2005.

[103] B. Wu and B. Davison. Cloaking and Redirection: A Preliminary Study. In *Adversarial Information Retrieval on the Web*, 2005.

[104] B. Wu and B. D. Davison. Identifying Link Farm Spam Pages. In *14th International World Wide Web Conference (WWW)*, 2005.

[105] J. Zhuge, T. Holz, C. Song, J. Guo, X. Han, and W. Zou. Studying malicious websites and the underground economy on the Chinese web. In *7th Workshop on Economics of Information Security 2008*, 2008.

[106] D. D. Zovi. Hardware Virtualization Based Rootkits, 2006. in Black Hat Briefings and Training USA 2006.

Curriculum Vitae

17. Februar 1982	Geboren in Dornbirn/Vorarlberg
1992 - 2000	Bundesrealgymnasium Dornbirn
2000 - 2006	Diplomstudium Informatik an der TU Wien
12/2006	Verleihung des akademischen Grades Dipl. Ing. Diplomarbeit “Behavior-Based Spyware Detection Using Dynamic Taint Analysis”.
2007 - 2010	Doktoratsstudium am International Secure Systems Lab der TU Wien Betreuer: Privatdozent Dipl.-Ing. Dr. Christopher Krügel und Privatdozent Dipl.-Ing. Dr. Engin Kirda