SECURITY AND PRIVACY MECHANISMS FOR WEB AND
THIRD-PARTY APPLICATIONS

A Dissertation in
Computer Science and Engineering
by
Wei Xu

© 2013 Wei Xu

Submitted in Partial Fulfillment
of the Requirements
for the Degree of
Doctor of Philosophy

August 2013
The dissertation of Wei Xu was reviewed and approved* by the following:

Sencun Zhu  
Associate Professor of Computer Science and Engineering  
Dissertation Advisor, Chair of Committee

Heng Xu  
Associate Professor of Information Science and Technology

Guohong Cao  
Professor of Computer Science and Engineering

David Miller  
Professor of Electrical Engineering

Lee Coraor  
Associate Professor of Computer Science and Engineering  
Graduate Officer

*Signatures are on file in the Graduate School.
Abstract

Web and third party applications are new forms of computer applications that heavily leverage the Internet and are compatible with various platforms such as PCs, tablets and smartphones. These applications are rapidly growing and are widely deployed. Many functionalities provided by these applications have been frequently used as alternatives to traditional host-based computer programs. A large user base reflects the unprecedented popularity of these applications. Unfortunately, the popularity of Web and third party applications has also drawn the attention of attackers, who exploit the vulnerabilities within these applications and pose great security and privacy threats to the users. To defend against these threats, this dissertation proposes several security defense and privacy protection mechanisms for Web and third party applications. The mechanisms proposed in this dissertation focus on two most popular platforms, i.e., Web browsers and smartphones. On each platform, different approaches are developed to protect users from the most severe and the most representative threats observed in the real world.

The two most representative types of threats on Web browsers are JavaScript-based malicious Web pages and worm propagation in online social networks. As the top Internet security threat in recent years, malicious JavaScript code often applies obfuscation techniques to hide its malicious purpose and to evade the detection of anti-virus software. To this end, we proposed an approach called JStill that can detect and prevent the execution of the obfuscated malicious JavaScript code in Web browsers. The propagation of worms in online social network websites such as Facebook is new attack vector. Attackers leverage social connections and social engineering in online social networks to facilitate the propagation of worms. Given the new features in the propagation of worms, we proposed an early warning detection system that can detect worms in online social networks when only a small number of user accounts are infected.

Android is the most popular smartphone operating system. Android system enforces an installation-time permission check mechanism, which can not effectively
prevent sensitive permissions from being granted to malicious applications. In view of this issue, we propose Permlyzer, a framework to automatically generate analysis on the use of permissions in applications. The information obtained by Permlyzer can not only help users to make informed decision before installation, but also help application vendors to vet applications before releasing to the public.
# Table of Contents

List of Figures ........................................... ix

List of Tables ........................................... xi

Acknowledgments ........................................... xii

Chapter 1
Introduction ........................................... 1
  1.1 Web and Third-Party Applications ..................... 1
  1.2 Security and Privacy Threats ......................... 2
  1.3 Contributions ....................................... 4

Chapter 2
Literature Study ......................................... 6
  2.1 Malicious JavaScript Detection ....................... 6
    2.1.1 Malicious JavaScript Detection ................ 6
    2.1.2 Obfuscated Malicious JavaScript Code Detection 7
  2.2 Worm Detection .................................... 9
  2.3 Android Security .................................. 11
    2.3.1 Android Permission Analysis .................. 11
    2.3.2 Smartphone Platform Security ................. 11
    2.3.3 Application Security ......................... 13
    2.3.4 Malicious Application Characterization and Detection 13

Chapter 3
Detection of Obfuscated Malicious JavaScript Code .... 15
  3.1 The Power of Obfuscation Techniques in Malicious JavaScript Code 16
    3.1.1 Categorization of the Observed Obfuscation Techniques in Malicious JavaScript Code 17
Chapter 5

**Automatic Analysis of the Use of Permissions in Android Apps**

5.1 Android Permission Model and Android Applications
- 5.1.1 Android Permission Model
- 5.1.2 Android Permission Declaration and Enforcement
- 5.1.3 Android Applications

5.2 System Overview

5.3 Automated Analysis of Permission Use
- 5.3.1 Functionality Exploration
- 5.3.2 Call Stacks Construction
- 5.3.3 Analysis of Permission Use
  - 5.3.3.1 The Location of Permission Use
  - 5.3.3.2 The Cause of Permission Use
  - 5.3.3.3 The Purpose of Permission Usage
  - 5.3.3.4 Evaluation of Potential Risks in Permission Use

5.4 Analysis of Permission Usage in Android Applications
- 5.4.1 Collection of Android Applications
- 5.4.2 Permission Analysis Coverage
- 5.4.3 Performance Analysis
- 5.4.4 Permission Use in Malicious Applications
  - 5.4.4.1 Malicious Behavior and Permission Requests
  - 5.4.4.2 Characterizing Permission Use in Malicious Applications
- 5.4.5 Permission Use in Free Applications
  - 5.4.5.1 Android Permissions Requests
  - 5.4.5.2 Permission Use in Free Android Applications
  - 5.4.5.3 Security/Privacy Risks in Free Applications’ Permission Use

5.5 Summary

Chapter 6

**Conclusion**

Appendix A

**A.1 Evading Effectiveness**
A.2 Algorithms

Appendix B

B.1 D-Gen and R-Eval Functions hooked in the implementation of JStill
B.2 JavaScript Obfuscation Tools

Bibliography
List of Figures

3.1 An Example of Randomization Techniques. (a) is the original code and (b) is the obfuscated code ........................................... 17
3.2 Examples of string data obfuscation. (a) is the original code; (b) uses string splitting obfuscation; (c) uses keyword substitution ... 18
3.3 An example of obfuscation using hexadecimal representation. ... 19
3.4 An example of obfuscation using encoding/decoding functions. (a) is the encoding function; (b) is the obfuscated code. .......... 20
3.5 Examples of logic structure obfuscation. (a) inserts independent instructions; (b) uses additional conditional branches. .......... 20
3.6 The Distribution on the Number of Obfuscation Technique Categories employed in samples .................................................. 22
3.7 The Detection Rates of 20 Anti-virus Software on Our Sample Set. 24
3.8 The Detection Rate of 20 Anti-Virus Software on Samples Obfuscated by Randomization ...................................................... 25
3.9 The Detection Rate of 20 Anti-Virus Software on Samples Obfuscated by Data Obfuscation .................................................... 26
3.10 Overview of JStill ................................................................. 28
3.11 Examples of disguised invocations of language-defined functions . 30
3.12 Example of legitimate function invocations in JavaScript ........... 32
3.13 Identify function invocations via bytecode .............................. 33
3.14 A Search Tree for Basic Strings .............................................. 38
3.15 An example of disrupting benign execution ............................ 40
3.16 Average Performance Overhead for Top 20 Websites ............... 45

4.1 Koobface worm infection cycle ............................................ 49
4.2 Detection System Overview .................................................. 51
4.3 An example of two level correlation ...................................... 57
4.4 OSN Worm Simulation Model .............................................. 59
4.5 Infection Number versus Different Initial User for Koobface Worm 61
4.6 Infection Number versus Different Initial User Accounts .......... 61
4.7 Infection Number versus Different Percentages of Friends lists ... 63
List of Tables

3.1 The usage of JavaScript Obfuscation Techniques . . . . . . . . . . . . 22
3.2 The Selection of Anti-virus Software . . . . . . . . . . . . . . . . . 23
3.3 False Positives and False Negatives in Malicious Obfuscation De-
tection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.4 Composition of False Positives . . . . . . . . . . . . . . . . . . . . 42

4.1 Infection Number on Detection for Koobface and Mikeyy Worms . . 60

5.1 Analysis of Permission Use in Malicious Applications . . . . . . . . 83
5.2 Analysis of the Most Representative Purposes of Permission Use in
Malicious Applications . . . . . . . . . . . . . . . . . . . . . . . . . 85
5.3 Cause of Permissions in Malicious Applications . . . . . . . . . . . . 86
5.4 Requested Android Permissions in Each App Category . . . . . . . 88
5.5 Analysis of Top Ten Most Checked Permission Use in Free Apps . . 89
5.6 Permissions Use Caused by Third-party Libraries . . . . . . . . . . . 91

B.1 D-Gen and R-Eval Function hooked by JStill . . . . . . . . . . . . . 97
B.2 JavaScript Obfuscation Tools . . . . . . . . . . . . . . . . . . . . . 98
Acknowledgments

I want to express my most sincere gratitude to my advisor, Dr. Sencun Zhu for his continued guidance, warm encouragement and invaluable advice through the doctoral program. It is his knowledge and his patience that helped me in the face of many challenging issues. I could not think of a better role model than Dr. Zhu and it has been my great honor to work with him.

I am also grateful to my other committee members, Dr. Heng Xu, Dr. Guohong Cao and Dr. David Miller for their time and helpful suggestions.

Finally, I would like to thank my family. My parents have inspired me with the love of science and technology and encouraged me in all my pursuits. My beloved wife has provided me with all the support, care and joy that I could ever ask for. Thank you.
Dedication

To my wife, Fangfang and my beloved son, Ethan
Chapter 1

Introduction

1.1 Web and Third-Party Applications

Web and third-party applications are new forms of computer programs that leverage the Internet and different platforms such as PCs, tablets, smartphones, etc. to provide various functionalities to facilitate users’ daily life. For example, online social networks (OSN) such as Facebook and Twitter are Web applications that are built on top of Web browsers and are across different platforms. Third-party smartphone applications (a.k.a apps) are programs developed for the smartphone platforms and intensively use the resources on the Internet. Numerous users use these Web and third-party applications on a daily basis. For example, Facebook has more than 500 million active users [1]; Twitter has 75 million active users [2]) and Android smartphone platform has over 400 million users [3].

Both the popularity and the diversity of Web and third-party applications are unprecedented. Many applications have attracted a substantially large user base. Different developer’s communities have become prosperous. Various techniques used in the development of these applications are rapidly emerging and evolving. New features and functionalities are proposed and implemented on a daily basis. In many ways, Web and third party applications have started to change the daily life of users.
1.2 Security and Privacy Threats

With the popularity and diversity of Web and third-party applications, various security and privacy threats have been observed in recent years targeting these applications. Among different threats, the most representative and widely observed ones are: malicious JavaScript code-based threats, worm posed threats and malicious app-based threat. Each of these three types of threats has its own unique characteristics.

Malicious JavaScript-based threats such as XSS, CSRF, Drive-by-download, etc. that exploit the vulnerabilities in JavaScript-based Web applications have been reported as the top Internet security threats in recent years [4] [5] [6]. The popularity of this type of threat is largely due to the dominant position of the JavaScript language in Web applications. JavaScript code is seen in most if not all the Web pages because JavaScript language has many properties that can be used to develop various dynamic features of Web applications. For example, JavaScript language supports dynamic generation that can generate JavaScript code from text. It also supports runtime execution that can execute text strings as JavaScript code in runtime. These are the features that enable the development of asynchronous JavaScript and XML (Ajax) framework. Meanwhile, these features also facilitate the creation of obfuscated malicious JavaScript code. Obfuscated malicious JavaScript code poses a more severe threat than un-obfuscated code because most of the anti-virus programs adopted by end users rely on signature-based detection that can be easily evaded by obfuscation techniques. Therefore, even after a piece of malicious JavaScript code has been detected, it can still be re-used after being obfuscated. Moreover, because JavaScript language is an interpreted language, JavaScript code is essentially text before execution. Various text-based obfuscation technique can be directly applied on JavaScript code.

Computer worms have been known as one the first forms of security attacks. However, within new Web applications such as OSN, worms have evolved into new attack forms and have been able to infect more users. For example, the OSN worm “koobface” used the messages function within OSN websites to propagate itself. This worm infected users from more than eight most famous OSN websites including Facebook, MySpace, etc. Unlike traditional computer worms, OSN worms
exploit the characteristic of OSN websites. The small-world network property and the scale-free network property of the underlying social graph provide OSN worms an opportunity to reach millions of user within a limited number of hops. OSN worms also leverage social engineering to achieve more robust infection cycles. The worm messages are customized with personal information of the target users. The information is harvested from the personal profiles on OSN websites. Such worm messages have higher click-through rate on the embedded malicious content (e.g., a link to malicious websites). As a result, the exponential increase of the number of infected users by OSN worm has caused havoc among the users and great impact on the popularity and function of OSN websites. Meanwhile, the new infection technique can not be detected by existing Internet worm detection approaches that rely on the evidence generated by worms in network traffic or the behavior of infected hosts, because no such evidence is generated in the propagation of OSN worms. Moreover, the security impact caused by OSN worms is not limited to infecting OSN users. OSN worms have also been observed to be integrated with other organized security attacks such as botnet to deliver selective malicious payloads to infected hosts. Therefore, OSN worms are one of the most representative security threats posed against Web and third-party applications.

Smartphone app-based threats target the smartphone platforms, among which Andriod is the most popular operating system. According to Google [3], Android users downloaded more than 1.5 billion apps from Google Play store each month. One of the reasons behind Android’s popularity is the comprehensive API framework for app development. These APIs have access to different resources and services in the platform, e.g., file systems, location sensors, system settings, etc. Many of the resources and services hold private information or are security sensitive. Therefore, Android applies a permission-based checking mechanism to protect these resources and services. Only after necessary permissions are obtained can an API access protected resources. The permissions must be explicitly requested by the apps at the beginning of the installation and the requests have to be granted by users at the same time. One issue with permission-based mechanism is that users do not have enough detailed information about the purpose and the use of the requested permissions to make informed decision when granting the permissions. Because Android permission mechanism provides users with only granting
all or terminating installation, most of the permission requests are granted without careful evaluation. This is one of the root causes to the failure of permission-based protection scheme in Android.

1.3 Contributions

The major contributions of this dissertation are as follows:

* Mostly Static Detection of Obfuscated Malicious JavaScript Code
  - Mostly static detection of obfuscated malicious code. This dissertation presents JStill, a lightweight, mostly static approach to detect obfuscated malicious JavaScript code, most of which can evade the detection of state-of-art anti-virus software. JStill can not only detect, but also prevent the execution of obfuscated malicious JavaScript code in a user’s Web browser.

  - Function invocation-based analysis This dissertation presents a function invocation-based analysis technique that leverages the combination of static analysis and runtime inspection. Our analysis is based on various different aspects of function invocations so that it can effectively distinguish malicious obfuscated JavaScript code from benign code.

  - Realtime protection system The performance overhead of JStill in terms of averaged increased webpage loading time is very small, making it a practical intrusion prevention system (IPS).

* Early Warning OSN Worm Detection System
  - Early warning detection system The OSN worm detection system proposed in this dissertation achieves early warning. Early warning detection can provide administrators the opportunity to apply worm containment and elimination measures without affecting a large portion of user accounts.

  - Two level detection system This dissertation presents a surveillance network leveraging the small-world and scale-free properties of the social graph to collect suspicious worm propagation evidence. This dissertation also proposed an algorithm to maximize the surveillance coverage by monitoring only
a small fraction of user accounts. This dissertation further proposed a two-level scheme to effectively filter out the noise in the surveillance network.

- **Real life social graph evaluation** Experimental results are based on a real life social graph dataset consisting of over 1.8 million users and 22.6 million friend links. The results demonstrated the effectiveness of the proposed detection system. The outbreak of an OSN worm can be detected when only 0.13% of the total user accounts are infected.

**Automatic Analysis of the Use of Permissions in Android Applications**

- **Automatic Analysis** This dissertation proposes and implements Permlyzer, a framework for automatically analyzing the use of permissions in Android applications. Permlyzer is evaluated by analyzing the characteristics in the use of permissions among 51 malware and spyware application families and over 110,000 free applications.

- **Call stack-based analysis** This dissertation proposes a call stack-based analysis approach to extract fine-grained information about various aspects of the use of permissions including location, cause and purpose.

- **Functionality exploration** This dissertation proposes a functionality exploration scheme that can automatically explore the functionality of Android applications. This dissertation also develops an algorithm to efficiently construct call stacks. Using both techniques, Permlyzer can automatically and efficiently analyze the use of permissions on a large scale.

- **First automatic permission analysis tool** To the best of my knowledge, Permlyzer is the first tool that can automatically discover the characteristics of the use of permissions from a large set of applications.
Chapter 2

Literature Study

This chapter reviews existing security and privacy mechanisms for Web and third-party applications.

2.1 Malicious JavaScript Detection

2.1.1 Malicious JavaScript Detection

Many approaches have been proposed to detect the malicious JavaScript code in Web pages. Hallarakar et al. [7] proposed an auditing system to examine the execution of JavaScript code on the client-side. To detect malicious JavaScript code, the audited code is compared with policies that specify the suspicious activities. For example, a policy can specify the state-transition happened when a cookie from a trusted website is misused by another malicious website. This policy can detect a type of cross-site scripting attacks. However, this approach requires knowledge about the attacks to define policies. Therefore, this approach can not detect malicious JavaScript code when the behavior of the code is unknown. It is also not a scalable solution because one policy can only detect one specific attack. Egele et al. [8] proposed an approach to automatically detecting drive-by downloads by identifying JavaScript string buffers that contain shellcode through x86 instruction emulation. Their approach focuses on the detection of drive-by download attacks that require client-side JavaScript code to allocate strings of x86 code (i.e., shell code). This approach can not detect other malicious JavaScript-based attacks. Yu
et al. [9] proposed a scheme to instrument suspicious JavaScript code and to confine run-time behavior of these instrumented code by customized security policies. A similar scheme was proposed by Reis et al. [10]. In their scheme, JavaScript code that is embedded in Web pages is rewritten into script that has equivalent functionality. The rewritten code is inserted with checks that will be invoked during run-time. This work provides a framework to check the run-time behavior of the JavaScript code against pre-defined security policies. However, when the behavior of malicious JavaScript code has not been observed previously, this approach can not detect the malicious code. Curtsinger et al. [11] proposed Zozzle, a JavaScript malware detection system that is based on features extracted from abstract syntax trees (AST) of JavaScript code. Zozzle focuses on detecting JavaScript malwares that contain heap spray and shellcode.

In summary, existing approaches that detect malicious JavaScript code either focus on specific attacks or require pre-knowledge of the malicious behavior exhibited by JavaScript code in run-time. Therefore, these approaches can not detect various malicious JavaScript code in general.

2.1.2 Obfuscated Malicious JavaScript Code Detection

Previous research on the detection of obfuscated malicious JavaScript falls into two general categories, namely static approaches and dynamic approaches. The static approaches often resort to static heuristics to discover obfuscated malicious JavaScript code. Likarish et al. [12] proposed leveraging classification techniques to detect obfuscated malicious JavaScript code. The features they used in their classifier include the length of the code, the number of strings in the code, the percentage of whitespaces in strings, etc. However, the combination of these features can not effectively distinguish obfuscated malicious JavaScript code from benign code because these features are also extensively used by benign code. Choi et al. [13] proposed an approach to detecting the obfuscated strings in malicious JavaScript code based on the characteristics of the obfuscated string objects, such as excessive usage of specific characters and excessive length of the strings. However, obfuscated string is only one type of obfuscation techniques adopted by malicious JavaScript code. Besides, obfuscated strings are also used by benign code. Seifert
et al. [14] proposed a static heuristic-based approach to detecting malicious Web pages. In their approach, JavaScript obfuscation is characterized by the encoded string values and the visibility and the size of iframes. Their approach can only detect string manipulation related obfuscated malicious JavaScript code. Kim et al. [15] proposed an entropy-based approach to detect obfuscated malicious JavaScript. However, the entropy of JavaScript strings varies in different obfuscation techniques. In some obfuscation techniques such as randomization obfuscation, the entropy of obfuscated malicious JavaScript code is similar to that of benign JavaScript code.

In summary, there are two limitations of previous static approaches. The first limitation is that these approaches all rely on the features that are also heavily adopted by benign JavaScript code. Without identifying the discrepancy between the purpose of malicious and benign JavaScript code, the manifestation of above static heuristics is prone to high false positive rate. The second limitation is that these approaches only focus on the detection of specific types of obfuscation techniques. Therefore, these approaches can not detect all obfuscated malicious JavaScript code.

Dynamic approaches detect obfuscation by examining the executed code in runtime. Cova et al. [16] proposed a detection scheme for malicious JavaScript code. Their scheme detects the obfuscated malicious JavaScript code by extracting features such as the ratio of string definitions and string uses, the length of dynamic code generation during the execution. These features, however, can be evaded by attackers by applying obfuscation techniques that manifest differently, for example, randomization obfuscation. Kaplan et al. [17] proposed “NoFus”, which also leverages an AST-based static classifier to detect if a piece of JavaScript code has been obfuscated for any purpose. That is, their approach is to detect the existence of obfuscation. However, obfuscation has been observed in both benign and malicious JavaScript code. Therefore, the existence of obfuscation techniques can not indicate the maliciousness of the JavaScript code.

Many dynamic analysis tools have been proposed as well to examine the execution of the JavaScript code. For example, Kolbitsch et al. [18] proposed a JavaScript virtual machine that can explore multiple execution paths in a single execution. Their approach can mitigate fingerprinting techniques adopted by ma-
licious JavaScript and save virtual machine resources in the dynamic analysis of JavaScript code. However, the existence of fingerprinting technique does no prove the maliciousness of JavaScript code. Spiffy [19] emulates a Web browser without the rendering functionality, so that obfuscated JavaScript code can be uncovered during execution. However, this is done by manual check in Spiffy after outputting the execution output and the arguments of function invocations. ToorConx [20] modifies the dlls of IE so that it can hook the most exploited functions (e.g., “eval” and “document.write”) to output the contents of the arguments passed to these functions. Again, it can not automatically identify whether the arguments are obfuscated or not. Other similar analysis tools include Caffeine Monkey [21], JSUNPACK [22].

In summary, these dynamic analysis tools have two limitations in the detection of obfuscated malicious JavaScript code. The first limitation is that some event-triggered JavaScript code may not be executed by these execution-based dynamic analysis tools because of the lack of the corresponding events (e.g., mouse move). The other limitation is that these tools can not distinguish benign obfuscated code with malicious obfuscated code because the features revealed by these tools are observed in both benign and malicious JavaScript code.

2.2 Worm Detection

In the area of Internet worm early detection, various detection strategies have been proposed. Gu et al. proposed a detection algorithm based on local victim information [23]. In their approach, they used destination-source correlation to capture the patterns in incoming and outgoing scanning traffic of a host before and after the host is infected by a scan-based worm. They also looked for a worm’s anomalous scanning patterns, such as high scan rate to identify the outbreak of the worm. However, their approach can not be applied to OSN worm detection because no such scan traffic is presented in the propagation of OSN worms. Dagon et al. proposed a detection technique [24] using honeypots to monitor the entire infection process (an infection cycle) rather than just the beginning and the end. They recorded memory events, network events and disk events to perform logistic analysis looking for correlations. Their approach requires no signature in advance and
has the advantage of coping with polymorphic worms. However, lack of infection processes in OSN worms prevents applying their approach here.

Bu et al. suggested a worm detection scheme [25] that is based on the extraction of the alteration of arrival unsolicited scan rates in the early stage of worm propagation. Their work suggested a novel signal indicating the outbreak of an Internet worm, but this approach suffers from the problem of too many potential sources for false positive rate. Wagner et al. provided an entropy-based worm detection algorithm [26]. They utilized entropy to quantify the difference of randomness observed in worm traffic and in normal traffic. The source IP address fields will be less random in worm traffic than in normal traffic since the scanning hosts’ IP addresses were seen more than other hosts. Their strategy offered an alternative way to detect the propagation activities of an Internet worm. However, both of these approaches rely on the characteristics exhibited in worms’ scanning traffic. For an OSN worm, no scan is performed by the worm and the infection traffic is relatively simple compared to that of internet worm. Unlike packets with various attributes transmitted during the propagation of Internet worms, OSN worms merely generate messages. Moreover, there is no hierarchical structure in the organization of a social network. All peers are equal in the social graph, which means no auxiliary information is available for determining the location of a worm detector.

There are some other worm detection algorithms that are not based on scanning traffic. Wang et al. proposed an anomalous payload-based worm detection algorithm [27], a worm propagation can be identified if correlations of ingress and egress payload are observed. In an OSN worm, the actual payload is downloaded in the browser, which cannot be observed from the OSN websites. This is actually exploited by OSN worms to bypass any filtering-based detection scheme deployed in OSN websites.

In summary, existing approaches of detecting Internet worms can not be applied in the detection of OSN worms because OSN worms exhibit very little worm activities (e.g., no infection procedures, no scanning traffic, no worm traffic in the network packets sent/received by the hosts) in the view of the infected hosts. Therefore, any approaches relying on the information collected from the infected hosts can not be applied in the detection of OSN worms. Moreover, there is no
existing approach to detect the propagation of OSN worms.

2.3 Android Security

Many approaches have been proposed in the area of Android security. These approaches are discussed from four categories in the Section.

2.3.1 Android Permission Analysis

This category includes advancements in analyzing Android permissions. Enck et al. proposed a permission analysis system named Kirin [28]. Kirin maps dangerous functionalities with the permissions required to perform them after specifying permission-based security rules. However, Kirin only focuses on the semantics of the security permissions. Therefore, it is prone to high false positives. Barrera et al. [29] studied the permission usage among a variety of categories of applications in Android market by mapping an application to a category based on its requested permissions. Their approach is based on the semantics of Android permissions, it did not address the issue of understand the use of permissions within an application. Felt et al. [30] manually compare the functionalities of 36 Android applications to the permissions requested by these applications. Their results show that 4 out of 36 applications are over-privileged. In the same work, Felt et al. [31] also proposed Stowaway, which identifies over-privileged applications by detecting unnecessary permissions for API calls in applications. The mapping provided in [31] is very helpful, but the mapping alone can not explain the causes and the purposes of the use of permissions.

In summary, existing works in Android permission analysis can not provide information about the use of the permissions.

2.3.2 Smartphone Platform Security

This category includes a wide range of approaches [32, 33, 34, 35, 36, 37, 38, 39, 40, 41] that aim at improving the security and privacy of smartphone platforms. For example, Enck et al. proposed TaintDroid [37]. TaintDroid provides a scheme to
monitor a third-party application’s use of sensitive information such as what information leaves a device and where the information is sent to. TaintDroid leverages Android system’s visualization to implement a four-level granularity (e.g., variable, method, message and file levels) taint propagation scheme to achieve low performance overhead. TaintDroid is used to analyze 30 Android applications for potential private information leakage. Andrus et al. proposed “Cells” [32]. Cells leverages lightweight OS visualization to provide a visualization architecture that allows multiple virtual smartphones to run on one physical phone at the same time in an isolated manner. Cells isolates virtual smartphones from each other so that malicious applications in one virtual smartphone can not affect other virtual smartphones. Cells provides both kernel-level and user-level device namespace mechanisms so that hardware devices can be effectively multiplexed among different virtual smartphones. Lange et al. proposed “L4Android” [39], which also provides isolated virtual smartphone OS environments to enhance the security of the platform. Fuchs et al. proposed “SCanDroid” [36]. SCanDroid automatically extracts security specifications to check whether data flows are consistent with the specifications. SCanDroid analyzes the data flows in an application statically. It leverages string analysis, pointer analysis and intra-procedural data flow analysis to generated a call graph of all the methods and to identify intra-component data flows. It then generates permission constrains induced by the data flows. Nauman et al. proposed Android permission extension “Apex” framework [40]. Apex allows users to selectively grant permissions to applications. It also allows user to specify detailed runtime constrains to restrict the use of sensitive resources. Zhou et al. proposed “TISSA” [41]. TISSA provides users with fine-grained control over private information and resources being access by third-party applications. TISSA adds another layer of checking on top the existing Android permission model so that it can enforce security policies that are defined by users. Grace et al. proposed “Woodpecker” [38]. Woodpecker detects capability leaks by applying data-flow analysis on pre-loaded applications in eight popular Android smartphone images. Woodpecker generates a CFG that is based on Dalvik bytecode and identifies possible execution paths that involve the use of privileged permissions.

In summary, these approaches leverage various analysis techniques to enhance the security of the Android platform and to detect capabilities leaks that may be
misused by malicious applications. However, these approaches do not discuss the context of the use of the Android permissions requested by an application, e.g., the cause of the use. The context information is useful because it can help users to understand the risk of granting a permission to an application and to help users make informed decisions. Besides, given the large number of Android applications, it is very important to be able to automatically interpret the use of permissions and to infer the potential security and privacy risks about granting the permissions.

2.3.3 Application Security

This category includes approaches to study the security of Android applications. For example, Felt et al. [42] propose inter-process communication (IPC) inspection to monitor messages used for IPC and to reduce the privileges of a recipient to the intersection of recipient’s and the requester’s permissions. Dietz et al. [43] proposed QUIRE to track the call-chain of IPC in order to defend against the confused deputy attack and to provide a mutual verification scheme for applications. Chin et al. [44] proposed ComDroid to detect the vulnerabilities in the inter-application communication. Bugiel et al. [45] proposed XmanDroid to prevent privilege escalation. XmanDroid monitors the communication between applications and applies policies to restrict the interaction. Gilbert et al. [46] proposed AppInspector, which leverages information-flow tracking on sensitive information to automatically identify private information leakage in an application. AppInspector faced challenges such as analyzing the logs collected from runtime and traversing all code paths.

In summary, these approaches solve specific threats such as confuse of deputy, privilege escalation, sensitive information leakage that are posed by Android applications. However, these approaches are not proposed to identify all the potential security and privacy risks with requested permissions of an application.

2.3.4 Malicious Application Characterization and Detection

The fourth category of related works aims at detecting malicious Android applications. Zhou et al. [47] systematically characterize the existing Android malwares into various malware families. They show that many of the malicious application
families sending out SMSs to premium numbers or harvesting user information. Besides, they also discovered that many malicious applications use the entry activity (e.g., “ACTION_MAIN”) to trigger the execution. In [48], Zhou et al. studied the infection of known malicious applications in various Android marketplaces by characterizing the behaviors of these malware families. They filtered out applications that do not request malware required permissions. They characterized known malwares from manifest files, API invocations and structural layouts. To detect unknown malwares, they proposed several heuristics and examined the run-time behaviors (e.g., looking for pre-defined malicious behavior) of the applications. Grace et al. [49] proposed a two-order detection scheme to identify known dangerous behaviors (e.g. sending background SMS messages) from three different categories of risks. To detect high-risk applications, they wrote signatures on known smartphone platform vulnerabilities (i.e., CVE). To detect medium-risk applications, they identify known behaviors that are associated with users being charged money surreptitiously or private information being uploaded to a remote server. For second order detection, they proposed to identify two specific behaviors: unsafe Dalvik code loading and encrypted native code execution.

In summary, these approaches are developed to identify similarities among similar malwares. These approaches all rely on the knowledge of known behaviors of malicious applications. Therefore, these approaches can not detect new and unknown malicious behaviors.
Chapter 3

Detection of Obfuscated Malicious JavaScript Code

As one of the top Internet security threats, malicious JavaScript code has been observed mostly in Web pages. Before a Web page is rendered by Web browsers, the page is normally examined by anti-virus software as a common defense mechanism. However, since most anti-virus software apply signature matching-based detection scheme, it is very easy to evade the detection by applying obfuscation techniques on the malicious JavaScript code. One example of the obfuscation techniques is string manipulation such as ASCII encoding, where the malicious JavaScript code is encoded into escaped ASCII characters. However, obfuscation techniques have also been applied to benign JavaScript code as well. For example, benign JavaScript code applies obfuscation to protect the intellectual property of the code. Therefore, the existence of obfuscation techniques does not suggest maliciousness.

Various obfuscation techniques have been observed from malicious JavaScript code in the real world. To better understand the obfuscation techniques adopted by malicious JavaScript code, we conduct a measurement study. In this study, we first categorize observed JavaScript obfuscation techniques. Then we perform a statistic analysis on the usage of different categories of obfuscation techniques in real world malicious JavaScript samples. We also study the detection effectiveness of twenty most popular anti-virus software products against obfuscation techniques. Based on the results, we analyze the causes of the popularity of obfuscation in malicious JavaScript code; the reasons behind the choice of obfuscation techniques and the
difference between benign obfuscation and malicious obfuscation.

We further discuss how to detect and prevent obfuscated malicious JavaScript code. More specifically, we propose a mostly static approach called JStill. JStill captures some essential characteristics of obfuscated malicious code by using function invocation-based analysis. It also leverages the combination of static analysis and lightweight runtime inspection so that it can not only detect, but also prevent the execution of the obfuscated malicious JavaScript code in Web browsers. We evaluate JStill using real-world malicious JavaScript samples as well as Alexa top 50,000 websites. The results show that JStill can achieve high detection accuracy (detect all malicious samples in our experiment) and low false positives. Meanwhile, JStill only incurs negligible performance overhead, making it a practical solution to preventing obfuscated malicious JavaScript code.

3.1 The Power of Obfuscation Techniques in Malicious JavaScript Code

To defend against malicious JavaScript code, most Internet users rely on the anti-virus software. Unfortunately, the effectiveness of static signature-based anti-virus software is often thwarted by obfuscation techniques. In fact, malicious JavaScript code has been increasingly applying obfuscation techniques to evade the detection of anti-virus software and to hide its malicious intent. Meanwhile, we also acknowledge that obfuscation techniques are not exclusive to malicious JavaScript code. For example, we find some benign Web pages (e.g., the frontpage of yahoo.com) also use obfuscation to prevent code plagiarism.

In this study, we discuss a categorization of obfuscation techniques in malicious JavaScript code, the statistics of the usage of obfuscation techniques and the difference between obfuscation techniques adopted by benign JavaScript code and by malicious JavaScript code.
3.1.1 Categorization of the Observed Obfuscation Techniques in Malicious JavaScript Code

We first classify the observed obfuscation techniques into the following four categories based on the operations performed by them.

**Randomization Obfuscation** Attackers may randomly insert or change some elements of JavaScript codes without changing the semantics of the codes. Common techniques include: *whitespace randomization* [50], *comments randomization* and *variable and function names randomization*.

Whitespace randomization is to randomly insert whitespace characters, including space character, tab, line feed, form feed and carriage return, in JavaScript code. Comments randomization randomly inserts arbitrarily created comments into JavaScript codes. These two take the advantage of the fact that JavaScript interpreters ignore whitespace characters and comments. Variable and function name randomization replaces variable names or function names by randomly created strings with non-obvious meanings. Usually two or more randomization methods are used together to improve the possibility of evading detections.

Figure 3.1 gives a demonstration of these two JavaScript obfuscation techniques. Figure 3.1(a) is the original code and Figure 3.1(b) is the obfuscated one. Strings highlighted by red rectangles are randomized variable names and function names. Strings starting with “//” are comments created randomly. Whitespace randomization generates line feed in the end of the first line between “function”

```
function myfunction(txt)
{
    alert(txt);
}
var mystring = "Hello World!";
myfunction(mystring);
```

(a)

```
function i23dfcnj(_fdji230fdj)
//_32akfaj0ufa
{// _dafaljlfamfdn
alert(_fdji230fdj); //_dkfahajkla13
}
var dfiazza192//_gcvdseapk
    = "Hello World!"; // gpokjk3424pkl
i23dfcnji23dfcnj,dfiazza192);
```

(b)
Data Obfuscation

Data obfuscation is to convert a variable or a constant into the computational results of one or several variables or constants. Two data obfuscation techniques have been wildly applied to string object. One is string splitting. The other one is keyword substitution. String splitting is to convert a string into the concatenation of several substrings. String splitting is usually used along with document.write() or eval() functions to execute the concatenated strings in a browser. Attackers could change the order of substrings and assign random variable names to them to make the code even harder to understand. Another obfuscation approach is to use a variable to substitute JavaScript keywords. Examples are shown in Figure 3.2. Figure 3.2(a) is the original code\(^1\). Figure 3.2(b) uses string splitting obfuscation, where substrings are in a random order to make the code harder to understand and detect. Figure 3.2(c) uses keyword substitution, where keyword “document” is represented by variable “mystring”.

Besides strings, numbers are another object of data obfuscation. For example, \(i = 10\) can be rewritten in many ways: \(i = 5 \times 2\), \(i = 11 - 1\) or \(i = 1000/100\), etc.

---

\(^1\)If not specified, we will use this code as the original code in all the following examples in this section.
Encoding Obfuscation

Normally, there are 3 ways to encode original code. The first way is to convert the code into escaped ASCII characters, unicode or hexadecimal representations. The second method uses customized encoding functions, where attackers usually use an encoding function to create the obfuscated code and attach a decoding function to decode it during execution. Figure 3.3 shows an example using hexadecimal representation to implement encoding. In Figure 3.4, we give a simple example of how to obfuscate JavaScript codes by customized encoding and decoding functions: Figure 3.4(a) is the encoding function, which increases the Unicode of each character by 1, where “document.write(‘Hello world!’)” is encoded into “epdvnfou/xsjuf)(Ifmmp!xpsme(*”; Figure 3.4(b) is the obfuscated code, which first applies a decoding function, and then executes the decoded instructions. In addition, some standard encryption and decryption methods can be employed to do JavaScript obfuscation. For example, JScript.Encode is a method created by Microsoft to encode JavaScript code. It can be used to protect source code as well as to evade detection.

```
 eval("\x64\x6f\x63\x75\x6d\x65\x6e\x74\x2e\x77\x72\x69\x74\nx65\x28\x27\x48\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x21\nx27\x29");
```

Figure 3.3. An example of obfuscation using hexadecimal representation.

Logic Structure Obfuscation

This type of obfuscation technique is to manipulate the execution paths of JavaScript codes by changing the logic structure, without affecting the original semantics. There are two ways to implement logic structure obfuscation. One way is to insert some instructions which are independent of the functionality. The other one is to add or change some conditional branches, such as `if ... else, switch ... case, for, while` etc. Examples are shown in Figure 3.5, where Figure 3.5(a) inserts independent instructions; Figure 3.5(b) uses additional conditional branches.
function encode(mystring)
{
    var c="";
    var i = 0;
    for(var i=0;i <mystring.length;i++){
        c = c + String.fromCharCode(mystring.charCodeAt(i)+1);
    }
    return c;
}

var c = encode("document.write('Hello world!');");
document.write(c);

var c = encode("document.write('Hello world!')");
document.write(c);

function decode(c)
{
    var mystring="";
    var i = 0;
    for(var i=0;i <c.length;i++){
        mystring = mystring + String.fromCharCode(c.charCodeAt(i)-1);
    }
    return mystring;
}

var c = "epdvnfou/xsjuf)(Ifmmp!xpsme(*";
var mystring = decode(c);
eval(decode(c));

var i = 111;
var i =0;
i = i+1;
for(i=0;i<10000;i++)
{
    if(i<10)
    {
        alart("Warning");
    }
    document.write('Hello world');
}

var i = 111;
i = i+1;
if(i<10)
{
    alart("Warning");
}
document.write('Hello world');
i = i+1;

Figure 3.4. An example of obfuscation using encoding/decoding functions. (a) is the encoding function; (b) is the obfuscated code.

Figure 3.5. Examples of logic structure obfuscation. (a)inserts independent instructions; (b)uses additional conditional branches.

3.1.2 The Usage of Obfuscation Techniques in Malicious JavaScript Code

Given the previous categorization, we now discuss the usage of obfuscation in malicious JavaScript code.
**Sample Collection and Screening** The malicious JavaScript samples used in our study are collected from VirusTotal [51], which provides a free interface to scan uploaded files using more than 40 state-of-the-art anti-virus software. We selected 1039 malicious HTML samples that have been detected by more than 5 anti-virus software. Since there is no way to only select malicious JavaScript samples, we have to filter out the malicious HTML samples that do not contain any JavaScript code. There are 248 (23.9%) such samples, among which 92 do not include any script code and 156 use other scripting languages (e.g., VBScript). The remaining 791 (76.1%) samples contain JavaScript codes. Although all 791 samples contain malicious JavaScript code, not all of these samples are detected as malicious because of malicious JavaScript code. For example, an HTML web page that contains only benign JavaScript code can still be malicious because of an embedded malicious URL. In order to obtain the most representative malicious JavaScript samples, we further filter the sample set by two criteria: 1) only keep one sample among samples that are in the same malicious JavaScript family (determined by name); 2) only keeping samples that are reported by more than 15 (out of 20) anti-virus software. We choose the most popular 20 anti-virus software in the market (the selection process will be discussed in Section 2.1.3) to improve our confidence in the verdict of a sample. Eventually, there are 510 samples left in our sample set.

We find 71% samples use various JavaScript obfuscation techniques. This indicates that obfuscation is a common practice among malicious JavaScript codes to evade detections and to pose an obstacle to code analysis. Among these 71%, 30% of them use at least two types of obfuscation techniques to better hide their malicious purposes. Figure 3.6 shows the distribution on the number of obfuscation technique categories employed in the samples.

We also compare the popularity of various obfuscation techniques. The result in Table 3.1 shows that string computation, e.g., string splitting and keyword substitution, is the most popular method. ASCII/Unicode/Hex code and customized encoding functions are also very common.
Figure 3.6. The Distribution on the Number of Obfuscation Technique Categories employed in samples

Table 3.1. The usage of JavaScript Obfuscation Techniques

<table>
<thead>
<tr>
<th>Obfuscation Category</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Randomization Obfuscation</td>
<td></td>
</tr>
<tr>
<td>Whitespace Randomization</td>
<td>3</td>
</tr>
<tr>
<td>Variable and Function Names Randomization</td>
<td>11</td>
</tr>
<tr>
<td>Comments Randomization</td>
<td>2</td>
</tr>
<tr>
<td>Data Obfuscation</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>45</td>
</tr>
<tr>
<td>Number</td>
<td>2</td>
</tr>
<tr>
<td>Encoding Obfuscation</td>
<td></td>
</tr>
<tr>
<td>ASCII/Unicode/Hex Coding</td>
<td>32</td>
</tr>
<tr>
<td>Customized Encoding Functions</td>
<td>23</td>
</tr>
<tr>
<td>Standard Encryption and Decryption</td>
<td>3</td>
</tr>
<tr>
<td>Logic Obfuscation</td>
<td></td>
</tr>
<tr>
<td>Insert Irrelevant Instructions</td>
<td>8</td>
</tr>
<tr>
<td>Additional Conditional Branches</td>
<td>3</td>
</tr>
</tbody>
</table>

3.1.3 The Effectiveness of Evading Anti-virus Software

We apply different previously categorized types of obfuscation techniques to obfuscate the malicious JavaScript samples in our sample set. After that, we use 20 most popular anti-virus software to scan these obfuscated codes to test the effectiveness of obfuscation techniques in evading anti-virus software. Although 71% of the sample set already adopt various obfuscation techniques, the fact that these samples can be detected by anti-virus software leads to the conclusion that anti-virus software must have generated signatures on the obfuscated malicious code.
Table 3.2. The Selection of Anti-virus Software

<table>
<thead>
<tr>
<th></th>
<th>Name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a-squared (Emsi Software GmbH)</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>AntiVir (Avira)</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>Avast! Antivirus (ALWIL)</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>AVG (AVG Technologies)</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>AVP (Kaspersky Lab)</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>BitDefender (BitDefender GmbH)</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>ClamAV (ClamAV)</td>
<td>17</td>
</tr>
<tr>
<td>8</td>
<td>Comodo (Comodo)</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>DrWeb Doctor (Web, Ltd.)</td>
<td>19</td>
</tr>
<tr>
<td>10</td>
<td>ESET NOD32 (Eset Software)</td>
<td>20</td>
</tr>
</tbody>
</table>

Therefore, whether these samples contain obfuscated malicious JavaScript code or not, in this test, we will apply another layer of obfuscation to these samples so that we can test the applied obfuscation technique's actual effectiveness of evading anti-virus software.

Since we could not find a widely accepted ranking of anti-virus tools, we choose 20 anti-virus software with the highest rankings on Alexa [52] (in the “Computers/Security” category). We verify our selection by comparing it to some other anti-virus ranking Web sites, e.g., [53] [54] [55] [56]. We noticed that in the top 10 anti-virus software of each ranking list, at least 8 are included in our selection. Therefore, we believe our selection is representative. Table 3.2 shows the names of these 20 selected anti-virus software.

Before we use the 20 selected anti-virus software to scan further-obfuscated samples, we list the detection rates of these 20 anti-virus software on the samples in our sample set. As shown in Figure 3.7, the average detection rate is 86.85%.

Detection against Randomization Obfuscation In this section, we obfuscate samples in our data set by randomly adding whitespace and comments, and changing variable and function names to randomly created strings. Figure 3.8 shows the detection rate of these anti-virus software on the obfuscated samples. The average detection rate is 55.3%, which is much lower than the previous one. Only No.2 and No.20 software demonstrate strong resistance to randomization obfuscation, while all others could be relatively easily bypassed. Because whitespace and comments do not change any feature of the codes, we believe that the other 18 anti-virus software rely on static signatures.

Detection against Data Obfuscation In this section, we obfuscate samples
in our data set by using string computation, keyword substitution, and/or number computation. Figure 3.9 shows the detection rate on obfuscated codes. The average detection rate is 45.7%. The results shows that data obfuscation is more effective to evade detection than randomization.

In one sample, we split a long string into many substrings, and then concatenate them together. All we change is adding "+" between two adjacent substrings. 16 anti-virus software cannot detect the obfuscated sample as malicious. It indicates that, with a high probability, those anti-virus software use exact matching to compare with the signatures. We also saw another extreme example: there is such an instruction in the sample:


We split the second parameter of cxw.value.replace() into 5 substrings, use variables to represent these substrings, and then concatenate them together. Obviously, the semantic of the sample does not change. However, none of the 20 anti-virus software can detect it as malicious. It is also a strong evidence of the
fact that these anti-virus software highly relay on signatures. For this sample, the signature is located in the string which we split.

Detection Rate against Encoding Obfuscation We obfuscate the samples using encoding obfuscation, e.g. Hexadecimal encoding. The encoded code is first decoded and then executed by invoke “eval()”. The results show that none of the 20 anti-virus software can detect these encoding obfuscated samples.

Comparison of Obfuscation Techniques Based on our experiment results, we believe that even simple obfuscation methods could effectively evade the detection of anti-virus software. We also believe that most popular anti-virus software use signature-based detection scheme. Some of them even use exact matching.

3.1.4 Analysis

The Cause of the Popularity of Obfuscation The popularity of obfuscation among malicious JavaScript code is caused by the following reasons. First, signature-based detection systems, e.g., anti-virus software, can be effectively evaded by obfuscation. As we have demonstrated, by applying encoding obfuscation, all the obfuscated samples can successfully evade the detection of any state-of-the-art anti-virus software in our study.
Second and more importantly, we discover that the dynamic features of JavaScript language such as dynamic generation (D-Gen) and run-time evaluation (R-Eval) functions often facilitate the creation of obfuscation routines. Since these two features provide a means of transforming text to code in JavaScript, at a high level, any string manipulation process can be combined with dynamic generation/run-time evaluation function (e.g., “document.write()”, “document.writeln()”, “eval()”, “window.setTimeout()”, “window.setInterval()”) to generate an obfuscation routine (e.g., [57]). Therefore, these two features are widely exploited in malicious JavaScript obfuscation. On the other hand, D-Gen and R-Eval functions are also commonly used features in benign JavaScript code [58]. In the case of conditional loading, an external JavaScript code is loaded using dynamic generation only when certain condition is met. This can avoid unnecessary bandwidth consumption. Another example is including runtime generated information in JavaScript code to increase the flexibility in programming. When JavaScript code contains information that can only be obtained during runtime from either user input or client-server interaction, it will leverage runtime evaluation. Therefore, the adoption of D-Gen and R-Eval functions does not always imply the existence of obfuscated malicious code.
Third, many of the obfuscation techniques can be stacked together to generate a multi-level obfuscation scheme. We noticed that this feature makes the reuse of obfuscated malicious code more easily since attackers do not need to deobfuscate an obfuscated code before applying another obfuscation technique. Therefore, when an obfuscated malicious code is detected by anti-virus software, attackers can simply wrap the obfuscated code with another layer of obfuscation to further evade detection.

Fourth, the improvement in the performance of JavaScript engine in current web browsers further helps to promote the popularity of obfuscation as well. We noticed that the execution time difference between un-obfuscated code and obfuscated code is almost negligible. Therefore, attackers no longer have to concern that the time difference will cause user’s suspect of any malicious activity. Moreover, with the improvement of performance, we also observe that nearly half of the obfuscated samples actually apply multiple levels of obfuscation to better hide their malicious intent.

The Choice of Obfuscation Technique by Malicious JavaScript Code
As the results illustrated, the top two most popular obfuscation technique is data obfuscation (e.g., string splitting) and encoding obfuscation (e.g., ASCII/Unicode/Hex encoding). This is because: 1) both techniques can be effortlessly applied to any JavaScript code without considering the logic of the code; 2) these two techniques can reduce the detection rate by around 40% and 100% respectively.

However, data obfuscation can not evade the detection of anti-virus software as effectively as encoding obfuscation. Therefore, we believe an effective obfuscation detection approach should focus on encoding obfuscation.

Benign Obfuscation vs Malicious Obfuscation
Both benign and malicious JavaScript code have been observed adopting obfuscation techniques; hence, obfuscation does not imply maliciousness. However, their purposes of obfuscation are different. Benign JavaScript code mainly leverages obfuscation to protect code privacy or intellectual property. This purpose requires obfuscated code to be human unreadable. Malicious JavaScript code exploits obfuscation to hide its malicious intent; therefore, the obfuscated code aims to evade automatic static inspection.
3.2 Overview of JStill

This section gives an overview on the design of JStill.

3.2.1 Function Invocation-Based Analysis

the basic observation in the design of JStill is that either the deobfuscation or the execution of obfuscated malicious JavaScript code has to involve function invocations. To leverage this observation, we first categorize functions in JavaScript into the following types: 1) JavaScript native functions (e.g., `eval`), 2) JavaScript built-in functions (e.g., `unescape, string.fromCharCode`), 3) DOM methods (e.g., `document.write, window.setTimeout`) and 4) user-defined functions. Since both obfuscated malicious JavaScript code and benign JavaScript code invoke these four types of functions, the challenge here is how to distinguish function invocations in obfuscated malicious code from that in benign code.

To this end, JStill captures the essential difference between obfuscated malicious invocations and benign invocations from the following three aspects.

**Function arguments.** We notice that for some language-defined functions that are often used in deobfuscation, e.g., the D-Gen and R-Eval functions, malicious invocations of these functions often hide their arguments from the static perspective, e.g., using the output of another function as arguments. This is necessary for obfuscated malicious code because the arguments of these function
invocations often contain part or all of the malicious code. Exposing these arguments will increase the chance of being detected by static inspections. For other language-defined functions used in deobfuscation, e.g., `unescape`, we noticed that the outputs of these functions are often used as or in the arguments of D-Gen and R-Eval functions. This is because in obfuscated malicious JavaScript code, functions like `unescape` can decode the obfuscated string, which will later be generated or evaluated as code.

**Function definition.** In benign code, a user-defined function is normally first defined before it is invoked. However, in many cases of obfuscated malicious code, a malicious function’s definition is either entirely or partially obfuscated in order to hide the semantics of the malicious code. Therefore, when the malicious function is invoked later, it would appear undefined from the static point of view, even though its definition has already been evaluated by a JavaScript engine. In addition to obfuscated malicious code, a coding bug can also cause invocations of undefined functions. The only difference here is that in a coding bug, the function is indeed undefined before its invocation.

Note that hiding definitions of user-defined functions is very rare in benign JavaScript code. As we mentioned before, the purpose of obfuscation for benign JavaScript code is mainly for intellectual property protection. Since JavaScript is delivered in source form, not as compiled machine code, its source code cannot be protected as much as in other languages. An instrumented browser can reveal the source JavaScript code to people who are interested, no matter what obfuscation is applied on that code. Therefore, rather than hiding the source code, most benign obfuscation focuses on reducing the interpretability of the source code to make it hard for human to understand the logic of the code. To this end, benign code favors randomization and substitution-based obfuscation techniques. Another reason that benign code normally does not adopt dynamic generation-based obfuscation is the concern with extra performance overhead, which is an important factor to evaluate the coding quality of a website.

**The context of a function invocation.** A context means what function is actually invoked. Based on our analysis of obfuscated malicious JavaScript code, we observed the disguise in invoking language-defined functions, in order to evade detection that leverages invocation patterns of language-defined functions.
Figure 3.11 lists two common disguise techniques. In part (a) of Figure 3.11, `eval()` function is assigned to an object `a`, which is later invoked as `eval()`. In part (b), the properties of a window object are traversed to look for the “document” keyword, which is assigned to object `b`, whereas object `c` actually contains the string “write” after the execution of “unescape()”, so together the last statement actually invokes “document.write()”. In these examples, static analysis may be able to trace back to the language-defined functions that are actually invoked, but it is neither reliable nor efficient. If the statement “a = eval;” in part (a) is escaped and evaluated in runtime using function “eval(unescape("%61%3D%65%76%61%6C"))”, static analysis will not detect that “eval” has been assigned to “a” unless it has access to the runtime generated code.

3.2.2 System Overview

To detect obfuscated malicious JavaScript code, JStill uses static analysis to examine function invocations from the above three aspects: function definition, content of arguments and context of invocation. As illustrated in Figure 3.10, the static analysis first parses JavaScript code and based on the parsing results it logs information such as strings, function definitions and function invocations. JStill leverages the static information about function definitions and invocations by comparing it with the runtime information about function definitions and invocations. This comparison can reveal what functions are statically undefined as well as what function definitions are hidden by obfuscation. The information about string is used by JStill in the analysis of hidden arguments, which will be discussed later.

Meanwhile, since static analysis itself is insufficient to discern coding bugs from
obfuscated malicious code, or to identify disguised function invocations, JStill also leverages its runtime component to assist static analysis. In runtime, JStill hooks the invocations of selected language-defined functions in a browser. In this way, it can examine the suspicious arguments just before the execution since the arguments are in clear-format at this stage. JStill can also spot disguised invocations of these hooked functions. Because no matter what disguise is applied, the invocation will always be handled by the hooked functions such that JStill has an opportunity to check if the invoked function in the code is actually the hooked function.

Since JStill detects obfuscated malicious JavaScript code from three aspects, it consists of three detection criterions: 1) disguised invocations of language-defined functions, 2) obfuscated function definitions, 3) obfuscated malicious arguments. JStill raises an alarm if at least one criterion is met. Note that the design of JStill does not rely on any unique specification in a browser’s implementation. Therefore, JStill can be implemented compatibly in any Web browser.

3.3 Design of JStill

In this section, we explicate the design of JStill, particularly the technical challenges and their solutions in enforcing the three detection criterions.

3.3.1 Identification of Disguised Function Invocations

To identify a disguised function invocation, two pieces of information are necessary: 1) what function is actually invoked in an invocation; 2) what function appears to be invoked in an invocation.

To gather the information about what function is actually invoked, static approaches, such as tracing back to the actual function being invoked, are either unreliable or infeasible. Therefore, JStill leverages runtime inspection to identify the function that is actually invoked. More specifically, JStill hooks the implementation of language-defined functions that are mostly likely to be disguised in obfuscated malicious JavaScript code. These functions include but not limited to D-Gen and R-Eval functions (e.g., the functions disguised in Figure 3.11, “eval” and “document.write”), and functions that are commonly used in string manip-
ulations (e.g., "unescape", etc.). These functions are mostly likely to be disguised because their prevalence in obfuscated malicious code makes them (part of) the widely used detection signatures in static inspections. When one of these hooked functions is invoked by a function invocation, JStill can identify the hooked function as the actually invoked function.

However, hooking a function’s implementation cannot reveal what function appears to be invoked. Many of these functions (e.g., DOM-based functions) are not implemented within the JavaScript engine; the invocations of these function are actually wrapped by a component (e.g., XPConnect in Firefox) that allows JavaScript code to invoke these functions without revealing the function name in the invocation (e.g., “a” in Figure 3.11(a)).

To address this issue and obtain the information about what function appears to be invoked, JStill marks all the statically identified invocations of a hooked function (not only from source code, but also from dynamically generated code). As a result, when a hooked function is invoked during runtime, JStill can check if this invocation has been marked; any unmarked invocation in this case indicates this invocation is disguised from the static perspective.

```javascript
// Original function definition
function addition(x,y) {return x+y;}

// function passed as variable
func3 = obj.func2;
func3(2,3);

// function passed as array
arr = new Array(addition, 2, 3);
arr[0](arr[1], arr[2]);

// Non-traceable function
if(userinput){
    func4 = addition;
}
else
    func4 = undefined;
result = func4(2,3);
```

Figure 3.12. Example of legitimate function invocations in JavaScript
The marking scheme must cover all the statically identifiable invocations of hooked functions to eliminate false positive in disguised invocation detection, but identification of function invocations in JavaScript is not a trivial task. Since function in JavaScript is merely a special type of object, it can be assigned as variables, stored as properties in other objects or as elements in arrays. In other words, a function definition can be passed as a value from one object to another object. For example, Figure 3.12 lists different means by which “function addition()” is passed to various objects (or properties) and gets invoked. In all four cases, the last statement actually invokes “addition(2, 3);”. In invocation 1, function “addition” is passed to an array element, e.g., “arr[0]”. In invocation 2, it is passed to an object’s (including other function object’s) property, e.g., “obj.func2”. After that, it is passed to a variable and gets invoked by a variable’s name, e.g., “func3” in both invocation 3 and 4.

Figure 3.13. Identify function invocations via bytecode
To identify all the function invocations despite the flexibility in the syntax of JavaScript, necessary syntactic information needs to be parsed from the source code. To this end, JStill leverages the bytecode that is compiled from source code, an example of which is shown in Figure 3.13. From bytecode in Figure 3.13, it is clear that there exist three function invocations (the three bytecode “call” marked by different colors). To leverage the bytecode to identify function invocations, JStill needs to understand the structured syntactic information offered by bytecode. The information is organized as a set of 3-tuples. As illustrated in Figure 3.13, each line of bytecode is a 3-tuple that represents a sequence number, a line number and a bytecode instruction, respectively.

When marking a function invocation for runtime inspection, JStill needs to make sure that the mark cannot be bypassed by malicious code. Meanwhile, it also tries to avoid modification on the source code to prevent unwanted impact on the runtime behavior of the code. As a result, JStill actually marks invocations by logging the locations of these invocation instructions. In this way, when a hooked function is actually invoked, JStill can determine whether the invocation is disguised by checking if the location of instruction for this invocation is marked. An unmarked bytecode will indicate a disguised invocation of a hooked function.

The same approach however cannot be applied on user-defined functions because these functions are not implemented in the browser and hence cannot be hooked. To identify disguised invocations of user-defined functions, JStill leverages the object hierarchy in JavaScript. Every user-defined function is a property of its parent object. When a user-defined function is invoked, JStill can identify what property this invocation actually uses. If the property’s name and the caller in the invocation do not match, it means the invocation is disguised.

Note that bytecode, as an intermediate interpretation of JavaScript source code, has different forms in various browsers (e.g., Firefox, Safari and IE). However, it is not indispensable to a browser’s implementation (e.g., Chrome’s JavaScript engine V8 escapes bytecode) or to JStill. The reason to use bytecode generated by a JavaScript engine rather than parsing the source code by JStill itself is that commodity JavaScript engines such as SpiderMonkey are highly optimized and robust facing malformed JavaScript code.
3.3.2 Detection of Obfuscated Function Definitions

As we discussed in section 2.2.2, the second aspect in JStill’s analysis of invocation is function definition. Since an obfuscated definition of a user-defined function is an indication of obfuscated malicious JavaScript code, we will discuss how to detect obfuscated function definition in this section.

A function definition can be obfuscated by either hiding the entire function definition or a part of the function body. When the entire function definition is hidden, the definition cannot be observed in the process of parsing the source code. Hence, an invocation of this function would appear statically undefined. However, a coding bug may appear the same way. Therefore, to provide a more accurate detection of obfuscated function definition, JStill checks every invocation to see whether the invoked function is actually defined or not. If the actually invoked function is defined only in runtime and the definition is hidden from static perspectives, it is an obfuscated function definition.

To examine whether a function has been defined or not, JStill checks all the function definitions it logs in parsing JavaScript code (both source code and dynamically generated JavaScript code). In this process, JStill uses both function names and the object hierarchy to match a function definition, since function name-based definition match can cause inaccuracy due to different JavaScript contexts. For example, functions with the same name can be defined within different objects. Therefore, to accurately match a function definition, information about where the function is defined also needs to be checked. Such information can be obtained by checking the object hierarchy of a function definition. Therefore, object hierarchy is also logged together with function definition during the course of parsing.

To identify obfuscated function definitions, JStill checks every function definition identified in runtime-generated code. If a function definition is generated from obfuscated arguments (the detection of which will be discussed in the next Section) of D-Gen and R-Eval functions, it is an obfuscated definition.

A function definition can also be partially obfuscated, i.e., only part of the function body is hidden. In this case, the function must contain code that is dynamically generated using obfuscated arguments. Therefore, partially obfuscated function definition can be identified via the detection of obfuscated arguments within a function body. Specifically, JStill marks the invocations of D-Gen and
R-Eval functions within a function body in parsing. When the marked invocations are detected as containing obfuscated arguments in JStill, partial obfuscation in the function body can also be detected.

Another practical issue is that there exist cases in which the function definition of an invocation cannot be determined statically. For example, in invocation 4 of Figure 3.12, the value of the variable “userinput” cannot be determined statically, so the actual definition of “func4” remains unknown from the static viewpoint. To solve this issue, JStill leverages the result from identification of disguised invocations such that it can determine which function definition is actually invoked before checking definition obfuscation.

### 3.3.3 Detection of Obfuscated Malicious Arguments

D-Gen and R-Eval functions used in obfuscated malicious code often obfuscate their arguments. Since these functions can transform text to JavaScript code, their arguments are hidden in the form of the outputs of string manipulation functions. These functions can be either language-defined or user-defined. In the example of Figure 3.13, the argument of `document.write` is the output of `unescape`. Moreover, the trace from the output of a string manipulation function to the arguments of D-Gen and R-Eval functions can be obfuscated as well. For the code in Figure 3.13, an attacker can change the first statement to “`document.write(s);`”, where “s” is a string that actually equals to “`unescape("%66...%7d")`” except this equivalence is disguised by other statements crafted by attackers.

Hooking D-Gen and R-Eval functions provides an opportunity to examine the content of arguments, but the content itself does not shed any light on whether it has been obfuscated. Besides, patterns that show a resemblance to the obfuscated malicious code in Figure 3.13 have been observed in benign JavaScript code as well, e.g., “`document.write('<script ' + 'src="' + urlStart + '.2mdn.net/' + iframeScriptFile + '"></script>');`”. In view of this, detection of obfuscated arguments in D-Gen and R-Eval functions is a challenging problem.

To solve the problem, JStill introduces a metric named obfuscated malicious argument (OMA) metric for all the arguments of dynamic generation and runtime evaluation functions. This metric indicates the possibility that an argument is
used in obfuscated malicious code. The main purpose of applying obfuscation on malicious arguments is to hide the content of the malicious arguments; hence, the malicious arguments, or most of the arguments must not be observed from the source code. In benign code, the arguments do not need to be hidden, but often need to be dynamically assembled (or concatenated), since some parts of the benign arguments depend on user input or environment variables. In other words, in benign JavaScript code, most (if not all) of the content of the arguments can be found in the Web page (including URLs). Based on this observation, JStill defines the OMA metric as the percentage of an argument that can be found in the Web page. For arguments with a low value on this metric, it is highly likely that they are obfuscated malicious code.

Since only the arguments that generate JavaScript code are potentially involved with obfuscated malicious code, other arguments can be excluded from examination to improve performance. For example, JStill rules out the invocations of D-Gen and R-Eval functions which create new HTML elements that are neither a script tag nor containing any event handler, because these arguments will not introduce new JavaScript code. Besides, arguments that create script tags used for dynamic inclusions (e.g., `<script src="a.js">`) are also excluded from this examination, because the dynamically included code will be analyzed by JStill later.

To calculate the metric of an argument, e.g., a string, JStill logs all the string variables from the parsed source code and the values of some environment variables that are commonly used in benign JavaScript code, such as window.location.href, navigator.userAgent, element IDs, etc. The details on the calculation of the metric will be explained in Section 2.3.4.

JavaScript provides many functions that can be used for D-Gen and R-Eval. Appendix A.3 lists the functions that JStill hooks in Firefox. We realize that this list is browser specific (e.g., Firefox in this work), but the design of JStill is not exclusive to a certain browser.
3.3.4 Obfuscated Malicious Argument Metric

OMA metric measures the possibility of an argument being used in obfuscated malicious JavaScript code. Given a set of strings and the content of an argument, the metric is calculated as the largest percentage of the argument’s content that can be found as the combination of the strings or substrings from the set.

\[
\text{Start of a string}
\]

\[
\text{a} \quad \text{z} \quad \text{A} \quad \text{Z} \quad 0 \quad \text{g}
\]

\[
\text{... a} \quad \text{z} \quad \text{A} \quad \text{Z} \quad 0 \quad \text{g} \quad ...
\]

\[
\text{...}
\]

Figure 3.14. A Search Tree for Basic Strings

In benign JavaScript code, not only the strings, but also their substrings may be used in the composition of benign arguments. Therefore, the logged strings in the set are first divided into basic strings, which are substrings consisting of consecutive letters and numbers with a minimal length of 2 characters. This is because benign code mostly uses substrings that are divided by symbol separators, such as query content divided by questions mark in a URL, or cookie id after an equal sign.

Given a set of basic strings (let the size of the set be \( p \), the average length of the strings in the set be \( m \)), the first calculation step is to find which basic strings are substrings of the argument (with size \( n \)). A brute-force algorithm has a complexity of \( O(pmn) \). This will cause a significant performance penalty when \( p \) and \( n \) are large numbers. In fact, \( p \) is usually large due to the large number of strings defined in JavaScript code.

To avoid the performance penalty, we propose a search tree-based algorithm. As illustrated in Figure 3.14, each internal node can have at most 62 children, which are mapped to the characters set \((a-z, A-Z, 0-9)\). This tree has \( p \) leaf nodes and an average depth of \( m \). Based on this tree, JStill can accomplish the first step using the Substring Identification Algorithm (Appendix A.2)
Given the set of matched basic strings, the next step is to find a subset so that the combination of strings in this subset matches the largest percentage of the argument. This problem reduces to the knapsack problem, which is NP-Complete, so we propose an approximate, the Metric Calculation Algorithm (Appendix A.2) that leverages the following greedy heuristic: always using the longest matched and non-overlapping substring. In practice, this algorithm works very well, because the overlapped substrings are not very common.

The time complexity of the Substring Identification Algorithm (Appendix A.2) is $O(mn)$, and the time complexity of the Metric Calculation Algorithm (Appendix A.2) is $O(qn)$, where $q$ is the number of matched substring set output by the Substring Identification Algorithm.

### 3.3.5 Whitelisting

As the most popular client-side language in Web development, JavaScript has many libraries and widgets. Most of these libraries are frequently used in many websites and are known to be benign, e.g., JQuery. Therefore, it is only a waste of resource to examine these known benign libraries in JStill.

To save the resource and to improve performance, we propose a whitelist mechanism in JStill. For a known JavaScript library that is often included as an external file, its hash value is computed and stored in JStill. During the examination, the hash value of every fetched external JavaScript file will be compared with the stored hash values. A match in the comparison will exempt the external file from further inspection. The same whitelist scheme can also be applied to web pages when JStill resides in a Web proxy and inspects all the incoming Web contents going through the proxy. When an http request hits the cache in the proxy, the requested Web page must have been examined by JStill; hence, JStill need not inspect the page again.

### 3.3.6 Prevention of Malicious Obfuscated JavaScript Code

The runtime inspection component can not only be used in the detection of obfuscated malicious JavaScript code, but also in prevention of the execution of detected malicious code in a browser.
For obfuscated malicious code that uses invocations of D-Gen and R-Eval functions, since JStill intercepts the arguments of the invocations, it can replace the malicious content with NULL and continue the execution. For obfuscated malicious code that uses user-defined functions, the same approach may disrupt the normal execution of the benign JavaScript code in some cases. Since the basic interpretation unit in most JavaScript engines is a code segment enclosed by "<script>" tags, disabling a detected malicious function will lead to the following benign code in the segment being skipped. For example, as illustrated in Figure 3.15, if the detected obfuscated function invocation "mal_ob()" is disabled or commented out, the rest of the benign JavaScript code will not be executed because the type of "flag" will be "undefined".

JStill tries to prevent the malicious code from being executed while keeping the execution of benign JavaScript intact. Therefore, JStill substitutes the invocation of a detected malicious user-defined function with an invocation of a NULL function, which does nothing except returning a NULL. In this way, when a user’s browser renders the web pages, these NULL function invocations will reduce the possibility of interrupting the execution of benign JavaScript code.

One concern regarding this substitution-based prevention scheme is that the false positives in obfuscation detection may cause loss of functionalities in the Web pages. However, based on our evaluation, which will be described in Section 2.4, normally this is not a big issue even a false positive occurs.
3.4 Evaluation

In this section, based on a prototype implementation, we evaluate JStill in terms of (1) detection effectiveness and (2) performance overhead. The prototype of JStill is implemented in Firefox (version 3.6), which uses a rendering engine Gecko (1.9.2) and a JavaScript engine SpiderMonkey (version 1.8). The implementation adds 1.1 KLOC into the source code of Firefox. We also automate the process of rendering a Web page from the instrumented Firefox using Python scripts such that the browser can check against either a list of URLs or a directory of offline Web pages.

3.4.1 Evaluation Setting

The prototype of JStill is tested in Ubuntu 8.0, which runs a Pentium 4 3.4 GHz single-core CPU, 1 GB RAM, 160 GB 7200 RPM hard drive and 100 Mbps ethernet interface.

Sample Collection We collect both benign and malicious samples from the real world. The benign sample set consists of Web pages crawled from Alexa [52] top 50,000 websites. The malicious samples are collected from VirusTotal (flagged by ≥ 5 AV vendors). There are two sets of malicious samples. The first set contains 2,327 samples, among which 1,499 samples include obfuscated malicious JavaScript code (identified by manual examination). The second set contains 10,501 samples. Since these samples have already been detected by AV vendors, to better evaluate the effectiveness of JStill in detecting obfuscated malicious code, we apply 3 JavaScript obfuscation tools on each sample in the second set. This process generates another 31,505 obfuscated malicious samples.

<table>
<thead>
<tr>
<th>Obfuscation Metric Threshold</th>
<th>FP</th>
<th>FN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.19%</td>
<td>0%</td>
</tr>
<tr>
<td>0.9</td>
<td>1.90%</td>
<td>0%</td>
</tr>
<tr>
<td>0.8</td>
<td>1.89%</td>
<td>0.13%</td>
</tr>
<tr>
<td>0.7</td>
<td>1.75%</td>
<td>0.53%</td>
</tr>
</tbody>
</table>
3.4.2 Detection Effectiveness

Table 3.3 shows the overall detection accuracy of JStill in the evaluation using both benign and malicious sample sets. Each row in Table 3.3 corresponds to a different obfuscated malicious argument (OMA) metric threshold used in evaluation. Note that OMA metric is the only adjustable detection criterion in JStill, the other two criterions are obfuscated function definitions and disguised invocations of language-defined functions. The purpose of choosing different values is to understand how detection accuracy (i.e., FP and FN) is affected by the threshold of OMA metric.

One insight from the results in Table 3.3 is that the OMA threshold leads to a trade-off between false positive rate and false negative rate. When the threshold is low (e.g., 0.5), the false positive rate is also low (1.63%). This is because some arguments of benign invocations of D-Gen and R-Eval functions contain strings that cannot be found in Web pages; thus, these arguments have relatively low OMA metric values. However, when JStill adopts a low threshold on obfuscation metric, these arguments may not cause false positives, hence the false positive rate is low.

On the other hand, when the threshold is high (e.g., 1.0), the false negative rate becomes low (0%). A high threshold means that an argument can be considered as benign only when a large portion of the argument can be found in the Web page. Attackers can increase the OMA metric of some malicious arguments (e.g., by only obfuscating part of the malicious content and leaving the rest of the arguments in plaintext) and cause false negatives by surpassing the threshold. However, this becomes very hard when the threshold is set high, because passing a high threshold requires most of the arguments not being obfuscated, in which case the chance of malicious code being detected by signature-based approaches also increases. Note that JStill is not designed to replace the signature-based schemes, but instead they are complementary to each other.

<table>
<thead>
<tr>
<th>Cause of False Positives</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obfuscated arguments of D-Gen and R-Eval functions</td>
<td>95.89%</td>
</tr>
<tr>
<td>Disguised invocations of language-defined function</td>
<td>0.46%</td>
</tr>
<tr>
<td>Obfuscated function definitions</td>
<td>3.65%</td>
</tr>
</tbody>
</table>
**False Positives** Table 3.4 lists the false positives incurred on each detection criterion of JStill. Most of the false positives are caused by obfuscated arguments of D-Gen and R-Eval functions. There are two main causes of the non-negligible false positives. The first cause is that information generated at runtime (e.g., random number, user inputs) takes a large portion in the arguments of the R-Eval functions. Given the fact that the arguments of R-Eval functions are JavaScript code and a large portion of the code cannot be observed in any static perspective, this case is very similar to that of obfuscated malicious code.

The second cause is that some benign Web pages actually adopt encoding-based obfuscation on some parts of their JavaScript code. One example is the Web page retrieved by the URL “www.360buy.com”. In this Web page, the argument of an invocation to “eval” is encoded using a customized encoding function. Meanwhile, a decoding function is also observed as part of the code. After decoding, the execution of “eval” evaluates a large body of JavaScript code, which is actually a Jquery library. Given the open-source nature of Jquery, the purpose of this obfuscation is not clear. Indeed, when benign code adopts the same obfuscation techniques as malicious code, the problem of differentiating one from the other is probably undecidable. We believe this problem may only be solved by observing the runtime behavior, which, however, is not an efficient approach to be deployed in any large-scale or realtime scenarios, not to mention the challenge in traversing all execution paths.

One concern regarding the false positives is the possibility of interrupting user’s browsing experience. However, in reality this normally is not a big concern. This is because: 1) the prevention scheme in JStill does not hinder the execution of the rest of the code; 2) Most of the false positives only affect a single function invocation in a Web page. Considering the popularity of tools such as NoScript, nullifying a single JavaScript function invocation would probably not affect user’s browsing experience.

It is also worth noting that we implicitly assumed that none of the 50,000 Web pages as well as their linked .js files is malicious in our evaluation. To verify whether it is the case, we will have to resort to a dynamic analysis approach (e.g., [20]). However, if some of these Websites are indeed malicious, the false positive rate of JStill will only be lower.
False Negative The analysis of JStill’s false negative rate is based on the examination of both obfuscated malicious sample sets. The overall false negative rate is listed in the third column in Table 3.3.

Since false negative rate is related to the threshold of the OMA metric, when the threshold is high (e.g., 1), the overall false negative is 0. That is to say JStill can detect all the malicious obfuscated JavaScript code in our malicious sample set when the threshold is set to 1. When the threshold is low (e.g., 0.7), false negatives start to happen in the first set of malicious samples. These false negatives are incurred on the criterion using the OMA metric. Most of the samples causing false negatives obfuscate only a part of the malicious arguments in D-Gen and R-Eval invocations. As we discussed before, this will also increase the chance of being detected by signature-based approaches. In fact, these samples are detected by multiple AV vendors. There is no false negatives in the second set even when the threshold of OMA metric is low. This means JStill can effectively detect obfuscated malicious code that is generated by JavaScript obfuscation tools.

3.4.3 Comparison with Other Techniques

Although there are many works on malicious JavaScript code detection, these works either detect general malicious JavaScript (e.g., Prophiler [59], JSAND [16]) or focus on specific malicious JavaScript (e.g., Nozzle [60], Zozzle [11]). Therefore, they are not comparable with JStill. However, several other works also focus on the detection of obfuscated (malicious) JavaScript code. For example, Kaplan et al. proposed NoFus [17] to detect obfuscated malicious JavaScript code; Kim et al. proposed an approach [15] to analyze the obfuscated JavaScript code in malicious websites. We compared the results of these two approaches with the results of JStill in terms of detection effectiveness. We found that JStill has similar false positive (FP) rates with NoFus (i.e., NoFus reported 1% false positive rate) and lower FP rates than the approach in [15]. JStill has lower false negative (FN) rates (i.e., 0.53%) compared with these two approaches (i.e., 5% in NoFus and 3.84% in [15]). However, we also noted that the comparison is not based on the same sample set. This is because each work obtains its own sample set through different approaches and the samples are not shared among different works.
3.4.4 Performance Overhead

In our evaluation, performance overhead is measured in terms of the average increased loading time for Web pages. To factor out the random fluctuations in network latency, JStill automatically visits the same website 20 times. Meanwhile, the option of caching visited Web pages is also disabled in the instrumented browser to make sure the Web pages are retrieved from the Web server instead of being loaded from the local cache. To calculate the overhead in loading time, the same evaluation is performed again using a non-instrumented Firefox, and the difference between the results from two evaluations is the performance overhead. Figure 3.16 lists the average performance overhead in loading time for Alexa top 20 websites. The overall average performance overhead is 4.9%, which makes JStill a practical realtime detection tool.
3.5 Summary

In this chapter, we first conduct a measurement study on the usage of obfuscation techniques in malicious JavaScript code. We find that JavaScript obfuscation is a common practice among malicious JavaScript code in order to evade detection. Moreover, some malicious code employs multiple levels of obfuscation to further complicate the analysis and to better hide their malicious purposes. The results demonstrate that all popular anti-virus products can be effectively evaded by various obfuscation techniques. From this measurement study, we better understand that JavaScript obfuscation techniques are dangerous threats to Web security due to the simplicity of applying such techniques on existing malicious codes and the lack of efficient and effective detection approaches.

In this chapter, we also present JStill, a mostly static approach to detect obfuscated malicious JavaScript code. JStill focuses on three aspects of function invocation analysis to provide efficient and effective detection and prevention of obfuscated malicious JavaScript code. It leverages the comparison of information obtained from both static analysis and runtime inspection. An evaluation has demonstrated the detection effectiveness of JStill as well as the low performance overhead. We see JStill as a good and practical complementary approach to existing signature-based detection systems. We also believe the design of JStill can shed some light on other obfuscation detection problems.
Chapter 4

Early Warning Detection of OSN Worms

As one of the first network security threats, Internet worms have been discussed and studied by many researchers. A set of approaches have been proposed to detect the propagation of worm in the network. In recent years, as online social networks (OSN) boosting, especially when these websites have accumulated a large user base (e.g., more than 1 billion users on Facebook), worm started to propagate in online social networks as well. By exploiting the small-world and scale-free properties of online social networks, OSN worms exhibit fast spreading characteristics similar to the ones observed in Internet worms.

However, existing Internet worm detection mechanisms can not be directly applied to detecting OSN worms. This is because Internet worm detection heavily relies on the unique patterns of worm scanning traffic or the misbehavior of infected hosts, but neither of them can be observed in the propagation of an OSN worm. From the perspective of OSN websites (i.e., the server side), an infected user account does nothing but sending messages or posting updates as normal users do when the actual infection is taking place in the browser (i.e., the client side). This makes the detection of OSN worms a new and interesting problem.

In this chapter, we discuss the characteristics of OSN worms and propose an early warning OSN worm detection system. Early warning is essential for OSN worm detection because it provides administrators the opportunity to apply worm containment and elimination measures without affecting a large portion of user
accounts. Meanwhile, achieving early warning in the detection of fast spreading worms is also a challenging problem [23].

Our approach leverages the properties of online social networks and the inherent propagation characteristics of OSN worms. More specifically, based on the small-world and the scale-free properties of the social graph, we first build a surveillance network to collect suspicious worm propagation evidence. We maximize the surveillance coverage by monitoring only a small fraction of user accounts. These accounts are selected by investigating the vertex properties of the social graph. We also realize that detection that is based on suspicious evidence alone is prone to high false positives, because worm activities are highly likely to be drown out in normal user activities. As such, we further propose a scheme to effectively filter out the noise in the surveillance network.

4.1 Background on OSN Worms

Various OSN worms spread themselves by exploiting different features of the OSN websites. Nonetheless, their propagation vectors share certain similarities, which will be characterized by examining two representative OSN worms: Koobface and Mikeyy.

4.1.1 Koobface Worm

Figure 4.1 illustrates the propagation flow of Koobface in Facebook. User A receives a worm message from one of her friends (step 1) after this friend was infected by Koobface. Within this worm message, there is a link to a video clip hosted on a fake YouTube website. When user A clicks that link, the browser is redirected to the fake YouTube webpage (step 2), where the user is prompted by a request to install an update for “Adobe Flash player” plugin, which is actually a malware. After user A installs the claimed browser plugin (step 3), Koobface infects user A’s Facebook account and iterate its infection cycle by sending similar worm messages to all the friends in user A’s profile (step 4). Actually, besides sending messages, Koobface also sends invitations or composes posts, both of which contain similar worm content.
4.1.2 Mikeyy Worm

Mikeyy worm propagates by posting updates on an infected user’s profile to encourage the “follower” (people who can automatically receive the updates in their profiles) to visit www.StalkDaily.com, which was owned by the attacker. When a follower who is interested in the update clicks to see the poster’s profile, a self-replicated JavaScript code is injected into the follower’s profile. After the injection, similar updates are posted on the infected follower’s profile to repeat the infection cycle.

In summary, despite the differences in the infection vectors of these two worms (E.g., downloading malware versus self-replicating JS code), both of them propagate following the social connections (E.g., friends or followers) of an infected account; in other words, their propagations follow the topology of the online social network (i.e., the social graph). One reason of this similarity is that social connections provide worms an opportunity to exploit social engineering such that the click-through rate of the malicious content can be increased. Besides, as mentioned before, topological properties of online social networks (e.g., small-world) can facilitate the spread of worms. Another similarity shared by both worms is the
generation of passively noticeable activities such as worm messages and worm updates. This is because OSN worms are normally generated with certain malicious purposes such as advertising malicious websites or distributing malware.

4.2 Design Overview

As suggested by the high clustering property, the neighborhoods (i.e., friends) of most user accounts are densely connected. Therefore, for each neighborhood, our system only needs to monitor the “popular” user (i.e., the one with most friends in a neighborhood) to cover the entire neighborhood. Meanwhile, the scale-free property implies that a user with a large friend set tends to be friends with other users with large friend sets. This indicates that not all the popular users need to be monitored. Indeed, our system will only select a few of such users to maintain the surveillance coverage.

The general idea of our detection system is to deploy a disguised surveillance network being part of the online social networks to collect worm propagation evidence and to identify worm infections. Figure 5.1 illustrates the framework of our detection system, which consists of four major components. The configuration module retrieves from the administrator of the OSN website the social graph, based on which it determines where to collect evidence. The evidence collecting module gathers suspicious worm propagation evidence observed in an OSN website. The worm detection module identifies and reports a worm infection based on the input from the evidence collecting module. When an infection is detected, this module passes an alarm together with the infection information to the administrator of the OSN website via the communication module. The communication module provides all the necessary communications between an OSN website and the other modules. We will explain the design details of these modules in the following subsections.

One noteworthy property of our design is that each module only represents a combination of certain functionalities, and these functionalities can be implemented either within a dedicated server or in a distributed way. This property extends the flexibility of the system implementation, and it will be further discussed in Section 4.
4.3 Construction of a Surveillance Network

The detection framework is based on a surveillance network that can collect evidence about OSN worm propagations.

4.3.1 Selecting Normal Users

Because of the practical concerns on applying decoys, our system only selects a small set of users (hereinafter referred to as “selected users set”) to be friends with decoys. Meanwhile, the objective of early warning favors a sufficiently large portion of users being kept under the surveillance of our system. Hence, the question is how to choose as small as possible a selected users set to achieve early warning. We formalize this problem in the context of social graph: Given a directed graph $G = (V, E)$, where each vertex denotes a user in the social network and each edge represents a connection between two users $^1$, choose a minimum set of vertices such that each vertex either belongs to the set or there exists a path that ends at this vertex and starts from some vertex within the set. The length of this path is at most $r$ hops. This problem is also known as extended dominating set problem [61] [62], which is NP-complete.

$^1$in the case of mutual acquaintance between two users, such as in Facebook if $A$ is friends with $B$, then $B$ is also friends with $A$, this connection needs to be represented by two directed edges.
The choice of $r$ affects the size of the selected users set. Therefore, it is carefully reasoned based on the following study of a real world social graph. Our study first confirms that given the same number of users, a larger $r$ can cover a larger portion of the same social graph, so a larger $r$ is desirable if at all possible. We also find that a worm starts from a single user can infect at most 0.08% of all users in two-hop propagation and 0.26% of all users in three-hop propagation. If our system sets $r = 3$, it is very likely that by the time worm propagation is detected, 0.26% of all users have been infected. This exceeds the early warning criterion (0.19%) suggested in [23], so our system sets $r = 2$ as the coverage radius. Our study also suggests that it is not necessary to cover the entire social graph, because degree distribution of the vertices in a social graph follows power law distribution [63]. This property indicates that many vertices do not even have any connections. For example, over 20% of users in our evaluation data have no connections. These vertices are very unlikely to become the victim of a worm, and the effort of covering such vertices would produce a set with the size comparable to the size of the graph. Based on these studies of the properties of a social graph, we relax the constraint about covering the entire graph and redefine the problem as follow:

**Maximum Coverage Problem:** Given a social graph $G = (V, E)$ and a number $k$, choose a set of vertices with size of at most $k$ such that the number of other vertices that are covered by this set with coverage radius $r = 2$ reaches the maximum.

The maximum coverage problem is also NP-complete. The previous extended dominating set problem reduces to it. Since both the scale-free property and the power-law distribution of degree suggest that high-degree vertices are more likely to be infected by a worm than most other low-degree vertices, these high degree vertices should be included in the selected users set with high priority. Besides, research on modelling epidemics in topological networks [64] [65] [66] also indicates that the more edges a node has, with a higher probability it will be infected quickly by an epidemic. These results suggest the following greedy heuristic: At each step, we add one vertex into the set such that the intersection between this vertex’s 2-hop coverage and the remaining vertices is maximum. Based on this heuristic, we design the following approximate algorithm.
Algorithm 1 Maximum Coverage Algorithm

Input: Graph $G = (V, E)$
Output: Monitored user set $C$

1: $C \leftarrow \emptyset$
2: while $|C| < k$ and $V \neq \emptyset$ do
3: \hspace{1em} maxcover $\leftarrow 0$
4: \hspace{1em} for $\forall v \in V$ do
5: \hspace{2em} cover$_v$ $\leftarrow$ 2-hops coverage of $v$
6: \hspace{2em} if maxcover $<$ cover$_v$ then
7: \hspace{3em} maxcover $\leftarrow$ cover$_v$
8: \hspace{2em} end if
9: \hspace{1em} end for
10: $C \leftarrow C \cup \{v\}$
11: $V \leftarrow V - \{v\}$
12: for $\forall u \in V$ and $(v, u) \in E$ do
13: \hspace{1em} for $\forall (u, w) \in E$ do
14: \hspace{2em} $V \leftarrow V - \{w\}$
15: \hspace{1em} end for
16: \hspace{1em} $V \leftarrow V - \{u\}$
17: end for
18: Update degree of each $v \in V$
19: end while

The time complexity of Algorithm 1 is $O(knm^2)$ where $n = |V|$ and $m = |E|$.

In practice, our system pre-processes the social graph to reduce the size of the graph such that the performance of the algorithm could be improved. Besides, since social graphs grow with time, we may run this algorithm periodically (e.g., once a week) to reflect such growth.

4.3.2 Assigning Decoy Friends

After a candidate selected users set is chosen, the configuration module sends the set to the OSN administrator, who will contact these users (with incentives) and return the final set of users that are willing to accept decoy friends. Upon receiving the final set, this module creates decoy profiles in the OSN website and associates two decoy friends to each user by adding them into the user’s friends list (The justification of this scheme is discussed in Section 3.4). This module also modifies
the account preference of each decoy friend (according to the setting of the OSN
website) so that information received by decoy friends can be collected by the
evidence collecting module.

4.3.3 Communications with OSN Websites

The communication module acts as the interface of the detection system since it
processes all necessary communications between the system and the OSN website.
For example, it coordinates the communications between the configuration module
with the administrator during system setting up. It receives propagation evidence
from the OSN websites and passes them to the evidence collecting module. It
also sends the worm infection alarm to the administrator on behalf of the worm
detection module.

4.4 Two-level Detection

4.4.1 Evidence Collection

The evidence collecting module is in charge of gathering worm propagation ev-
idence (e.g., worm messages, worm updates). However, given the huge amount
of information exchanged in an OSN website, the challenge is how to collect only
suspicious worm evidence. Since OSN worms follow the social connections in prop-
agation, a friend of an infected user account is more likely to receive worm prop-
agation evidence. To leverage this advantage, we adapt the idea of honeypot here
as “decoy friend”. A decoy friend is a low-interactive honeypot, and it is created
and added into a normal user’s friends list by the detection system. When a user
account is infected by an OSN worm, decoy friends of that account can receive
worm evidence. Similar ideas have been suggested in [67, 68] for other types of
networks. In [67], the authors assume decoys only receive malicious messages.
However, the same assumption does not hold in our work. In fact, our system
treats the collected information from decoys only as suspicious evidence because
some normal user activities can also be observed by decoys.

Decoys form a disguised surveillance network. We assign each decoy to be
friends with several normal users so that a decoy can not be easily spotted because
of its small number of friends. In addition, there are a few practical concerns regarding applying decoy friends in real world OSN websites. The first potential concern is related to user’s information privacy because decoys collect suspicious information in the network. However, since users’ data are all stored and kept in the OSN websites, we think our system will not cause new data/information leakage. Nevertheless, to alleviate such possible concern, our system will only keep the suspicious information for a short period of time. The second concern is that users might be reluctant to accept decoy friends. As such, a website will need to consult its users before assigning decoy friends to them. In fact, the OSN websites could provide incentives to encourage users to accept decoy friends. After all, both users and the OSN websites try to avoid worm infections for their own benefits. The third concern is on the number of decoy friends to be deployed in an OSN website. Besides user’s reluctance, the population of decoys may negatively impact the popularity of an OSN website, because decoy friends do not contribute to any interactive activities such as discussions or communications. To this end, our system strives to limit the number of decoy friends while preserving the detection effectiveness. We will discuss this design issue in the next section.

4.4.2 Worm Detection

This module identifies the infected user accounts based on the suspicious worm propagation evidence. To distinguish actual worm evidence from normal user communications, this module applies correlation test on the suspicious evidence. The correlation test is based on similarities in the content and the structure of worm propagation evidence. One reason behind this similarity is that worm messages or updates composed by the same worm usually serve the same purpose (e.g., advertising a malicious link). Another reason is that the automatic message generation algorithms run by worms tend to reuse words and phrases because of the limited size of their candidate words set.

In this module, we employ a two-level spatial-correlation scheme, namely local correlation and network correlation. To provide necessary information to correlations, our system maintains a data structure called suspicious propagation evidence list (SPEL), which is associated with each selected user. In SPEL, every piece of
evidence is stored as a \{decoy friend ID, receiving time, content\} tuple.

**Local Correlation:** Local correlation performs similarity test among suspicious evidence collected by two decoy friends assigned to the same selected user. The purpose of associating two decoy friends with one user is to offer a local reference such that upon receiving any evidence from one decoy friend, the system can search the other decoy friend’s SPEL for similar evidence. One of the following scenarios will happen:

1. Only one of the two decoy friends has received this message. With a high probability, this is worm propagation evidence because a normal user is unlikely to send messages to one of his/her decoy friends, especially given that he/she knows which is a decoy.

2. Both decoy friends have received *similar but not identical* messages. With a high probability, this is worm propagation evidence because only the infected users send customized worm messages to each friend.

3. Both decoy friends receive the same message or update \(^2\). It could be either a group message or a worm message with the same content. In this case, the scheme resorts to *network correlation* for further identification.

**Network Correlation:** Network correlation is performed with input from all decoy friends. Upon receiving the same evidence from two decoy friends of a user, network correlation searches for similar evidence by computing the similarity score between the received evidence and any other evidence in the SPELs of other decoys. If similar evidence (e.g., with a similarity over 90\%) is found, with high probability, both pieces of evidence can be confirmed as worm propagation evidence. We realize that some normal communications among users may have the similar propagation pattern as worm messages. For example, the outbreak of a large-scale event may cause similar or same messages distributed within an OSN. We will discuss this case in Section 6.

To examine the *similarity* between two pieces of suspicious propagation evidence, our scheme applies a simple measurement of similarity, which is based on

\(^2\)Since updates are automatically displayed to all the friends by the OSN website, updates cannot appear in the previous scenarios.
the metric of edit distance \textit{editDist()} [69]. By this measurement, the \textit{similarity} between evidence \( E_a \) and evidence \( E_b \) can be evaluated as follow:

\[
\text{sim}(E_a, E_b) = \frac{1}{1 + \text{editDist}(E_a, E_b)}
\]  

(4.1)

where \textit{editDist()} follows the definition of Levenshtein edit distance [70]. We acknowledge that more complex similarity measurements may generate more accurate results, but since that is not the focus of this paper, we will consider the other metrics in our future work.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.3.png}
\caption{An example of two level correlation}
\end{figure}

Figure 4.3 gives an example illustrating the mechanism of two-level correlation. In this example, users \( A, B, C \) belong to the selected users set, and each of them is associated with two decoy friends. We assume that the worm first infects \( D \),
then it infects both $E$ and $A$. After that, $B$ is infected as well. At the time $A$ was infected, if the worm sends customized messages to $A$’s decoy friends $A1$ and $A2$, the system can identify $A$’s infection by local correlation between $A1$ and $A2$ (Scenario 2). If both $A1$ and $A2$ receive the same message (or update), the system checks the SPELs of all other selected users (e.g., $B$ and $C$) and compares the received suspicious evidence with stored evidence. Because neither $B$ or $C$ is infected at this time, no match is found. Therefore, the evidence received by both $A1$ and $A2$ is stored in the SPEL of $A$. After $B$ is infected, the same procedure is performed and a match can be found between the evidence stored in the SPEL of $A$ and the evidence just received by $B1$ and $B2$. At this time, the infection of both $A$ and $B$ can be identified.

4.5 Evaluation

We evaluate the detection system with three objectives. The first objective is early warning. We define early warning in terms of the number of infected accounts by the time the worm is detected. This metric is borrowed from Internet worm detection. As suggested in [23], a worm detection system is deemed as an early warning system if the worm propagation is detected when less than 0.19% of all vulnerable hosts are infected. The second objective is to test the detection system under various worm propagation behaviors as well as under practical constraints such as user reluctance. The third objective is to examine the effectiveness of our system in worm containment with simple countermeasures provided.

4.5.1 Data Collection

The evaluation uses a set of real-world OSN data. The data is crawled from the popular OSN website Flicker [71] for a measurement study [63]. There are 1,846,198 users and 22,613,981 friend links in the social graph, and the average friend number per user is 12.24.
4.5.2 Simulation Model

Our simulation model consists of four modules as shown in Figure 4.4. The initialization module takes the social graph as input and output the set of selected users (the default size of the set is 500). For each selected user, this module adds two edges in the social graph from the vertex (representing the user) to the two new vertices (representing two decoy friends). It also creates and associates a SPEL with each selected user. Worm propagation starts from certain user(s). The propagation module models worm propagation by repeating two consecutive phases, namely sending worm messages (both messages and updates are referred as messages in evaluation) and infecting users. In the process of sending worm messages, our simulator chooses either to send worm messages to all the friends or to a fraction of friends randomly chosen from the friends list of an infected user. Each worm message is randomly selected from the predefined worm messages set. We assume each recipient of worm messages gets infected with a probability of $P_{user}$. The probability is randomly assigned for each user and keeps constant in each worm propagation. The monitor module performs correlation tests on suspicious worm messages. Once a worm infection is identified, the post-processing module can output the infection statistic such as percentage of infected users.

![Figure 4.4. OSN Worm Simulation Model](image)

Since a few random variables are used in each propagation, we repeat each
propagation for 100 times in our evaluation to reduce the impact of randomness. After that, we display the results as the mean values of these iterations as well as 95% confidence intervals of the means.

### 4.5.3 Early Warning Detection

In this part of evaluation, we examine whether the detection system can achieve early warning. To this end, we test the detection system in the propagation of two representative OSN worms, namely Koobface worm and Mikeyy worm. A user account infected by Koobface worm sends different (customized) messages to its two decoy friends. In Mikeyy worm propagation, an infected account delivers the same message to both decoys.

A crucial factor in worm propagation is the initially infected user account(s) (i.e., hitlist) because the total friend numbers of the initial account(s) can affect the worm propagation speed. To conduct a more comprehensive test, we choose 22 accounts from the Flickr dataset with different friend numbers (from 1 to 26,185, where 26,185 is the maximum number of friends) as initial infected user accounts and start worm propagation from these accounts in both worm cases. Then we measure the average infection numbers by the time these worms are detected and the results are showed in Table 4.1 and Figure 4.6.

Table 4.1 lists the average infection numbers on detection for both worms. The maximum infection number is 2420, which is only 0.13% of all the users. This indicates the detection system fulfills the early warning requirement. In addition, Table 4.1 shows the average infection number of Mikeyy worm is larger than that of Koobface worm. This is because the detection of Mikeyy worm infections requires both local correlation and network correlation. Next, we will use the Mikeyy worm propagation model for evaluation.

Figure 4.6 confirms that in general propagations starting from “popular” users can infect more user accounts. However, we also notice that in some cases worm

<table>
<thead>
<tr>
<th>Worm Type</th>
<th>Avg. Infection #</th>
<th>Max Infection #</th>
<th>Min Infection #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Koobface</td>
<td>700</td>
<td>1851</td>
<td>2.75</td>
</tr>
<tr>
<td>Mikeyy</td>
<td>1023</td>
<td>2420</td>
<td>2.8</td>
</tr>
</tbody>
</table>
propagation starting from less popular initial users can infect more user accounts than the propagations starting from more popular initial users. For example, in both worms, propagations starting from the user with 531 friends infects more accounts than propagations from the user with 1246 friends. The reason is that popular users are more likely to be included in the selected users set. If a user A, who has more friends than another user B, has decoy friends, the infection of
A will be detected faster than infection of B. Therefore, a worm starting from A may infect less accounts than the same worm starting from B. Another reason, as demonstrated in [63], is that social graph tends to exhibit a tightly-connected “core” of high-degree vertices connected with each other because of its scale-free metric and the positive assortatively coefficient. This property implies that when a “popular” user is infected, even if it does not belong to the selected user set, it is very likely to infect another “popular” user in a short time and this infection can be detected because the other “popular” user has decoy friends. This observation also justifies the heuristic we adopt in Algorithm 1, where we start with high-degree vertices.

4.5.4 Impact of Worm Behaviors

In this part, we evaluate the detection system under behavior discrepancy of OSN worms. More specifically, we consider that an OSN worm randomly chooses a fraction of friends of an infected user as its propagation targets. We note current OSN worms infect all friends, making them easier to detect. Our evaluation is to show the effectiveness of our scheme against more intelligent worms. The impact of this behavior discrepancy is that some decoy friends may not receive propagation evidence even if the user accounts to which they are attached have been infected. To test our detection system under this assumption, we simulate worm propagations with usage of friend lists from 10% to 100%. For each percentage, worm propagation starts from the above 22 different accounts and the average results are showed in Figure 4.7.

In Figure 4.7, all the worm propagations are detected when less than 1600 user accounts are infected. This indicates that the detection system can still achieve early warning when worms randomly choose targets from friend lists. We notice that even the worms using only 10% of the friend lists can still be detected. Hence, as long as worms have no knowledge about the identities of decoys, shrinking the size of target lists to reduce the probability of hitting decoys is ineffective to the attacker. On the other hand, if somehow worms spot some decoys, they can evade these known decoys (we will discuss the impact of this scenario in the next section). Another trend illustrated by Figure 4.7 is that when more friends are targeted,
more accounts can be infected by worms. Therefore, OSN worms will tend to use as many contacts in the friends lists as possible if their goal is to enlarge infection.

### 4.5.5 Impact of Selected Users Set

As mentioned previously, not all the users in the selected users set are willing to accept decoy friends. Besides, we also consider the scenario where some of the decoys are spotted so that worms can avoid these decoys in propagation. To evaluate the detection system in these scenarios, we study worm propagations where only a part (randomly chosen) of the selected users set has decoy friends.

We first choose a selected users set of 2000 users by Algorithm 1. After that, we randomly choose 100 to 2000 users from this set to assign decoy friends and then run worm propagations for each case. Again, for each set, worm propagation starts from the previously mentioned 22 different user accounts and the average infection numbers by the time of detection are illustrated in Figure 4.8.

Figure 4.8 clearly shows that a larger set of selected users with decoy friends can detect worms with fewer infected user accounts. For example, 1500 selected users set can detect worm propagations when only 314 user accounts are infected. However, when the size of the set is smaller or equal to 100, the infection number
Figure 4.8. Infection Number versus the Size of Selected Users Set

is larger than 3000, which implies that early warning may not be fulfilled with a selected users set at this size. On the other hand, the infection numbers are restrained less than 1000 when the set size is larger or equal to 500. This result shows the effectiveness of our detection system under the impact of a partial working surveillance network. It also indicates that the administrator of an OSN should encourage more users (if at all possible) to accept decoy friends, e.g., with incentives.

4.5.6 Containment Measures

Upon detecting a worm propagation, the system will notify the administrators of OSN websites. In addition, the detection system can assist in suppressing worm propagations by adopting some simple countermeasures, such as warning the friends of infected users [72] (1-hop warning) by decoy friends or also warning the friends of friends (2-hops warning) if the privacy setting of a selected user allows the decoy friends to retrieve the friends information. In this study, we assume users will raise their vigilance after receiving the warning messages and for simplicity here we assume they will not be affected by the worm. The following results demonstrate the effectiveness of such warning mechanisms in worm containment.
Figure 4.9. Worm Propagation versus Different Containment Measures

Figure 4.9 compares the infection numbers with and without warnings (both 1-hop and 2-hops warnings). The results are based on a setting where a set of 500 selected users are used to detect and warn other users. Figure 4.9 illustrates that warnings can effectively suppress the worm propagation. A 2-hop warning approach can limit the infected user number to a small value compared with the case in which no warning is issued. A 1-hop warning prevents nearly half of the users from being infected compared with the no warning case. These results with the simple countermeasures (e.g., 1-hop and 2-hops warnings) indicate the detection system is helpful in worm containment.

4.6 Example Implementation

We discuss a practical implementation of the worm detection system in detail based on a real-world OSN website, Facebook. The system can reside in a dedicated server.
It has access to the Internet so that the server can visit Facebook and communicate with the administrator of Facebook through a secured channel (e.g., HTTPS or SSH). The modules run as programs on the server. All the modules have access to a database, which stores datasets such as the social graph of Facebook, selected users set, accounts of generated decoys and their login credentials, SPELs set, and identified worm evidence. The configuration module retrieves the social graph from the database and outputs selected users set to the database. It can also invoke an instance of a Web browser to register the decoy accounts on Facebook. The evidence collecting module is composed of an email client and a Web browser. By configuring the preference of each decoy account in Facebook, this program will be notified via emails about any suspicious evidence received by decoys. Upon the arrival of a notification email, the program of collecting module logs into the decoy account through the Web browser, retrieves the suspicious evidence and writes the evidence into the corresponding user’s SPEL in the database. After that, the program invokes the worm detection program to process the evidence. The worm detection program can send notification to the administrator of Facebook via the secure channel if it identifies any infection based on the existing evidence in the database.

Although this implementation is based on Facebook, the features of Facebook used by the system (e.g., email notification) are widely supported by other OSN websites. Therefore, we believe this implementation can be adapted for other OSN websites with a slight modification. Moreover, our system can also be implemented in a distributed way. For example, the detection functionalities of the system may be distributed among decoys such that each decoy can perform its own worm detection by sharing suspicious evidence with other decoys.

4.7 Summary

In this chapter, we design a system that can effectively detect the propagation of OSN worms. By exploiting the properties of OSNs, we construct a surveillance network embedded in the OSN websites using decoy friends. We also propose an algorithm that is based on the heuristic derived from the topological properties of social graphs to keep the OSN websites under surveillance by monitoring only a
few hundreds of users. We leverage both local and network correlations of worm propagation evidence in our detection system to achieve early warning detection.

Based on the real-world social graph of Flickr, our evaluation with two known worms, Koobface and Mikeyy, shows that the detection system can effectively detect OSN worm propagations when less than 0.13% of total user accounts are infected. Even taking user reluctance into consideration, our system can still achieve early warning detection. Moreover, the detection system is also demonstrated to be applicable to worm containment by adopting some simple countermeasures. This can provide valuable assistance to OSN websites in fighting against worm propagations in future.
Chapter 5

Automatic Analysis of the Use of Permissions in Android Apps

Android has become one of the most popular mobile platforms for third-party applications. There are over 675,000 active applications [17] in Google Play App Store (i.e., the former Android Market) and the growth rate of new applications is over 12,000 per month [9]. There are also more than 100,000 active publishers who are publishing new applications [9]. One of the reasons for the popularity of Android platform and the prosperity of Android developers community is a comprehensive framework API supported by Android. This API provides applications with the capability of accessing hardware information (e.g., GPS location), reading phone state, reading/writing user’s data, modifying phone settings, etc.

Some of the API methods involve security/privacy sensitive operations. Therefore, Android enforces a permission-based mechanism to restrict and to control the use of these sensitive API methods. Android requires applications to statically declare all the permissions they need, and Android will prompt declared and potential dangerous permissions to users at the beginning of the installation process. Users can choose to accept all the permissions or to cancel the installation. Once the required permissions are granted by users, they cannot be withdrawn until the application is uninstalled. Besides, permissions cannot be granted at runtime.

There are several problems with the permission-based mechanism. First, the prompting before installation only provides users with high-level, coarse-grained information about the requested permissions. For example, the description of
“INTERNET” permission is “full Internet access”. Based on the descriptions, users can neither know the detailed use of requested permissions in an application (e.g., where and how the permission is used), nor understand the potential risks of granting the permissions. In fact, Au et al.[73] show that users often ignore the permission requests partially because the information is too coarse-grained to reflect the details of the permission use. As a result, providing fine-grained information in the prompt is necessary for users to make informed decisions, and it can also improve the effectiveness of the install-time permission mechanism.

Another problem is that install-time permission mechanism does not examine the actual use or the necessity of requested permissions before asking user to grant the requests. Stowaway [31] partially solved this problem by statically examining requested permissions in applications using a permission-to-API-calls map. However, it is difficult for static examination to determine whether an API method is necessary or whether it will be invoked. For example, malicious developers can insert unnecessary but permission related API methods into an application’s source code to obtain unnecessary permissions. Besides, Java reflection makes static analysis-based approaches prone to inaccuracies [74, 75].

In view of these issues in the Android permission mechanism and the limitations of the existing analysis tools, we aim to provide an approach that can analyze the use of requested permissions in greater details. Besides, the analysis has to be accurate and resilient to static evasions as we mentioned above. Moreover, given the large number of existing applications and the high increasing rate of new applications, the analysis has to be performed in an automatic and efficient way.

To this end, we propose Permlyzer, a framework to automatically generate analysis of the use of Android permissions. Permlyzer can identify where a permission is actually used in the execution of the application. It can also determine how the permission is used by analyzing the context of API calls that trigger the permission check. In order to perform an accurate and thorough analysis, Permlyzer examines both an application’s runtime behavior and its source files. Based on the analysis of the use of permissions, Permlyzer can further evaluate the privacy/security risks of the application. The information generated by Permlyzer can not only help application users to make informed decisions before installation, but also help application vendors to vet applications before releasing them to the public.
5.1 Android Permission Model and Android Applications

5.1.1 Android Permission Model

Android is a privilege-separated operating system for mobile devices including smartphones and tablets. Applications need to explicitly specify their requirements to access resources and data from the system or from other applications by requesting permissions. Before users install an application, the Android system prompts users with the requested permissions. Users can only grant all the requested permissions by the application or deny all of them at the cost of not installing the application. Once a permission is granted at install-time, the application can have permanent access to the resource protected by that permission until it is uninstalled by users or remotely removed by the market administrator [76]. No additional permissions can be granted during runtime; therefore, any attempt by an installed application to use permissions that are not granted at install-time will result in a permission check failure. When a permission failure happens, users are not prompted by Android platform. Instead, a SecurityException will be thrown back to the application. As a result, it is a developer’s responsibility to avoid and handle SecurityException to ensure his/her application runs smoothly.

5.1.2 Android Permission Declaration and Enforcement

Both the Android system and an application can define permissions, but most of the permissions requested by Android applications are defined by the Android system because these permissions controls the access to the resources and functionalities provided by the Android system. There are 130 Android system defined permissions [73], among which 122 permissions are available to third party applications [77]. Permissions are defined with four different protection levels, which characterize the potential risks implied in the permission and enforce different install-time approval processes. These four levels include: 1) Normal 2) Dangerous 3) Signature and 4) SignatureOrSystem. Only dangerous permissions are presented to users for their explicit approval. Signature permissions are automatically granted when requesting application is signed with the same certificate...
as the application that declared the permissions. SignatureOrSystem permissions are essentially limited to applications that are pre-installed in Android’s “/system” partition OR signed with the firmware key. Normal permissions are always granted by the system automatically. Android defined permissions are checked when an application tries to interact with the Android API or to access a system content provider or to send and receive specific system Intents.

In this dissertation, we focus on Android-defined permissions that has a dangerous protection level, since these permissions are more frequently used than others and they may cause security and privacy risk to the system and to users.

The naming of a permission in Android follows the Java naming convention, e.g., all Android system defined permissions share a common pre-string “android.permission.*” in their names. Other than the Android permissions, any application can actually define and enforce its own permissions. An application-defined permission is declared by adding a `<permission>` tag in the `AndroidManifest.xml` file. These permissions are used to limit access to specific components and/or features of the applications, e.g., preventing one application from launching activities of other applications, governing access to a content provider of another application, controlling binding or starting a service of other applications, etc.

There are 137 Android system defined permissions [73], among which 122 permissions are available to third party applications, as recommended in [77]. These permissions are defined with four different protection levels, which characterize the potential risks implied in the permission and enforce different install-time approval processes. These four levels include: 1) Normal 2) Dangerous 3) Signature and 4) SignatureOrSystem. Only dangerous permissions are presented to users for their explicit approval because these permissions give applications access to private user data or security sensitive API calls (e.g., dangerous permissions are required to collect phone identification information and to send SMS messages). Signature permissions are automatically granted without user’s approval when the requesting application is signed with the same certificate as the application that declared the permission. SignatureOrSystem permissions are essentially limited to applications that are pre-installed in Android’s “/system” partition OR signed

---

1In the rest of this paper, we refer to an Android permission by using only the last segment of the permission name, e.g., “android.permission.INTERNET” → “INTERNET”
with the firmware key. In contrast to these three levels, normal permissions are always granted by the system automatically since they present minimal risk to the system, the user and other applications.

Android defined permissions are checked when an application tries to interact with the Android API. There are over 8,000 API methods defined in the public API of the Android platform [78]. These API methods are frequently invoked by Android applications to perform various operations. Therefore, some of these methods are protected by Android defined permissions. In addition to system API, Android defined permissions are also checked when an application tries to access a system content provider or to send and receive specific system Intents. In this dissertation, we focus on Android-defined permissions that has a dangerous protection level, since these permissions are more frequently used than others and they may cause security and privacy risk to the system and to users.

5.1.3 Android Applications

Android applications are distributed in a compressed file format (i.e., .apk file) that contains a manifest file (i.e., AndroidManifest.xml), compiled Dalvik executables (i.e., class.dex) and other resource files (e.g., files in the “res/” folder). The manifest file not only lists all the permission requests and permission definitions, it also enumerates all the components of the application. The resource files include definitions of UI layouts, application’s menu, raw resource files, etc. The information in these files is used to render UIs.

Android applications are built upon application components, which include four types: activities, services, content providers and broadcast receivers. Activity components define the application’s user interface (UI) by implementing the View or View group objects that render a specific area of the smartphone’s screen. Both service and activity components implement the functionality of the application except service components do not provide user-interface. Content providers provide database functionality to share application data between applications. Broadcast receiver components receive the send “Intents” between applications and system. Each of these application components has its own life cycle and therefore most of the components can be invoked individually. We focus on activities and services
components since most functionality of an application is implemented in these two types of components.

5.2 System Overview

![Diagram showing the system overview of Permlyzer]

**Figure 5.1.** An Overview of Permlyzer

The goal of this work is to design a tool that is capable of analyzing of permission uses in Android applications. To achieve this goal, we outline the following design requirements that are highly desired: (R1) Capability to analyze permissions from various aspects (e.g., location, cause and purpose). Since the purpose of the analysis is to help the users and the developers to understand the use of permissions in Android applications, information collected from different aspects can characterize the use of permission and provide detailed information to users and developers; (R2) Resiliency to static evasions. As we pointed out, existing static analysis-based approaches can be spoofed by inserting unnecessary or unreachable
API methods that requires sensitive permissions (e.g., an API method resides in a conditional branch which will never be reached because the condition is impossible to be met). Therefore, it is important to analyze only the permissions that are indeed necessary to the application’s functionality and actually are used in runtime; (R3) Capability to analyze different permissions. Different permissions have different purposes. Some permissions protect the access of sensitive information, and some others restrict the invocation of sensitive operations. As a consequence, an analysis approach that can only track sensitive information may not be applied on the analysis of permissions that do not involve any sensitive information. Given this, the analysis approach should be generic so that it can be applied to various permissions with different purposes; (R4) Scalability to analyze a large number of applications. Since both the number of existing Android applications and the increasing rate of new Android applications are high, the analysis approach should be efficient so that it can analyze millions of applications.

To meet these requirements (R1-R4), we design Permlyzer to leverage information from both application’s runtime and source code. Figure 5.1 illustrates an overview of Permlyzer. At a high level, Permlyzer first decodes and decompiles an application’s installation package into Java source files. It extracts meta-information (e.g., list of requested Android permissions) about the application. For each requested Android permission, Permlyzer lists the Android API methods, the invocation of which can trigger the permission check, based on the permission-to-API-calls map [31]. After that, it automatically explores the functionality of the application and logs the execution. These log files are then processed using method profiling to identify the permission triggering API calls and to analyze the context of these calls. Method profiling also leverages static analysis to locate the identified API calls in the Java source files. Based on the call stacks of the permission triggering API calls, Permlyzer can analyze the use of each checked Android permission in terms of location, cause and purpose of the permission use. In the analysis, Permlyzer can also evaluate the potential security/privacy risk in the use of Android permissions.

**Design Challenges.** We met several challenges in our design of Permlyzer. First (C1), in order to perform comprehensive and thorough analysis on requested permissions, most, if not all, of the permission related functions in an application
must be executed. However, Android applications are event-based applications, specific user events (e.g., touch a button, navigate through menu) are necessary in order to execute a function. This poses the first challenge: how to explore all the permission related functions of an application. Second (C2), functionality exploration generates a large amount of log information containing millions of method invocations on various levels. How to efficiently parse the logged information in order to generate call stacks for permission triggering API calls is another challenge. Third (C3), there is a semantic gap between the information generated by method profiling (i.e., call stacks and call sequences of permission triggering API calls) and the explanation of the permission use (i.e., the cause and purpose of a permission being checked). How to determine the use of permissions based on call stacks is also a challenging issue. We will elaborate how we address these challenges in the next section.

5.3 Automated Analysis of Permission Use

5.3.1 Functionality Exploration

Permlyzer explores the functionality of an application by invoking all the activity and service components in that application. Each Android application has one entrance activity so that when users start the application, the entrance activity will be executed first. Nevertheless, most application components (i.e., activity and service) can be executed individually. As such, Permlyzer first identifies all the activity and service components, and it then starts each of the identified components individually to reveal the functionality implemented in that component. The identification of activity/service components are achieved by parsing the meta-information stored in the AndroidManifest.xml file. More specifically, Permlyzer parses the ids of “<activity>” and “<service>” tags in the AndroidManifest.xml file. To start a component, Permlyzer parses the intent-filter and sends out Intent message to the target component using the Android debug bridge (adb) console.

Each activity component defines multiple functions that can only be triggered by proper user events such as touching a specific button. To trigger these functions, Permlyzer first leverages the layout information of the user interface. Each
activity has its layout information, which specifies the types of the UI elements (e.g., textview, button, etc.) as well as their positions. With the layout information, which is stored in the “/res/layout/”, Permlyzer can send specific user events to the positions of the target UI elements so that it can automatically trigger the functions. Permlyzer uses the adb tool and a testing tool MonkeyRunner [79] to generate and to send user events. However, the positions of some UI elements are not explicitly defined. This is because either the positions are inherited from the parent objects in the view tree, or the positions are relative in order to fit in various screen sizes.

To this end, Permlyzer leverages a second mechanism to interact with the UI elements. Permlyzer uses the trackball movement events, the function of which is similar to the Tab order in a form. By sending enough trackball movement events, each UI element that can receive focus will be selected at least once. Following each trackball movement event, Permlyzer sends a set of user events so that it can trigger the function associated with the selected UI element.

By applying both activity-based and layout-based UI elements interaction, Permlyzer can automatically explore most of the functionality of an activity component. We note that the mechanism has some limitations and we will discuss them in Section 4.5. Service components do not provide any user interface. Therefore Permlyzer does not need to inject user events in order to trigger the functions in service components.

5.3.2 Call Stacks Construction

The construction of call-stacks for hundreds of candidate API methods based on the logs generated by functionality exploration proves to be a very time-consuming task. The reason is that a single log file normally contains several millions of method invocations and there are usually tens of log files generated for one application. The efficiency of call-stack construction process is important for Permlyzer to be able to scale to a large number of methods and Android applications (e.g., 400,000). To this end, we propose a search tree-based algorithm to efficiently construct call stacks for candidate API methods.

We first use a search tree to speed up the identification of an API method in
logs. Each leaf node in this tree represents an API name, and the depth of the tree equals to the length of the longest common substring between any two API names. For each invoked method in the log, Permlyzer compares the name of the method to the names of the candidate API methods. The comparison is performed by following the path composed of the characters in the method name. If there exists such a path in the tree and the path ends at a leaf node, a match to one of the candidate API methods has been found.

To build a call stack, an intuitive approach would be to trace back from the identified API method to the root. The complexity of this approach is $O(nm)$, where $n$ is the number of method invocations in a log and $m$ is the number of candidate API methods. To improve the performance, we propose an algorithm that can avoid tracing back by maintaining a call-stack during the search for candidate API methods. Each line in a log file contains the following information: the thread ID, the name of the method and the call-depth of the invocation. At each step, the call-stack is updated by comparing the call-depth of the current invocation with that of the previous invocation. The algorithm is listed in 2. The complexity of our proposed algorithm is $O(n)$.

### 5.3.3 Analysis of Permission Use

The analysis of permission use in Permlyzer aims at answering the following questions about a checked permission. First, where the permission is used (i.e., what API method triggers the permission check?) Second, what causes the use of the permission (i.e., what action triggers the permission check?). Third, what is the purpose of the permission use? To answer these questions, Permlyzer uses information collected from both runtime and static examination.

#### 5.3.3.1 The Location of Permission Use

To identify where a permission is used, Permlyzer searches for the API methods that can trigger the check of the permission in the logs recorded in functionality exploration. For each found API call, Permlyzer also analyzes its call stack to determine the component where the API call resides in. That is, Permlyzer traces back from the API call to the user-defined method that invokes the API method.
Algorithm 2 Call Stacks Construction

Input: a sensitive API set $A = a_1, \ldots, a_n$, a log file $tf$

Output: call stacks $CS_{a_i}$ for each found API $a_i$

1: an array of thread number $TN \leftarrow empty$
2: $i \leftarrow 0$
3: while $i < \text{length}(tf)$ do
4: $\text{apiname} an \leftarrow \text{getAPIName}(tf[i])$
5: $\text{apidepth} d \leftarrow \text{getAPIDepth}(tf[i])$
6: $\text{threadid} t \leftarrow \text{getThreadID}(tf[i])$
7: if $t \notin TN$ then
8: a stack $CS_t \leftarrow empty$
9: a array $CD_t \leftarrow empty$
10: $TN.append(t)$
11: end if
12: if $d = 0$ then
13: $CS_t.push(an)$
14: $CD_t \leftarrow [0]$
15: else
16: if $d \geq \text{length}(CD_t) - 1$ then
17: $CS_t.push(an)$
18: if $d > \text{length}(CD_t) - 1$ then
19: $CD_t[\text{length}(CD_t):d] \leftarrow 0$
20: end if
21: $CD_t[d] \leftarrow \text{length}(CS_t) - 1$
22: end if
23: else
24: $CS_t.pop()$ (repeat $\text{length}(CS_t) - 1 - CD_t[d]$ times)
25: $CS_t.push(an)$
26: $CD_t[d] \leftarrow \text{length}(CS_t) - 1$
27: $CD_t \leftarrow CD_t[0:d + 1]$
28: end if
29: end if
30: if $an \in A$ then
31: $CS_{an} \leftarrow CS_t$
end if
32: end while

and the container of the user-defined method (i.e., a user-defined class). The names of all user-defined classes are obtained by parsing the source files so that they can be distinguished from Android libraries and Java native methods.

5.3.3.2 The Cause of Permission Use

The cause of a permission use instance (i.e., a permission check) consists of two pieces of information: the action/event that triggers the permission check; the type
(i.e., activity or service) of the application component where the triggering API method resides in.

Permlyzer determines the type of the component based on the user-defined class that contains the triggering API method and the component type information parsed in the functionality exploration step. That is, Permlyzer locates the container user-defined class in the component where the class is defined based on the source tree structure of the application.

We are interested in the most direct action/event that causes the permission check. In Android, action/events are processed by corresponding event handlers, the identification of which is not straightforward, since an event handler can be registered in several ways: 1) overriding the default event handler of a View class; 2) registering a customized event listener; 3) implementing a customized event handler. To this end, we leverage the fact that in Permlyzer, events are program injected (i.e., by MonkeyRunner). More specifically, to identify event handlers, Permlyzer instead identifies the event injection methods since these methods are always followed by the corresponding event handlers. After that, Permlyzer traces back the call stack of the triggering API call to determine the most direct event/action that causes the permission check.

A challenge in determining the cause of a permission usage instance is that we can not simply trace back to the root of a call-graph and use it as the cause. For most call-graphs generated by Permlyzer, the root represents the start of a thread in the app’s process. From the aspect of programming, a permission usage instance can be triggered by two causes. First, if the usage instance is thread-triggered, i.e., the permission will be used as long as the thread is created, the real cause of the instance is actually the cause of the creation of the thread. On other hand, if the instance is event-triggered, i.e., the permission is used after user touch a button on the screen, the cause of the instance is actually the event (e.g., touch a button) or the object that receives and processes the event (e.g., the button object). Therefore, in order to determine the cause of a permission usage instance, we traverse its call graph backward applying a heuristic that if the API call is not triggered by an event, then it must be caused by the same reason that creates the thread it resides in.
5.3.3.3 The Purpose of Permission Usage

Permlyzer determines the purpose of a permission usage instance from two aspects. First aspect, the functionality of the API call that triggers the permission check. For example, a call to API “android.location.LocationManager.getLastKnownLocation()” indicates the reason to check the permission “ACCESS_FINE_LOCATION” is to obtain the last known (cached) geographic location information on the smartphone. Meanwhile, a call to API “java.net.HttpURLConnection.<init>” indicates that the reason to request “INTERNET” permission is to start a HTTP connection to a remote server. The information obtained from examining the functionality of triggering API calls is helpful in determining the purpose of permission checks, but not comprehensive. Since one permission check may be related to another check, and only their relation can expose the true purpose of both permission checks. For example, a check of “READ_PHONE_STATE” (e.g., to collect phone identification information) followed by a check of “INTERNET” (e.g., to communicate with a remote server via Internet) suggests that the purpose of both checks is to send collected identification information to a remote recipient.

To this end, Permlyzer uses the correlations between multiple permission checks as the second aspect in the analysis of the purpose of permission use. Two API calls are correlated if they appear on the same execution path. Permlyzer discovers correlations among individually identified permission checks based on the call stacks of their triggering API calls. More specifically, it compares one call stack with another to find a common sub-sequence of calls between them. When correlations are discovered, Permlyzer combines the call stacks together to form a new call stack to represent the correlation.

5.3.3.4 Evaluation of Potential Risks in Permission Use

Permlyzer evaluates potential risks in permission use by comparing the analyzed instances of permission use to known malicious use patterns. These patterns are obtained from our analysis of malicious applications. Given the large number of applications, the first step in the comparison is to filter out applications that do not request any combination of permissions that is necessary to perform malicious behavior. These combinations are also obtained from our analysis of malicious
applications. This step can effectively reduce the number of applications need to be compared. For the rest of the applications, Permlyzer compares the correlations (if any) of their permissions use with a set of correlations of permission use found in malicious applications to determine whether the correlations indicate any malicious behaviors of the application.

For a sensitive permission, each call stack of every permission related API calls are compared with the call stacks of the same API call in the database. For a permission combination, each call sequence between permissions related API calls is compared with the call sequence between the same API calls in the database.

### 5.4 Analysis of Permission Usage in Android Applications

We evaluate the analyzing capability and scalability of Permlyzer by applying it on known malicious applications and free applications respectively.

#### 5.4.1 Collection of Android Applications

As our first dataset, we collected 325 known malicious Android applications from VirusTotal [51]. All of the collected malicious applications have been detected by more than four anti-virus vendors listed on VirusTotal.

The second dataset is free Android applications. This dataset consists of 113,237 free Android applications collected from 25 categories in Google Play App Store. Since Google Play App Store lists only top 500 applications in any category (other applications can only be accessed by search), we built an application crawler that can automatically discover package information of available applications in Google Play App Store from an external website. After the package information is crawled, the application crawler retrieves the installation package of the discovered application from Google Play App Store to a local server.

---

2The collected malicious applications are submitted to VirusTotal between Dec, 2011 to March, 2012
5.4.2 Permission Analysis Coverage

In the evaluation, Permlyzer covers 90% of the permissions requested by applications. Note that Permlyzer leverages the API-to-permission map provided in Stowaway [31]. Therefore, if a permission does not map to any API in the API-to-permission map, Permlyzer cannot analyze the use of that permission. We acknowledge that the API-to-permission map obtained in [31] is based on Android 2.2 (Android API level 8), which are currently used in 12.9% Android devices [80]. [80] also shows that Android 2.3 (Android API level 9,10) is the most widely installed Android version (e.g., on 55.8% of the devices). Besides, the difference between Android 2.2 and Android 2.3 in terms of code happens only in 58 classes [81, 82], most of which are adding new fields instead of changing API-to-permission map.

Further investigation shows that 9% of the requested permissions cannot be located in source files. This is because either the developers make mistakes in claiming requested permissions (e.g., over-claiming, wrong permission name), or the permission is obsoleted in the evolution of Android development platform.

5.4.3 Performance Analysis

On average, Permlyzer took less than 3 minutes to analyze one application. Most of the time (i.e., over 70%) is consumed by method profiling, which includes uploading the application to a phone, installing the application, starting activities, generating logs and delete the application. The decoding and decompiling process takes less than 10 seconds on average. The time used by method profiling varies from a few seconds to tens of seconds depending on the size and number of log files.

5.4.4 Permission Use in Malicious Applications

Malicious applications often demonstrate distinct features in permission use. To evaluate Permlyzer’s capability in exposing and characterizing these features, we first manually determined the malicious behavior of these applications. After that, we used Permlyzer to analyze the permission use in each application and to discover the connection between aggregated permission uses of a malware category and its malicious behavior.
<table>
<thead>
<tr>
<th>Behavior</th>
<th>Top checked Permissions</th>
<th>Top Locations</th>
<th>Top Causes</th>
</tr>
</thead>
<tbody>
<tr>
<td>C&amp;S</td>
<td>INTERNET</td>
<td>java.net.URL.openConnection()</td>
<td>Activity.OnCreate</td>
</tr>
<tr>
<td></td>
<td>READ_PHONE_STATE</td>
<td>android.telephony.TelephonyManager.getDeviceId()</td>
<td>Activity.OnResume</td>
</tr>
<tr>
<td></td>
<td>ACCESS_WIFI_STATE</td>
<td>android.net.wifi.WifiManager.getConnectionInfo()</td>
<td>Service.OnStart</td>
</tr>
<tr>
<td></td>
<td>ACCESS_COARSE_LOCATION</td>
<td>android.location.LocationManager.getLastKnownLocation()</td>
<td>User events</td>
</tr>
<tr>
<td>OFG</td>
<td>SEND_SMS</td>
<td>android.telephony.SmsManager.sendTextMessage()</td>
<td>Activity.Run</td>
</tr>
<tr>
<td></td>
<td>INTERNET</td>
<td>android.webkit.WebView.&lt;init&gt;()</td>
<td>Activity.OnCreate</td>
</tr>
<tr>
<td></td>
<td>READ_PHONE_STATE</td>
<td>android.telephony.TelephonyManager.getLine1Number()</td>
<td>User events</td>
</tr>
<tr>
<td></td>
<td>CAMERA</td>
<td>android.telephony.TelephonyManager.getDeviceId()</td>
<td>User events</td>
</tr>
<tr>
<td>AU</td>
<td>WAKE_LOCK</td>
<td>android.os.PowerManager$WakeLock.acquire()</td>
<td>Activity.OnStart</td>
</tr>
<tr>
<td></td>
<td>INTERNET</td>
<td>android.location.LocationManager.getLastKnownLocation()</td>
<td>Service.OnStart</td>
</tr>
<tr>
<td></td>
<td>ACCESS_COARSE_LOCATION</td>
<td>android.net.wifi.WifiManager.getConnectionInfo()</td>
<td>User events</td>
</tr>
<tr>
<td>MSA</td>
<td>READ_PHONE_STATE</td>
<td>java.lang.Runtime.exec()</td>
<td>Activity.OnCreate</td>
</tr>
<tr>
<td></td>
<td>INTERNET</td>
<td>android.telephony.TelephonyManager.getDeviceId()</td>
<td>Activity.OnStart</td>
</tr>
<tr>
<td></td>
<td>READ_LOGS</td>
<td>java.net.URL.openConnection()</td>
<td>Service.OnStart</td>
</tr>
<tr>
<td></td>
<td>ACCESS_FINE_LOCATION</td>
<td>android.telephony.TelephonyManager.getCellLocation()</td>
<td>User events</td>
</tr>
</tbody>
</table>
5.4.4.1 Malicious Behavior and Permission Requests

We identified 51 Android malware/spyware families and categorized their malicious behavior into four categories:

- Collect and Send (C&S). Malicious applications collect sensitive information (i.e., device ID, IMEI/IMSI, device model, geographic location, personal SMS etc.) from the phone and then send the (encrypted) information to a remote server. The information is normally sent through SMS or Internet. 47% of the malicious applications exhibit this behavior.

- Obtain a Financial Gain (OFG). One of the goals of malicious application developers is to obtain a financial gain. This is achieved in three ways based on observed behavior. Some applications send SMS messages to predetermined or fetched premium rate numbers to charge unwitting users. Some applications silently issue multiple HTTP requests to promote a specific website. There are also some applications that register the phone to a charged service by sending the phone number in a SMS message to the service. 57% of the malicious applications exhibit this behavior.

- Annoy Users (AU). Some applications affect smartphone users by performing annoying operations without user’s awareness. These actions include displaying advertisements, consuming battery, modifying the configuration of the phone (e.g. changing the wallpaper, etc). 6% of the malicious applications exhibit this behavior.

- Monitor Smartphone Activities (MSA). Some applications monitor the victim smartphone’s activities such as content in sdcard storage, network activities, etc. 11% of the malicious applications exhibit this behavior.

This categorization is by no means a comprehensive one, but it does cover a large part of the malicious behavior observed in Android applications [83]. Our analysis shows that these malicious applications request 6.5 Android permissions on average and 64 unique permissions in total.

\[3\text{some malicious applications exhibit behavior that are in more than one category}\]
### Table 5.2. Analysis of the Most Representative Purposes of Permission Use in Malicious Applications

<table>
<thead>
<tr>
<th>Behavior</th>
<th>Correlation</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>C&amp;S</td>
<td>android.telephony.TelephonyManager.getDeviceId android.telephony.TelephonyManager.getSimSerialNumber android.telephony.TelephonyManager.getSubscriberId android.telephony.TelephonyManager.getCellLocation android.location.LocationManager.getLastKnownLocation java.net.URL.openConnection java.net.URLConnection.setDoOutput</td>
<td>Collect phone information to send to remote server</td>
</tr>
<tr>
<td>OFG</td>
<td>android.telephony.SmsManager.sendTextMessage android.telephony.SmsManager.sendTextMessage android.telephony.TelephonyManager.getLine1Number java.net.URL.openConnection java.net.URLConnection.setDoOutput</td>
<td>Send SMS messages to pre-defined number Send phone number to remote server</td>
</tr>
<tr>
<td>AU</td>
<td>android.os.PowerManager.newWakeLock android.os.PowerManager$WakeLock.acquire (android.os.PowerManager$WakeLock.release)</td>
<td>Keep the device on to consume battery</td>
</tr>
<tr>
<td>MSA</td>
<td>java.lang.Runtime.exec ... java.lang.Runtime.exec</td>
<td>Monitor system activity</td>
</tr>
</tbody>
</table>

### 5.4.4.2 Characterizing Permission Use in Malicious Applications

Permlyzer successfully analyzed 310 (out of 325) malicious applications. There are 15 applications that could not be executed or installed. Ten of these applications request root privilege, which we do not grant automatically due to security concerns. The other five applications contain “wrong certificate” errors in the installation packages. These errors are most likely caused by improper signing, which often occurs when repackaging an existing malicious application.

Table 5.1 summarizes the analysis results. We will interpret the characteristics discovered by Permlyzer in the form of findings from four aspects: 1) checked permissions; 2) locations of the permission use; 3) causes of permission use; 4) purposes of permission use.

1) Checked Permissions:

**Finding 1:** Permlyzer’s results show that permissions that are checked in runtime indicate the behavior observed in the malicious applications. For instance, for applications that exhibiting “Collect and Send” behavior, permissions such as “INTERNET” and “ACCESS_WIFI_STATE” are frequently checked because these

---

4 the causes listed in Table 5.1 are summarized from multiple instances for the sake of clarity. Therefore, we use the description (e.g., `element.OnClick`) to represent a click event on all UI elements.
permissions are required to send out the sensitive information via local network proxy. “READ_PHONE_STATE” permission is also checked by most of the applications in this behavior category because this permission is necessary to obtain identification information of the phone (e.g., device ID). For malicious applications aiming at obtaining a financial gain, permission such as “SEND_SMS” is the topmost checked permission because it is essential for sending SMS to premium rate numbers, which is the most commonly observed behavior in this category. Another permission that is frequently checked by applications that try to obtain a financial gain is “READ_PHONE_STATE”. This permission is used to obtain the phone number so that the applications can register the phone to a charged service by sending the phone number to the service. For malicious applications that monitor the activities of the phone, two unique permissions are frequently checked are “READ_LOGS” and “ACCESS_FINE_LOCATION”. We noticed that “READ_LOGS” is not checked by applications in other categories because those applications do not access low-level system log files (e.g., activity manager state).

2) Locations of Permission Use:

Finding 2: Permlyzer’s results show that 51% of the instances of permission use that are related to malicious behavior are located in the main activity (i.e., the default entry activity of an application) of malicious applications. This indicates a tendency among the developers of malicious applications. We believe that this tendency is caused by the fact that the methods in the main activity are more likely to be executed than the methods in other activities because they do not need further activity navigation by users.

Table 5.3. Cause of Permissions in Malicious Applications

<table>
<thead>
<tr>
<th>Cause</th>
<th>Behavior Category</th>
<th>C&amp;S</th>
<th>OFG</th>
<th>AU</th>
<th>MSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Event</td>
<td></td>
<td>3.8%</td>
<td>28.2%</td>
<td>4.6%</td>
<td>0%</td>
</tr>
<tr>
<td>Activity</td>
<td></td>
<td>86.7%</td>
<td>71.7%</td>
<td>83.9%</td>
<td>27.1%</td>
</tr>
<tr>
<td>Service</td>
<td></td>
<td>9.3%</td>
<td>0%</td>
<td>11.3%</td>
<td>72.9%</td>
</tr>
</tbody>
</table>

3) Causes of Permission Use: Table 5.3 further summarizes the causes of permission use in malicious applications.

Finding 3: Permlyzer’s results show that over 90% of the permission uses in malicious applications are caused by the start of application components (i.e.,
activities and services), whereas less than 10% of the permission uses are caused by user events (e.g., touch, click, etc). Combining with previous finding that the default entrance activity (i.e., main activity) is the location where most malicious behavior is coded, it is clear that permission checks related to malicious behavior are mostly triggered without user interaction. Besides, Android applications can self start. This characteristic in the permission use of malicious applications suggests that malicious application requires minimal, if any, user-interactions in order to perform the behavior. This finding stresses the importance of a user’s understanding about an application’s permission use at the install-time.

**Finding 4:** Permlyzer’s results also show that in applications that exhibit “Collect and Send”, “Obtain a Financial Gain”, “Annoy Users” behavior, over 80% of the instances of permission use are caused by the start of activity components, whereas in applications that exhibit MSA behavior, over 70% of the instances of permission uses are caused by the start of service components. Since service components run in the background to perform long-running operations, tasks such as monitoring smartphone’s activity suit better in the service components. This is another characteristic of the permission use in malicious applications.

4) **Purposes of Permission Use:** Permlyzer presents the purpose of a permission use instance from two aspects: the functionality of the API call that triggers the permission check and the correlations with other instances. Permlyzer discovered 537 correlations that involve 2703 instances (out of 2775). Table 5.2 lists the most representative correlation among permission use in each category in terms of a sequence of API calls. It also lists the purpose of each correlation.

**Finding 5:** The results from Permlyzer indicate that correlated permission use can better expose its purpose. For instance, in “Collect and Send” category, the most representative correlation is actually among “READ_PHONE_STATE” permission (i.e., triggered by the first 4 API calls), “ACCESS_FINE_LOCATION” permission (i.e., triggered by the 5th API call), and followed by “INTERNET” permission (i.e., triggered by the last two API calls in the sequence). This correlation indicates multiple phone information is first collected and then send to a remote server, which could cause a privacy leak. In “Obtain a financial gain” category, one correlated permission is “READ_PHONE_STATE” followed by “INTERNET”, the purpose of which is to send phone numbers to a remote server and the combined security risk is
Table 5.4. Requested Android Permissions in Each App Category

<table>
<thead>
<tr>
<th>Category</th>
<th>#</th>
<th>%</th>
<th>Category</th>
<th>#</th>
<th>%</th>
<th>Category</th>
<th>#</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Books &amp; Reference</td>
<td>61</td>
<td>91.1%</td>
<td>Business</td>
<td>75</td>
<td>84.1%</td>
<td>Comics</td>
<td>42</td>
<td>87.3%</td>
</tr>
<tr>
<td>Communication</td>
<td>68</td>
<td>90.2%</td>
<td>Education</td>
<td>60</td>
<td>91.7%</td>
<td>Entertainment</td>
<td>65</td>
<td>81.4%</td>
</tr>
<tr>
<td>Finance</td>
<td>65</td>
<td>85.7%</td>
<td>Health &amp; Fitness</td>
<td>63</td>
<td>96.1%</td>
<td>Libraries &amp; Demo</td>
<td>67</td>
<td>87.2%</td>
</tr>
<tr>
<td>Lifestyle</td>
<td>73</td>
<td>84.3%</td>
<td>Media &amp; Video</td>
<td>65</td>
<td>87.4%</td>
<td>Medical</td>
<td>59</td>
<td>80.8%</td>
</tr>
<tr>
<td>Music &amp; Audio</td>
<td>68</td>
<td>86.7%</td>
<td>News &amp; Magazines</td>
<td>41</td>
<td>94.9%</td>
<td>Personalization</td>
<td>75</td>
<td>83.6%</td>
</tr>
<tr>
<td>Photography</td>
<td>67</td>
<td>89.1%</td>
<td>Productivity</td>
<td>81</td>
<td>79.4%</td>
<td>Shopping</td>
<td>48</td>
<td>91.3%</td>
</tr>
<tr>
<td>Social</td>
<td>66</td>
<td>80.0%</td>
<td>Sports</td>
<td>73</td>
<td>78.3%</td>
<td>Tools</td>
<td>91</td>
<td>75.5%</td>
</tr>
<tr>
<td>Transportation</td>
<td>53</td>
<td>83.9%</td>
<td>Travel &amp; Local</td>
<td>62</td>
<td>79.2%</td>
<td>Weather</td>
<td>45</td>
<td>85.6%</td>
</tr>
<tr>
<td>Game</td>
<td>71</td>
<td>77.2%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*a Number of unique Android permissions requested by the apps in a category
b Percentage of Android permissions in the total requested permissions

Finding 6: In addition to correlations among the use of different permissions, the results also suggest that the correlations among multiple use instances of the same permission can reveal the purpose as well. In “Monitor smartphone activity” category, “READ_LOGS” permission is checked multiple times (i.e., multiple “java.lang.Runtime.exec()”) and these checks are correlated with each other because this permission is used to access low-level system log files periodically.

Using Permlyzer’s results, we identified 19 combinations of permission requests as potential indicators of malicious behavior and 44 correlations of call stacks that characterize the malicious behavior.

5.4.5 Permission Use in Free Applications

In this section, we evaluate the effectiveness of Permlyzer in analyzing permission use in free applications by applying it on our large dataset of free Android applications. We evaluated the scalability of Permlyzer leveraging the large number of free applications. Meanwhile, we also discuss several aggregated results obtained by Permlyzer to demonstrate the capability of Permlyzer in helping us better understand the permission uses in applications.

5.4.5.1 Android Permissions Requests

Table 5.4 summarizes the requested Android permissions by applications in each category. The collected free applications request 111 unique and legitimate Android permissions in total and three on average. Among the total requested per-
<table>
<thead>
<tr>
<th>Permission</th>
<th>Top Locations</th>
<th>Top Causes</th>
<th>Top Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERNET</td>
<td>java.net.Socket. &lt;init&gt;</td>
<td>Activity.onCreate</td>
<td>Client/Server communication</td>
</tr>
<tr>
<td></td>
<td>android.webkit.WebView. &lt;init&gt;</td>
<td>Activity.onStart</td>
<td></td>
</tr>
<tr>
<td>READ_PHONE_STATE</td>
<td>android.telephony.TelephonyManager.getDeviceId</td>
<td>Activity.onStart</td>
<td>Collect and send device ID</td>
</tr>
<tr>
<td></td>
<td>android.telephony.TelephonyManager.getLine1Number</td>
<td>Activity.onResume</td>
<td></td>
</tr>
<tr>
<td>ACCESS_COARSE_LOCATION</td>
<td>android.location.LocationManager.getBestProvider</td>
<td>Activity.onStart</td>
<td>Obtain location info.</td>
</tr>
<tr>
<td></td>
<td>android.location.LocationManager.getLastKnownLocation</td>
<td>Activity.onResume</td>
<td></td>
</tr>
<tr>
<td>ACCESS_WIFI_STATE</td>
<td>android.net.wifi.WifiManager.getWifiState</td>
<td>Component.Run</td>
<td>Connect to remote server</td>
</tr>
<tr>
<td></td>
<td>android.net.wifi.WifiManager.isWifiEnabled</td>
<td>Activity.onCreate</td>
<td></td>
</tr>
<tr>
<td>ACCESS_FINE_LOCATION</td>
<td>android.location.LocationManager.isProviderEnabled</td>
<td>Activity.onResume</td>
<td>Obtain location info.</td>
</tr>
<tr>
<td></td>
<td>android.location.LocationManager.getLastKnownLocation</td>
<td>Activity.onStart</td>
<td></td>
</tr>
<tr>
<td>WAKE_LOCK</td>
<td>android.os.PowerManager$WakeLock.release</td>
<td>Activity.onPause</td>
<td>Keep device on</td>
</tr>
<tr>
<td></td>
<td>android.os.PowerManager$WakeLock.acquire</td>
<td>Activity.onResume</td>
<td></td>
</tr>
<tr>
<td>READ_CONTACTS</td>
<td>android.content.ContentResolver.query</td>
<td>Component.onCreate</td>
<td>Query user data</td>
</tr>
<tr>
<td></td>
<td>android.content.ContentResolver.openFileDescriptor</td>
<td>User events</td>
<td></td>
</tr>
<tr>
<td>GET_ACCOUNTS</td>
<td>android.accounts.AccountManager.getAccountsByType</td>
<td>User events</td>
<td>Purchase and billing</td>
</tr>
<tr>
<td></td>
<td>android.accounts.AccountManager.getAccounts</td>
<td>Activity.onCreate</td>
<td></td>
</tr>
<tr>
<td>READ_LOGS</td>
<td>java.lang.Runtime.exec</td>
<td>Activity.onCreate</td>
<td>Access low-level</td>
</tr>
<tr>
<td></td>
<td>java.lang.Runtime.exec</td>
<td>Activity.onResume</td>
<td></td>
</tr>
<tr>
<td>CAMERA</td>
<td>android.hardware.Camera.open</td>
<td>User events</td>
<td>System logs start camera</td>
</tr>
<tr>
<td></td>
<td>android.hardware.Camera.native_setup</td>
<td>User events</td>
<td></td>
</tr>
</tbody>
</table>
missions, 91.7% are Android permissions. Meanwhile, there are 12 requested permissions that do not exist. This is either because of errors in the development (e.g., "WRITE" instead of "WRITE-*", "ACCESS_COURSE_LOCATION" instead of "ACCESS_COARSE_LOCATION") or because the requested permissions are no longer supported (e.g., "ACCESS_LOCATION", "ACCESS_GPS").

Table 5.4 shows that most of the requested permissions are Android permissions. We focus our analysis on the Android permissions, but we note that Permlyzer can also be applied to application-defined permissions.

### 5.4.5.2 Permission Use in Free Android Applications

Table 5.5 summarizes the analysis of location, cause and purpose of top checked permissions in free applications. Given the large number of applications in our dataset, we only discuss the most representative statistic results in this section.

**Finding 1: Third Party Libraries** From Permlyzer’s results, we found that many permission checks are actually triggered by third-party libraries in free applications. For instance, we noticed that 23% of checked “INTERNET” permissions are actually caused by included third-party libraries (e.g., Ad networks, etc.). “INTERNET” is the most checked permission according to our results. In fact, 85% of all the checked Android permissions are “INTERNET”. In general, the popularity of “INTERNET” is because of the fact that most Android applications are developed following the client/server model. Therefore, the “INTERNET” permission is intensively checked to grant the data communication between applications and their servers. However, as we have suggested, a great deal of the communication actually happens with third-party servers. We further analyzed the purposes of observed third-party libraries and we summarized these libraries into four categories: advertisement networks, mobile analytic services, common data provider, other (e.g., search portal, etc.) Table 5.6 lists the percentage of each category of purpose. We noticed that most of the third-party libs that trigger “INTERNET” permission checks are advertisement network.

**Finding 2: Collection of Identification Information** Permlyzer’s results show that 82% of the checked “READ_PHONE_STATE” permissions are used to obtain device ID (e.g., IMEI) of the smartphone and the rest of the permission checks are used to obtain other identification information (e.g., phone number,
serial number of SIM, IMSI) as well. The results also show that most of these permission checks are caused by the start and/or resume of activities, only a small part (i.e., 2%) are caused by user events. This means that most of the access to the identification information happens without user’s interaction and therefore it is most likely that this piece of sensitive information is obtained without user’s knowledge.

Besides, the results also demonstrate that almost half (49%) of the permission checks of “READ_PHONE_STATE” are followed by the checks of “INTERNET” permission, which are used to send the obtained identification information to remote servers. Further investigations shows that 52% of these remote servers are actually related to third-party libraries. As shown in Table 5.6, 93.8% of the third-party libraries that cause the access and send out the identification information are advertisement networks. We did not find any claim of collecting identification information from smartphones based on the description of these advertisement networks, but we believe one purpose of collecting such information is to deliver more customized advertisements to the smartphone users. In fact, there also exists a financial incentive for the application developer to not only integrate with the third party libraries but also provide information to increase the click-through rate of the advertisements shown as part of the application’s UI. From user’s perspective, this finding however is less than desirable, since this leads to the leak of identification information. Therefore we believe the analysis of Permlyzer can provide users with more understanding of the privacy risks to grant an Android permission.

**Finding 3: Collection of Location Information** Permlyzer’s results show that 4% of all the permissions checks are related to location access permissions (i.e. “ACCESS_FINE_LOCATION” and “ACCESS_COARSE_LOCATION”). Moreover, most of the API calls that trigger the checks of these permissions are to obtain the location information of the smartphone. The analysis of Permlyzer

<table>
<thead>
<tr>
<th>Permission</th>
<th>Ad Network</th>
<th>Analytic</th>
<th>Data</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERNET</td>
<td>81.3%</td>
<td>10.0%</td>
<td>8.6%</td>
<td>&lt;1%</td>
</tr>
<tr>
<td>READ_PHONE_STATE</td>
<td>93.8%</td>
<td>3.5%</td>
<td>0%</td>
<td>2.6%</td>
</tr>
<tr>
<td>ACCESS_FINE_LOCATION</td>
<td>28.2%</td>
<td>68.1%</td>
<td>0%</td>
<td>3.7%</td>
</tr>
</tbody>
</table>

Table 5.6. Permissions Use Caused by Third-party Libraries
shows that the primary cause of these permission checks is the start and/or resume of activity components, which causes 59% of the permission checks. We also notice that 33% of the causes are related to third-party libraries. As shown in Table 5.6, 68.1% of the third-party libraries belong to analytic services. We believe this is because location information can help application developers to monitor the geographical distribution of the downloads and use of their applications. In fact, location information is required by all the analytic services that we have identified. However, the collection of smartphone’s location information may also raise privacy concern to the users, especially when the users are unaware of most of such collections, as the primary cause of permission checks suggests.

5.4.5.3 Security/Privacy Risks in Free Applications’ Permission Use

To evaluate the effectiveness of Permlyzer in identifying the potential security and/or privacy risk in free applications. We study the similarity of permission use in top 100 free applications from each category with the most representative permission use in Collect and Send malicious applications. More specifically, we compared the correlation of permission use in free applications with the representative correlation of permission use in Collect and Send malicious applications. The comparison shows that 4.2% of the free applications exhibit similar behavior to the collect and send malicious applications. Note that this does not indicate that these free applications are malicious because some of these application’s advertised functionalities include collect and send information. However, given the analysis results of Permlyzer, a user can better understand the potential risk of granting requested permissions based on their own security preference.

5.5 Summary

In this chapter, we proposed and developed a framework to automatically analyze the use of permissions in Android applications. Permlyzer can perform accurate analysis of the permission use in Android applications. Unlike existing approaches, Permlyzer uses call stack-based analysis scheme to provide fine-grained information on the use of the permissions from various aspects including locations, causes and purposes. By leveraging the automatic functionality exploration and an efficient
call-stack construction algorithm, Permlyzer can be applied to the analysis of the use of permissions in a large number of applications. Our evaluation demonstrates that Permlyzer can automatically identify characteristics in the use of permissions among over 110,000 free applications and 51 malware/spyware application families. Moreover, we believe Permlyzer can help not only users to make informed decisions about granting requested permissions, but also application vendors to vet the permission requests of a large number of applications.
Chapter 6

Conclusion

This dissertation focuses on three most representative and severe security and privacy threats posed against Web and third party applications, namely obfuscated malicious JavaScript code, worm propagations in OSN and use of permissions in Android applications.

For each threat, a defending/detection approach is proposed, implemented and evaluated. More specifically, a most static detection scheme that captures the characteristics of malicious obfuscation techniques is developed to detect the obfuscated malicious JavaScript code. A decoy friend-based surveillance network is proposed to provide early warning on worm propagations in OSN. An automated permission analysis framework is designed to analyze the detailed use of permissions in Android applications.

Each approach is evaluated using a large set of real-world samples/data so that the practicability is examined. The evaluation results demonstrate that these mechanisms proposed in this dissertation are effective and efficient in protecting the security and privacy of Web and third-party applications. Moreover, we believe that these mechanisms can also shed light on defending other applications that share similar characteristics as the applications discussed in this dissertation.
Appendix A

A.1 Evading Effectiveness

![Graph showing detection rate of 20 anti-virus software on samples obfuscated by data obfuscation.]

Figure A.1. The Detection Rate of 20 Anti-Virus Software on Samples Obfuscated by Data Obfuscation

A.2 Algorithms
Algorithm 3 Substring Identification

Input: Basic strings set $S = s_1, ..., s_p$ in tree $T_T$, Argument string $arg$

Output: Subset of basic strings $C$

1: $C \leftarrow \emptyset$
2: $T \leftarrow S_T$
3: $i \leftarrow 0$
4: searchtree($T, i$)
5: while $i < \text{length}(arg)$ do
6:     search level 1 of tree $T$ for key $arg[i]$
7:     if key $arg[i]$ is found on node $j$ then
8:         if node $j$ is leaf then
9:             $C \leftarrow s$ ($s \in S$ $s := \text{path from root to } j$)
10:        else
11:            $T_j := \text{subtree from node } j$
12:            searchtree($T_j, i \leftarrow i + 1$)
13:        end if
14:     end if
15:     $i \leftarrow i + 1$
16: end while

Algorithm 4 Metric Calculation

Input: Substrings set $S = s_1, ..., s_q$, argument string $arg$

Output: max percentage $max_p$

1: $max_p \leftarrow 0$
2: while $S \neq \emptyset$ do
3:     find the longest substring $s_i$ in $S$
4:     $max_p \leftarrow max_p + \text{length}(s_i)/\text{length}(arg)$
5:     $S \leftarrow S - \{s_i\}$
6: end while
Appendix B

B.1 D-Gen and R-Eval Functions hooked in the implementation of JStill

Table B.1. D-Gen and R-Eval Function hooked by JStill

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>document.write</code></td>
<td>dynamic code generation</td>
</tr>
<tr>
<td><code>document.writeln</code></td>
<td>dynamic code generation</td>
</tr>
<tr>
<td><code>window.setTimeout</code></td>
<td>evaluate the 1st argument as code</td>
</tr>
<tr>
<td><code>window.setInterval</code></td>
<td>evaluate the 1st argument as code</td>
</tr>
<tr>
<td><code>eval</code></td>
<td>evaluate the argument as code</td>
</tr>
</tbody>
</table>

B.2 JavaScript Obfuscation Tools
<table>
<thead>
<tr>
<th>Tools</th>
<th>Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thicket</td>
<td>D, A, S</td>
</tr>
<tr>
<td>Jasob</td>
<td>D</td>
</tr>
<tr>
<td>JS Obfuscator</td>
<td>D, A</td>
</tr>
<tr>
<td>Stunnix</td>
<td>D, A</td>
</tr>
<tr>
<td>JCE Pro</td>
<td>D, A</td>
</tr>
<tr>
<td>ScrEnc</td>
<td>D, A, C</td>
</tr>
<tr>
<td>Shane</td>
<td>D, A</td>
</tr>
<tr>
<td>Dean</td>
<td>D, A</td>
</tr>
<tr>
<td>Jammer</td>
<td>D</td>
</tr>
<tr>
<td>JSCrunch Pro</td>
<td>D</td>
</tr>
</tbody>
</table>

D: Data Obfuscation  
A: ASCII/Unicode/Hexadecimal encoding  
C: Customized Encoding Functions  
S: Standard Encryption and Decryption  
*Encoding/encryption-based obfuscation includes A, C, S
Bibliography


Proceedings of the 7th symposium on Operating systems design and implementation, OSDI ’06, USENIX Association, Berkeley, CA, USA, pp. 61–74.


Wei Xu is a PhD candidate in the Department of Computer Science and Engineering at Pennsylvania State University since 2008. He received the B.S. degree and M.S. degree in Electrical Engineering from Tsinghua University, Beijing, China in 2005 and 2007 respectively.

Publications and Presentations


