ANALYZING AND DETECTING MALICIOUS ACTIVITIES IN EMERGING COMMUNICATION PLATFORMS

A Dissertation

by

CHAO YANG

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Guofei Gu
Committee Members,	James Caverlee
	Anxiao Jiang
	Narasimha Reddy
Head of Department,	Nancy Amato

August 2014

Major Subject: Computer Engineering

ABSTRACT

Benefiting from innovatory techniques, two communication platforms (online social networking (OSN) platforms and smartphone platforms) have emerged and been widely used in the last few years. However, cybercriminals have also utilized these two emerging platforms to launch malicious activities such as sending spam, spreading malware, hosting botnet command and control (C&C) channels, and performing other illicit activities. All these malicious activities may cause significant economic loss to our society and even threaten national security. Thus, great efforts are indeed needed to mitigate malicious activities on these advanced communication platforms.

The goal of this research is to make a deep analysis of malicious activities on OSN and smartphone platforms, and to develop effective and efficient defense approaches against those malicious activities. Firstly, this dissertation performs an empirical analysis of the cyber criminal ecosystem on a large-scale online social networking website space. Secondly, through reverse engineering OSN spammers' tastes (their preferred targets to spam), this dissertation provides guidelines for building more effective social honeypots on the online social networking platforms, and generates new insights to defend against OSN spammers. Thirdly, this dissertation shows a comprehensive empirical study on analyzing the market-level and network-level behaviors of the Android malware ecosystem. Lastly, by grouping the common program logic among malware families, this dissertation designs an effective system to automatically detect Android malware.

ACKNOWLEDGEMENTS

First of all, I would like to express my deep gratitude to my advisor, Prof. Guofei Gu, for his continuous support and guidance throughout my PhD study. His advice helped me all the time in my research. Without his inspiration, this work would not have been possible. I would also like to thank the rest of my thesis committee members, Prof. James Caverlee, Prof. Anxiao Jiang, and Prof. Narasimha Reddy, for their comments and encouragements.

Furthermore, I am very grateful to my collaborators and friends, Dr. Phil Porras, Dr. Vinod Yegneswaran, Robert Harkreader, Jialong Zhang and Zhaoyan Xu. It has been a fruitful and fun experience working with them over the past few years.

Last but not the least, I take this opportunity to thank my parents. Through the good times and bad, they always stand behind me and devote their love. I cannot be more thankful to them.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

TABLE Page

1. INTRODUCTION

In the recent years, with the innovation of Online Social Networking (OSN) platforms (e.g., Twitter and Facebook) and Smartphone platforms (e.g., Android), many people have changed their lifestyle, from posting their recent experiences, finding out what friends are up to, and keeping track of the hottest trends, to viewing interesting photos or videos, and playing games with friends.

However, cyber-criminals have also utilized OSN platforms and smartphone platforms to launch malicious activities such as sending spam, spreading malware, hosting botnet command and control (C&C) channels, and performing other illicit activities. All these malicious activities have caused significant economic loss to our society, and even threaten national security. Thus, great efforts are indeed needed to mitigate malicious activities on these advanced communication platforms. The goal of this research is to make a deep analysis of malicious activities on these two types of emerging communication platforms, and to further develop effective and efficient defense insights against those malicious activities.

In this chapter, we first introduce the malicious activities in these two types of the platforms, and then outline the research challenges for the analysis and detection of those malicious activities. Next, we show the common characteristics of these two types of platforms that are essentially utilized by cyber-criminals to launch malicious activities. We further provide an overview of our solution: a deep understanding of spammers' social networks, and a comprehensive measurement of spammers' strategies of selecting spamming targets, and two Android malware detection approaches (one is built based on the understanding of Android malware ecosystem; the other one is built by disassembling Android apps, and analyzing the programming pro-

cedure shared by known malware.). Finally, we present the contributions of the dissertation.

1.1 Malicious Activities in the OSN and Smartphone Platforms

Traditionally, cyber-criminals typically require great efforts to build their own platforms to induce victims to visit their spam/malicious websites or further download their malware. Generally, the effectiveness of cyber-criminals to successfully induce victims is highly restricted by two major factors: (1) the number of users that the links to the cyber-criminals' spam/malicious websites can be exposed to; (2) click-through rate, which is the probability that the users would click links to visit spam/malicious links (or further download malware). Cyber-criminals typically spread their malicious links in popular forums to induce victims. Since most of the forums are interested by particular groups of people, and most people only read recent news typically shown in the top a few pages, the effectiveness of this approach to induce victims is highly limited by the number of users that can access the malicious links.

However, the emergency of OSN and smartphone platforms eased the process of cyber-criminals to induce victims. Cyber-criminals have already utilized such communication platforms to launch multiple types of malicious activities. Table 1.1 shows several selective attacks that historically are launched in the OSN platforms.

As seen from this table, in the early stage of the OSN platforms, due to the week security vetting process, cyber-criminals can easily spread spam [81] or phishing attacks [56] by posting unsolicited messages on their faked accounts' profile, or sending unsolicited messages to other users. As a very stealthy channel, cyber-criminals have also utilized OSN platforms to host C&C commands to coordinate with their controlled bots. Later, cyber-criminals evolved to exploit the security vulnerabilities

Attack Type	Year	Attack Details
Spam	2009	Twitter spam invades trending topics [81]
Phishing	2009	A new phishing scam spreads through direct messages [56]
Hosting Botnet	2009	Twitter-based Botnet Command Channel [59]
Clickjacking	2010	Facebook clickjacking attack spreads through Facebook likes [120]
Cross-site Scripting	2010	Twitter onMouseOver security flaws [28]
Distributing Malware	2011	New Koobface malware spreads on Facebook [17]
Hacking Accounts	2010	Twitter phishing hack hits BBC, Guardian and cabinet minister [6]

Table 1.1: Selective attacks in the online social networking platforms.

of OSN platforms to launch more complex attacks (e.g., clickjacking attacks [120] and cross-site scripting attacks [28]). In addition, once some famous OSN accounts owning thousands of OSN friends (e.g., followers in Twitter or friends in Facebook) are hacked, cyber criminals utilized these accounts to spread their malicious content very efficiently, and made great cost for victims [6].

Similarly, cyber-criminals have also unleashed a great number of smartphone malware to achieve multiple malicious goals (e.g., hijacking phones [72], privacy leaking [20] and money stealing [98, 119]). According to a survey in China, until March 2012, over 210 thousands smartphones have been injected malicious code that can steal victims' money by stealthily making phone calls or sending SMS messages to premium-rate numbers. It makes victims to lose over 10 million dollars per year [98]. Meanwhile, the number of malicious smartphone apps also increases quickly during these years. According to a Mobile Report, the number of malicious Android apps received by F-Secure grew from 139 in the first quarter (Q1) of 201, to 3,063 and over 10,000 during the same Q1 period of 2012 and 2013, respectively. F-Secure also receives 153 new Android malware families in the first quarter (Q1) of 2013, which increases to 252 in the third quarter (Q3) of 2013. Trend Micro identified about 5,000 malicious Android apps in the first quarter of 2012, a number that rose to

20,000 by the end of June. Google has realized the seriousness of the malware threat and implemented the Google Bouncer for its GooglePlay marketplace. But, Google-Play remains inaccessible in countries like China, where users have no choice but to rely on dubious third-party marketplaces. Furthermore, clever malware could still fingerprint and evade the analysis of Bouncer [125]. Trend Micro also reports that it found 17 malicious apps in GooglePlay (Google's official Android marketplace), and those apps were downloaded more than 700,000 times before Google removed them. In addition, different from spreading desktop malware, malware authors can utilize Android markets to spread Android malware more efficiently. According to a recent report in 2013, during a security check of over 90,000 Android apps from 24 Chinese third-party Android markets, 860 types of Android malware were discovered and had been downloaded over 8.5 million times [99]. One type of the Android malware, named Skullkey, was inserted to over 6,000 Android apps and spread widely in those third-party markets. This malware could even bypass the detection of existing commercial anti-virus tools and achieve multiple malicious goals (e.g.,steal sensitive information and send SMS to premium-rate numbers).

According to the previous discussions, we can clearly see that we need to take great efforts to design effective approaches to defend against those malicious activities launched in such emerging communication platforms. Thus, the desire of understanding and further detecting such new types of malicious activities forms the key motivation of this dissertation.

1.2 Common Charactersitics between OSN and Smartphone Platforms

The reason why the emergency of the OSN and smartphone platforms facilities cyber-criminals to launch attacks is mainly due to the following four major common characteristics:

- *User-based.* Both of these two types of platforms rely on users' contribution (e.g., Tweets in Twitter, Walls in Facebook, and Android apps in GooglePlay). Restricted by limited policies, users have a great freedom to submit any information they like to share. Also, in the current design of these two types of platforms' architecture, the content submit by the users can be pushed to (or recommended to) other users. Thus, utilizing such platforms, cyber-criminals can stealthily submit their malicious content, and expose them to potential victims more efficiently.

- *Global Centralized.* Both of these two types of platforms have been widely used by the people from all of the world. By Jan. 1st of 2014, Twitter has over 645 million active registered users and over 58 million tweets per day. By Jan. 14th of 2014, smartphones have taken over 90% of global market shares (Android takes up around 51.7%). By July 2013, over 1 million Android apps are available in GooglePlay and over 50 billion times of apps have been downloaded by users. Such a global centralized architecture essentially can be utilized by cyber-criminals to expose their malicious content to more potential victims at a very fast speed.

- *Interactive.* Unlike traditional chatting rooms and forums, users can interact with their friends (e.g., sharing interesting news, personal photos, and playing games) more frequently and easily in these emerging communication platforms. In the real-world, people tend to trust the information sent from their friends, or trust the items selected by a large number of people. With the same habitat, users are typically less aware of potential security risks in the messages (e.g., Tweets and Walls) post/sent from their OSN friends (or famous OSN users), and also more likely to download those hot apps that have been down-

loaded a lot. Thus, once cyber-criminals successfully compromise such trust to post malicious content (e.g., compromise famous OSN accounts, or fraudulently increase the downloading numbers of smartphone apps), these malicious content can be spread at a very fast speed. In fact, many benign users' OSN accounts are compromised due to their less awareness of the security risks in the messages sent from their friends or their favorite stars.

- *Week Security Vetting Process.* In the early stage of these two types of platforms, organizers typically pay more attentions to encouraging users to contribute more content to the system, than the quality (potential security risks) of those content. Once these platforms become larger and more popular, the great amount of information post by users makes it extremely changeling for the organizers to vet potential security risks or attacks in every piece of the information within a short time period.

1.3 Research Challenges of Detecting Malicious Activities

The first challenge of detecting malicious activities in these two types of platforms is that we lack basic insights of the strategies that are utilized by cyber-criminals to launch malicious activities. More specifically, in terms of the OSN platform, cyber-criminals typically require to register a corpus of fake (spam) accounts to spread malicious activities. Although existing studies have been made to detect sybil nodes. These studies rely on the assumption that it is difficult for sybil nodes to mix with benign nodes. However, is this assumption held in the real-world OSN platforms? If not, how do spam accounts mix into the real-world OSN platforms? How are spam accounts correlated with each other? How do spam accounts find their spamming targets? How do malware authors utilize app markets to spread smartphone malware? How do malware authors build networking infrastructure to

communicate with their malware? The deep understanding of these questions are essentially very important to facilitate to design effective approaches to detect those malicious activities.

The second challenge is that how to design effective and efficient approaches to detect (or to guide the sample of more likely) malicious activities, given the limited time/resource. Given the fact that there are millions of OSN accounts and billions of OSN messages sent per day, it is extremely difficult to analyze every account and every message within a short time of period. Thus, a guided inference to those more likely spam accounts is highly desirable. Also, due to the socialization property of the OSN platform, to design effective defensive approach, we may also need to borrow knowledge from other areas (e.g., graph theory and nature language processing theory). Similarly, in the smartphone platform, not every smartphone app market (especially those third-party markets) has sufficient resource/time/expertise to make a deep security analysis of the all apps that are uploaded to the markets. Even for the official smartphone app market (e.g., GooglePlay), it will be very challenging (or even impossible) to vet every app within a short time period. Thus, an effective and lightweight approach to sample those more likely malicious apps is also desirable. In addition, given the special programming design of smartphone platforms, we also need to deeply analyze the programming procedure in known smartphone malware to design an effective approach to automatically detect smartphone malware.

1.4 Research Goal and Solution Overview

In this section, we present the research goal and the overview of our solutions. As illustrated in Table 1.2, this dissertation aims at providing in-depth analysis of the malicious activities, and further providing defense insights against those malicious activities in the OSN and Smartphone platforms.

	OSN Platform	**Smartphone Platform**
Malicious Activities	Malicious OSN Accounts	Malicious Android Apps
In-depth Analysis	Spammers' Social Network	Market-level and Networking-level Behaviors
Defensive Insights	Social Honeypot	Malware Programming Procedure

Table 1.2: Research overview of the dissertation.

More specifically, to understand the characteristics of the social relationships among malicious OSN accounts, we aim at describing a detailed analysis of OSN spammers' social network (i.e., cyber criminal ecosystem) on Twitter. After understanding the characteristics of OSN spammers' spamming targets, we plan to further provide detection approaches including guidelines of building effective social honeypots to attract spammers and two inference-based algorithms to sample spam accounts. Similarly, we aim at providing a deeper understanding how Android malware authors spread malicious Android apps by analyzing the Android malware ecosystem, and further designing an automatic approach to detect malicious Android apps.

Based on our research goal, Figure 1.1 illustrates a general framework of our solution, including three major phases: collecting data, making in-depth analysis, and generating defensive insights.

In the phase of collecting data, we first build an effective crawler to collect a large-scale of real-world dataset. In this dissertation, we choose Twitter and Android markets as our case studies. Due to the challenging of obtain a perfect ground truth for such large-scale datasets, we require to adopt a practical and relatively accurate policy to identify malicious activities from our collected datasets (i.e., malicious/spam Twitter accounts in Twitter, and malicious Android apps in the Android markets).

In the phase of making in-depth analysis, we mainly analyze the following three aspects of the malicious activities in Twitter: Spammer Behaviors (the social behaviors of individual spam account), Malicious Account Community (the relationships among multiple malicious Twitter accounts), and Strategies of Spreading Twitter spam (the strategies used by spammers to find spam targets). From a similar research viewpoint, we mainly analyze the following three aspects of the malicious activities in the Android platform: Malware Author Behavior (the market behavior of individual malware author), Malicious App Community (the relationships among multiple malicious Android apps), and Infrastructure of Spreading Malicious Apps (the market and networking infrastructure used by malware authors to spread malicious apps).

In the phase of generating defensive insights against Twitter spammers, our solution obtains indications from analyzing Tweet Content (content of the tweets), Insert URL (URLs that are inserted in the tweet), Spammer Behavior (social behavior of spam account), Community Relationships (the relationships among multiple spam accounts), Social Honeypot (fake Twitter accounts that are used to capture spam accounts' contact). Similarly, in the phase of generating defensive insights against Android malware, our solution obtains indications from analyzing Programming Procedure (Android framework APIs and control-flow logics that are commonly used in Android malware), Remote Server (the remote servers that are used to communicate with Android malware), Author Behavior (Android malware authors market behavior), and Community Relationships (the relationships among multiple malicious Android apps).

We integrate our solution into four technical chapters: Twitter Spammer Ecosystem (an in-depth analysis of the ecosystem of Twitter spammers), Reversing Engineering Twitter Spammers (a comprehensive measurement of the strategies used

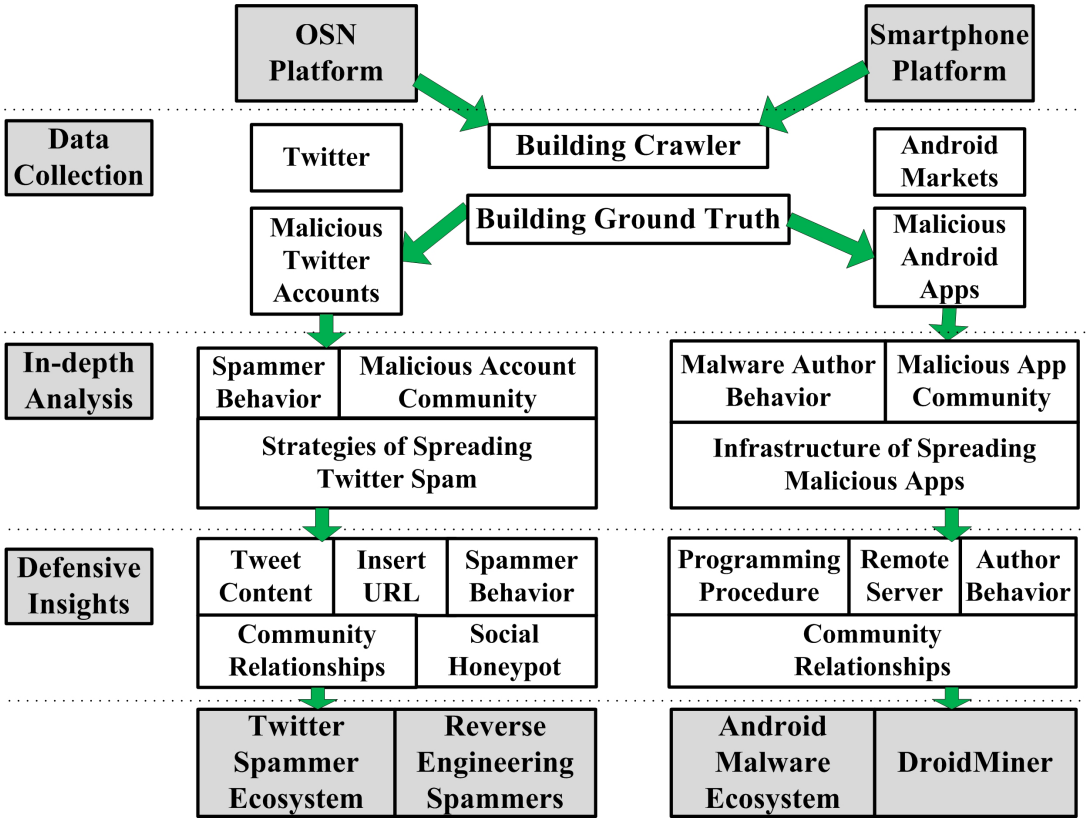


Figure 1.1: The illustration of the framework of our solution.

by Twitter spammers to select spamming targets), Android Malware Ecosystem (an in-depth analysis of the ecosystem of Android malware), and DroidMiner (an automatic Android malware detection system). We next present an overview of each of the four technical chapters with more details.

In Chapter 4, this dissertation empirically analyzes the cyber criminal ecosystem on Twitter, including malicious account community composed of malicious accounts, and malicious supporter community composed of other accounts who have close friendships (following relationships) with malicious accounts. Specifically, this dissertation analyzes inner social relationships in the malicious account community to examine how malicious accounts socially connect with each other. Then, it analyzes

outer social relationships between malicious accounts and their malicious supporters to reveal the characteristics of those accounts who have close friendships with malicious accounts. Through these analyses, this dissertation aims at understanding how malicious accounts survive and mix into the whole Twitter space, and presenting new defense insights to effectively catch more malicious accounts on Twitter.

In Chapter 5, through reverse engineering spammers' tastes (their preferred targets to spam), this dissertation provides guidelines for designing effective social honeypots, and design lightweight and guided strategies to actively sample more likely social spam accounts. To achieve this goal, this dissertation use Twitter as a case study due to its great popularity and publicity. Specifically, to reveal which behaviors tend to incur spammers' contact, we implement 96 "benchmark" Twitter social honeypots with 24 diverse fine-grained social behavior patterns to trap spam accounts. After launching our social honeypots for five months, we successfully garner around 600 spam accounts. Using these data, we analyze spammers' tastes (how spammers find their targets), through comparing the effectiveness of social honeypots with different behavior patterns. Based on these analyses, we design and implement 10 more effective ("advanced") honeypots to trap Twitter spammers. Within the same time period, using those advanced honeypots can trap spammers around 26 times faster than using "traditional" honeypots. To further understand spammers' tastes, we also design an algorithm to extract semantic topic terms, which may highly attract spammers' attentions. In addition, with the concern of limited time/resource, through reverse engineering spammers' strategies of selecting targets, we gain the insights to design two guided approaches to prioritize the *active* sampling of more likely spam accounts from Twittersphere, which is an effective complement to existing *passive* social honeypots.

In Chapter 6, this dissertation empirically performs the first comprehensive mea-

surement study on analyzing the market-level and network-level behaviors of the Android malware ecosystem. Through the analysis, we provide more deep analysis on how Android malware is spread, and generate new defense insights against Android malware. In the phase of analyzing the market-level behaviors, we mainly investigate: (1) whether the location of the market is an effective indication to the quality of Android apps; (2) whether the downloading number has a strong correlation with the quality of Android apps; (3) whether the public Android anti-virus blacklist is effective to stop malware authors from submitting their malicious apps to the markets; (4) whether malicious accounts have specific temporal behavioral patterns to submit malware samples. In the phase of analyzing the network-level behaviors, we investigate: (1) which IP address spaces are mainly used by Android malware; (2) which special networks tend to be used to host remote servers; (3) whether existing IP/domain blacklists are effective to be used to find Android malware; (4) the characteristics of malware communities. Spurred by our analysis, we design an Android malware inference algorithm, to infer more malicious Android apps by starting from a small seed set of known malicious ones.

In Chapter 7, this dissertation presents DroidMiner for discovering and automatically extracting malware modalities. While our efforts are primarily focused on identifying and then characterizing malware behavior, aspects of our methodology are also directly applicable to automated characterization of a broad class of Android application behaviors, including the detection of shared security vulnerabilities. We evaluate DroidMiner using 2,466 malicious apps, identified from a corpus of over 67,000 third-party market Android apps, plus an additional set of over 10,000 official market Android apps. Specifically, this dissertation measure the utility of DroidMiner modalities with respect to three specific use cases: (*i*) malware detection, (*ii*) malware family classification, and (*iii*) malware behavioral characterization. Our

results validate that DroidMiner modalities are useful for classification and capable of isolating a wide range of suspicious behavioral traits embedded within Android applications. Furthermore, the composite of these traits enables a unique means by which Android malware can be identified with a high degree of accuracy.

1.5 Contributions

In this thesis, we make the following main contributions:

- In order to analyze the cyber criminal ecosystem on Twitter, this dissertation finds a few observations about inner- and outer-social relationships among Twitter spammers. We have two main findings: (i) Malicious accounts tend to be socially connected, forming a small-world network; (ii) Compared with malicious leaves, malicious hubs are more inclined to follow malicious accounts. We also find that malicious accounts in some particular malicious campaign tend to have strong semantic and timing coordinations. To analyze how Twitter spammers mix into Twitter space, we propose a new algorithm to extract malicious supporters who have close friendships with malicious accounts. We also investigate the characteristics of three representative categories of malicious supporters. We design a new algorithm to selectively sample and infer more malicious accounts based on a known seed set by analyzing their social relationships and semantic coordinations with other accounts.

- To provide guidelines for designing effective social honeypots, and design a lightweight algorithm to actively sample more likely social spam accounts, this dissertation contributes a set of new defense insights. We present a deep analysis of spammers' tastes: spammers tend to contact with accounts that tweet messages and follow accounts related to specific topics. We present our guide-

lines of deploying more effective honeypots, and two lightweight, guided approaches to prioritize the sampling of more likely Twitter spam accounts.

- Through analyzing the market-level and network-level behaviors of the Android malware ecosystem, this dissertation provides a series of security observations among Android malware, and design defense approaches against them. Through analyzing the market-level behaviors, this dissertation find that: (1) Neither the location of the market nor the popularity of the apps has a strong correlation with the quality of the apps; (2) The public Android anti-virus blacklist is too slow at identifying new Android malware; (3) The same malware authors tend to submit multiple malicious apps, and within a short time period. Through analyzing the network-level behaviors, this dissertation find that: (1) There is a strong provider locality property in the Android malware's remote servers hosting infrastructure; Android malware authors tend to use cloud vendors to host remote servers to communicate with their malware samples; (2) Existing IP/domain blacklists are not effective to be used to find Android malware; (3) A few malware communities (sharing common authors or remote servers) contribute to a large portion of Android malware. We design a novel algorithm to infer more malicious apps by exploiting their community relationships, which requires neither the disassembling of Android apps nor the deep domain knowledge of the Android system.

- We design and implementation a novel system for automated extraction of Android app modalities, and use machine learning strategies to classify a given app under the modality pattern. We made an in-depth evaluation of DroidMiner with respect to its run-time performance and efficacy in malware detection, family classification, and behavioral characterization.

2. BACKGROUND AND TERMINOLOGY

2.1 Background and Terminology of Twitter

Twitter is one of the most famous online social networking websites that allows users to register accounts and use them to personal messages. In one message, at most 140 characters is allowed. Through this platform, people can easily follow up the newest update from other users they are interested. A great number of people and organizations have utilized this platform to successfully promote themselves or their business to the public. To guarantee a better service, Twitter also releases a series of Twitter Rules [108] to restrict some cyber-criminals' malicious operations on Twitter. Once an account is judged to have violated Twitter Rules by Twitter, Twitter will suspend this account. As one of the most representative Twitter Rules, "a Twitter account can be considered to be spamming, and thus be suspended by Twitter, if it has a small number of followers compared to the amount of accounts that it follows."

We next introduce basic terminologies of Twitter:

- **Tweet.** Once a Twitter user register a Twitter account and use it to post a message, this message is named as "tweet", and the user is said to have "tweeted". Users typically have a great freedom to post whatever they want to, or even post links to other websites. And, each URL will be automatically recognized by Twitter. In addition, due to the limitation of the number of characters in one Tweet, users prefer to post short URL instead of real URL.

- **Follow.** One user can choose other users they are interested in to follow. Once a user follows other users, the user will automatically and instantly receive

other users' updates, without the requirement of visiting other users' accounts' homepages. Updates for all the users one account follow will appear in reverse chronological order with the most recent update on top of the page.

- **@mention (@reply).** @metion allows one user to send a Tweet to another specific target user. No matter whether the target user followed the sender or not, the target user can read the @mention messages, which contains a string composed of the symbol of "@" and the target user's username. Before 2012, @metions will be directly shown on the users' public time. Since then, users will receive senders' @mentions in the users' Notifications, which is a tab in the users' homepage showing users' social interactions with others.

- **Direct Message.** A direct message is a private message from one Twitter user to another, which can only be seen in receivers' message inbox. Thus, direct messages can only be seen by the receivers, after they login into their Twitter accounts and check their message inboxes.

- **# (Hashtag).** Once a Tweet contains a symbol of "#" with another keyword, it implies that this tweet represents a topic. This tweet will be also indexed by Twitter and searched out by using the keyword as the search query.

2.2 Background and Terminology of Android

Android apps are composed of several components and have a complex and event-driven programming paradigm involving multiple entry points. Android defines a component-based framework for developing mobile apps. Android apps comprise four types of components: Activities, Services, Broadcast Receivers, and Content Providers. Each component in an app works as a unit performing certain tasks:

- *Activities* support basic functionalities such as interacting with end-users through graphical user interfaces (GUIs); each GUI (screen) is controlled by one Activity.

- *Services* are designed to provide interfaces in the background for communicating with other components and applications. Thus, unlike activities, services do not represent any GUI and cannot be activated/stopped by users. They will run as background processes forever until they are stopped by some certain application components.

- *Broadcast Receivers* are designed to achieve the mechanism of incident response in Android. A receiver will continuously listen to system-wide broadcast messages. When it receives relevant messages, it will automatically trigger corresponding registered events/operations.

- *Content Providers* act as database management systems, from where other components/apps could query or store an app's data without the requirement of knowing how the data is stored.

Android application authors implement Android components in an app as Java classes by inheriting corresponding super classes defined in the Android SDK (e.g., Activity, Service, BroadcastReceiver or ContentProvider). Android components are identified by other components through registration in the applications' manifest file ("AndroidManifest.XML"). This enables these components to interact with each other by using specific intents and framework API calls defined in the Android Framework. For example, an activity could activate a service by invoking the `startService()` Framework API call. In addition, unlike traditional software, the lifetimes of Android components are controlled by a series of lifecycle API functions

defined by the Android platform (e.g., `onStart()` and `onDestroy()` used in a service will start and stop the service, respectively). Moreover, the (data and control) sub-flows in an app are typically loosely connected. All these differences make Android program analysis uniquely challenging and different from traditional malware analysis.

3. RELATED WORK

In the previous two chapters, we first identified the research scope of this dissertation, and the challenges for malicious activity detection on the OSN and smartphone platforms. Then, we introduced the basic background and terminologies of these two platform. In this chapter, we will answer the following questions: why are existing techniques not sufficient for malicious activity detection on these two platforms? How are they related to or different from our solution?

3.1 Related Work on Understanding and Detecting Malicious Activities on the OSN Platforms

3.1.1 Analysis of OSN Characteristics

Due to the great popularity of the OSNs, some work has studied OSN characteristics. Mislove *et al.* present a large-scale measurement study and analysis of the structure of multiple OSNs including Flickr, YouTube, LiveJournal, and Orkut [76]. Kwak *et al.* have shown a comprehensive and quantitative study on Twitter accounts' behavior [63]. Wang *et al.* use Twitter to study the unbiased sampling algorithm for directed social graphs [117]. Cha *et al.* utilize different metrics to measure the user influence on Twitter [19]. Galuba *et al.* focus on characterizing and modeling the information cascades formed by individual URL mentions in the Twitter follower graph [42]. Castillo *et al.* design automatic methods for assessing the credibility of a given set of tweets [18]. Metaxas *et al.* analyze political community behavior and the spread of political opinions on Twitter [75], and Ratkiewicz *et al.* analyze the spread of Astroturf memes on Twitter [92].

3.1.2 Detection of OSN Malicious Accounts

Since spam and attacks are so rampant in the OSNs, many researchers have studied detecting OSN malicious accounts. A framework to detect tag spam in tagging systems is proposed in [61]. This work prevents the attackers who desire to increase the visibility of an object from fooling the search mechanism. Benevenuto *et al.* [11, 12] utilize machine learning techniques to identify video spammers on YouTube. Meanwhile, most Twitter malicious account detection work can be classified into two categories. The first category of work, such as [65, 10, 116, 102], utilizes machine learning techniques to classify legitimate accounts and malicious accounts according to their collected training data and their selections of classification features like "following-follower ratio". The second category of work, e.g., [51], detects and analyzes malicious accounts by examining whether URLs or domains posted in the tweets are labeled as malicious by public URL blacklists or domain blacklists.

3.1.3 Utilization of Honeypots

A honeypot is a decoy (e.g., a computer, data, or a network site) mainly set up to attract attackers. Traditionally, the honeypot techniques have been widely used for capturing malware and related malicious activities. Server-side honeypots are mainly implemented by emulating vulnerable services or software to trap attacks, aiming at collecting malware and malicious requests [135], understanding network and web attacks [58], building network intrusion detection systems [62], or preventing the spread of spam email [34]. Client-side honeypots are mainly used to detect compromised (web) servers [87, 52, 118, 77]. In [4], Antonatos *et al.* proposed an approach to detect instant messaging (IM) threats using IM honeypots.

In the context of OSN, social honeypots are defined as OSN accounts that appear to belong to real users, but are actually fake accounts used for attracting spammers.

Due to its simplicity and low false positives, social honeypots are a great way to collect spammers for further study, e.g., understanding their characteristics and then further building effective machine-learning features to detect them. Many existing studies [102, 65] use this social honeypot technique. However, an important missing component in this line of research is that, we still know little about the interactions between users' behaviors and spammers' actions, e.g., why this social honeypot can attract few (many) spammers. Essentially, we need *a systematic analysis on how to build more effective social honeypots*, which is an important goal of this work. Thus, this paper bridges the gap in existing research using social honeypots.

3.1.4 Measurement of Spam Campaigns and Networks.

Yardi *et al.* analyzed Twitter spam accounts' social behaviors and network structures by investigating a specific spam campaign [134]. In [43], Gao *et al.* conducted a study on detecting and characterizing social spam campaigns on Facebook, based on the observation that spam accounts in the same spam campaign, tend to send similar spam messages simultaneously. In [105], Thomas *et al.* analyzed tools, techniques, and support infrastructure utilized by spam accounts through retrospecting suspended accounts.

While most existing approaches [11, 65, 10, 102, 132] focus on detecting Twitter criminal accounts individually, we still understand far less about the properties of those criminal accounts' *social relationships* on Twitter. Yet, it is these very relationships that may be utilized by criminal accounts to increase their influence or to avoid detection and suspension. Specifically, since Twitter users can automatically obtain their following accounts' updates, criminal accounts' social relationships can aid them in increasing the visibility of their malicious content – thus in obtaining more victims. In addition, by gaining more followers, Twitter criminal accounts

can evade existing detection approaches such as "Twitter Rules" and break through Twitter's "Follow Limit Policy"[1], while maintaining their high visibility. Particularly, according to Twitter Rules [108], "a Twitter account can be considered to be spamming, and thus be suspended by Twitter, if it has a small number of followers compared to the amount of accounts that it follows."

However, we lack basic insights into the characteristics of criminal accounts' social relationships. How do criminal accounts socially connect with each other on Twitter? What is the topological structure of social relationships among those criminal accounts? Due to the fact that legitimate accounts normally do not like to follow criminal accounts, what are the main characteristics of criminal accounts' followers? Can we exploit these miscreants' tactics to build effective defense strategies against cyber criminals? The desire of addressing these questions empirically – and thus obtaining insights for defending against Twitter criminal accounts – forms the core motivation of this dissertation.

In addition, among many existing research and engineering efforts in fighting against spam/spammers, social honeypot techniques are quite promising, and have been widely deployed in existing studies to collect spammers [65, 66, 102]. A social honeypot is essentially a specially created fake account with the intent to capture spammers' social interactions. However, current social honeypots are designed to be either *too static* (few behaviors performed by honeypots) or *too uniform* (few variations among honeypots' behaviors). As a result, those honeypots are not used in an optimal or effective way to trap as many spammers as they can. The fundamental reason is that we still lack the basic insights of the strategies utilized by spammers to select spam targets. Thus, a good understanding of spammers' tastes is pressing

[1]According to this policy, once an account has followed 2,000 users, the number of additional accounts it can follow is limited to its follower number [113].

and we seriously need *systematic guidelines for building more effective (attractive) social honeypots*.

Compared with previous work on analyzing and detecting OSN malicious activities, our work focuses more on analyzing cyber criminal ecosystem – investigating inner social relationships in the malicious account community and outer relationships between malicious accounts and their supporters. Our proposed sampling strategies can provide a guided approach to prioritize the sampling of more likely spam accounts (instead of blind/random crawling) in the huge Twittersphere, thus providing a good first-layer filter for existing detection approaches. In addition, we perform a deep social honeypot measurement study to understand spammers' tastes, thus help to design new guidelines for building better social honeypots and guided strategies to prioritize the sampling of more likely spam accounts. Thus, our work is a new supplement to existing work.

3.2 Related Work on Understanding and Detecting Malicious Activities on the Smartphone Platform

3.2.1 Android Malware Detection

The growing threat of malicious mobile applications, particularly on the smartphone platform, has attracted considerable research attention. We group proposed detection approaches for mobile malware into the following three subcategories, based on the inputs that each algorithm consumes.

3.2.1.1 System Call Monitoring

Systems such as [16, 85, 94, 95] detect malware by monitoring and analysis of system calls. A fundamental shortcoming of such approaches is the semantic gap between the system calls and specific behaviors (e.g., it is exceedingly difficult to know whether an app sends an SMS to a premium number by analyzing a sequence

of Android kernel-level system calls). DroidScope [131] is designed to reconstruct both OS-level and Java-level semantics. Their dynamic analysis approach is limited by path exploration challenges, but is a useful complement to DroidMiner's static-based approach.

3.2.1.2 Android Permission Monitoring

Enck et al. studied the security of Android apps by analyzing the permissions registered in the top official Market apps [36]. Stowaway [40] and COPES [9] are designed to find those apps that request more permissions than they need. PScout [7] analyzes the usage trend of permissions in Android apps. Kirin [37] detected malicious Android apps by finding permissions declared in Android apps that break "pre-defined" security rules. More recent work also detected malicious Android apps by designing several classifiers, whose features were built primarily on the application categories and permissions [84]. A concern with these approaches is false positives stemming from the coarse-grained nature of permissions and the highly common nature of benign apps to over-claim their set of required permissions.

3.2.1.3 Framework API Monitoring

Bose et al. detected malware on Symbian OS through analyzing the temporal pattern of the usage of APIs in the DLL files [13]. TaintDroid [35] tracks the data flow and the usage of framework API calls to detect those apps that may leak users' privacy information. However, it is not designed to detect other kinds of malicious behaviors such as stealthily sending SMS. RiskRanker [143] detects malicious apps based on the knowledge of known Android system vulnerabilities, which could be utilized by malicious apps, and several heuristics, e.g., malware intends to charge the victims while blocking notifications to the victims. DroidRanger [142] detects malicious Android apps by statically matching against "pre-defined" signatures (per-

missions and Android Framework API calls) of well-known malware families. It also includes a heuristic-based approach to detect malicious applications from unknown families that requires semi-manual analysis of suspicious system calls. In [129], the frequencies of API calls were used as detection features, and more recently in [1], the names and parameters of APIs and packages were used as detection features. Both studies differ fundamentally from DroidMiner in that our modalities capture the connections of multiple sensitive API functions, not just the frequency or names of APIs. In addition, DroidMiner introduces the use of δ-analysis for sensitive node identification and associative rule mining in identifying malicious modalities. Pegasus [21] is designed to detect Android malware through abstraction of Android apps into permission event graphs, and checking whether such graphs contain predefined malicious intents. However, such manual selection of heuristics (or detection patterns) is not systematic and not robust to the evolution of malware.

3.2.2 Android Security Extensions

Existing studies have also developed several security extensions to improve the security mechanism of current Android platform including defending against confused deputy attacks and collusion attacks [32, 15], achieving fine-grained access control policies [80, 79, 29, 130, 78, 37], protecting privacy leak [53], and securing smartphone OSes [85, 64, 3, 97]. These complementary studies are developed to increase the security assurance from the phone-side, which focuses more on the quality of the smartphone systems.

3.2.3 Analysis of Attackers

A series of studies have also been conducted to understand attackers' (or spammers') behaviors in different attack scenarios. Ramachandran *et al* studied the network-level behavior of spammers such as IP address ranges that send the most

spam and common spamming modes [90]. Leontiadis *et al* measured and analyzed search-redirection attacks in the illicit online prescription drug trade [67]. Christin *et al* analyzed an online confidence scam (One Click Fraud) [24]. One recent Android malware measurement studies is made on analyzing the working mechanism of malware (e.g., the activation of the malware) [141].

3.2.4 Analysis of Mobile Traffic

A few existing studies have been conducted on analyzing mobile traffic to uncover general mobile network characteristics [39, 44, 38]. Falaki *et al* found that the browsing contributes over half of the traffic [39]. Erman *et al* examined cellular video traffic and find that only 40% of the videos are fully downloaded [38]. Through analyzing malicious traffic in cellular carriers, Lever *et al* claimed that only a vanishingly small number of mobile devices appear to be infected, and Apple's App Store and operating system do not make devices in the ecosystem more secure [68].

While most existing research efforts are spent on detecting Android malware [13, 35, 142, 129, 143, 21, 139] or designing new security extensions to defend against specific types of attacks [32, 15], we still lack some basic insights on the whole ecosystem of spreading Android malware. It is known that malware authors typically need to submit Android malware to the markets to attract victims' downloads, and build remote servers to communicate with the malware to achieve malicious goals (e.g., C&C control and compromising victims' privacy). However, the characteristics of the **market-level behaviors** and **network-level behaviors** of the Android malware ecosystem are still not well understood. Are there any special characteristics of those market accounts that submit malware? Are there any special networks mainly utilized by Android malware authors to host their remote servers? Are there any large communities among Android malware? The desire of addressing these

questions empirically, and obtaining insights for defending against Android malware, forms the core motivation of this work.

In addition, existing static analysis approaches for detecting Android malware rely on either matching against manually-selected heuristics and programming patterns [142, 21] or designing detection models that use coarse-grained features such as permissions registered in the apps [84]. We design a new system, named DroidMiner, to salably detect and characterize Android malware through robust and automated learning of fine-grained programming logic and patterns in known malware. While DroidMiner also relies on analyzing Framework API calls, it differs from existing approaches in the following ways: (1) it uses a learning-based approach to automatically generate behavior models, which are composed of individual modalities and could be used to detect malware instance from unseen families; (2) rather than simply examining whether or not the target app is malicious, it also reports specific app behavior traits (modalities); (3) instead of focusing on analyzing isolated usage of (or even the number of) Framework APIs, our detection model considers the API usage sequence, enabling DroidMiner to capture the semantic relationships across multiple APIs.

4. ANALYZING SPAMMERS' SOCIAL NETWORKS*

We have introduced the malicious activities that are launched on social network platforms, and briefly explained why the analysis of OSN spammers characteristics is important to design effective detection approaches. In this chapter, we provide our deep analysis of the spammers' social networks to reveal how malicious OSN accounts are socially connected in the OSN, and further provide an effective inference-based algorithm to sample more likely Twitter spammers [133].*

We analyze the **cyber criminal ecosystem** on Twitter, containing *criminal account community* composed of criminal (spam) accounts, and *criminal supporter community* composed of those accounts outside the criminal account community who have close friendships (following relationships) with criminal accounts, defined in our work as criminal supporters (See Figure 4.1). Specifically, we analyze **inner social relationships** in the criminal account community to reveal insights on how criminal accounts socially connect with each other. Meanwhile, we analyze **outer social relationships** between criminal accounts and their criminal supporters to reveal the characteristics of those accounts who have close friendships with criminal accounts. We also aim at finding possible reasons why criminal supporters outside the criminal community become criminal accounts' followers. Essentially, these supporters aid criminal accounts in avoiding detection by increasing criminal accounts' followers, and in preying on more victims due to the "social-intercourse" nature of Twitter (Twitter users may visit their friends' friends' profiles). Through these analyses, we aim at understanding how criminal accounts mix into the whole Twitters space, and

*Reprinted with permission from "Analyzing Spammers' Social Networks For Fun and Profit – A Case Study of Cyber Criminal Ecosystem on Twitter" by Chao Yang, Robert Harkreader, Jialong Zhang, Seungwon Shin, and Guofei Gu, 2012. Proceedings of the 21st International World Wide Web Conference, Copyright[2012] by IW3C2.

presenting new defense insights to effectively catch Twitter criminal accounts.

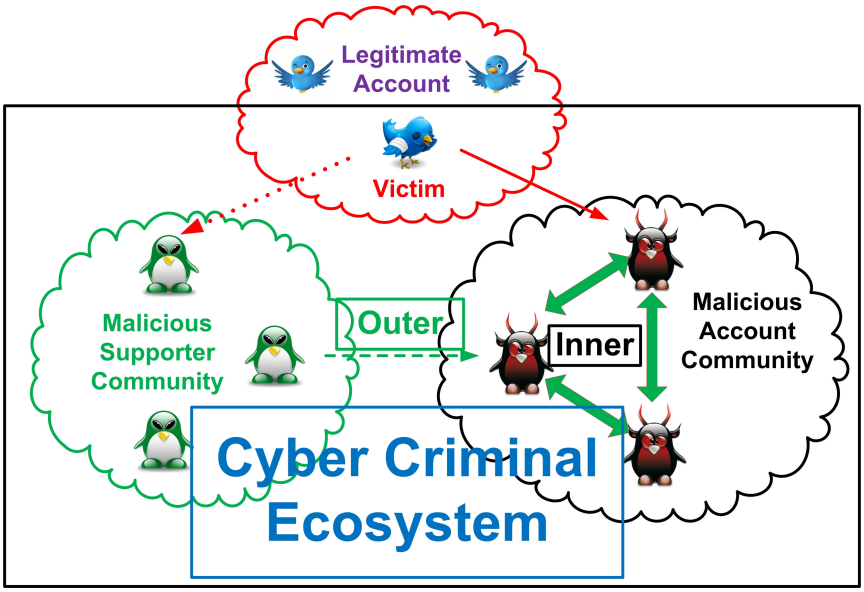


Figure 4.1: Structure of the cyber criminal ecosystem.

We conduct our empirical analysis based on a sample dataset containing around half million Twitter accounts with around 14 million tweets and 6 million URLs. After building a sample criminal account community composed of 2,060 identified spammer accounts in that dataset, we analyze its inner relationships by building and analyzing the social relationship graph. To analyze outer relationships, we propose a *Malicious Relevance Score Propagation Algorithm (Mr.SPA)* to extract criminal supporters. We then observe typical characteristics of three categories of supporters and provide possible reasons why these supporters have close friendships with criminal accounts.

We design a *Criminal account Inference Algorithm (CIA)*, to infer unknown spam Twitter accounts by starting from a seed set of known criminal ones and exploiting

the properties of their social relationships and semantic coordinations with other criminal accounts.

We make the following major contributions:

- We present the first in-depth case study of analyzing social relationships among malicious accounts. We have two main findings: (i) Malicious accounts tend to be socially connected, forming a small-world network; (ii) Compared with malicious leaves, malicious hubs are more inclined to follow malicious accounts.

- We also find that malicious accounts in some particular malicious campaign tend to have strong semantic and timing coordinations.

- We propose a new algorithm *Mr.SPA* and have extracted 5,924 malicious supporters who have close friendships with malicious accounts. We also investigate the characteristics of three representative categories of malicious supporters.

- We find that around 64% of supporters tend to build a lot of social friendships. We also find around 48% of supporters will follow back the accounts within 48 hours, who initially follow them. However, less than 2% of normal accounts would do this. This implies that malicious accounts could fully utilize these accounts to mix into Twitter.

- We design a new algorithm CIA to selectively sample and infer more malicious accounts based on a known seed set by analyzing their social relationships and semantic coordinations with other accounts. Using CIA, this dissertation can infer over 20 times more malicious accounts than that of using a random selection strategy.

4.1 Research Goal and Dataset

4.1.1 Research Goal

Our research goal is to provide the first empirical analysis on *how criminal accounts mix and survive in the whole Twitter space*. Specifically, we target on those criminal accounts as defined by Twitter Rules [108], who mainly post malicious URLs linking to malicious content with an intention to compromise users' computers or privacy. Through analyzing inner social relationships in the criminal community composed of criminal accounts (in Section 4.2), we aim at answering the following questions: What is the structure of criminal accounts' network? What are possible factors and inherent reasons leading to that structure? Are there any different social roles for different types of criminal accounts? Through analyzing outer social relationships (in Section 4.3), we aim at answering the following questions: what are typical characteristics of the accounts outside the criminal community that tend to follow criminal accounts? What are possible reasons that these accounts have close friendships with criminal accounts? Then, through exploiting criminal accounts' social relationships, we design an inference algorithm to catch more criminal accounts (in Section 4.4).

4.1.2 Dataset

To analyze criminal accounts, we crawl a large dataset of Twitter account profiles and identify Twitter spam accounts from the dataset. More specifically, we develop a Twitter crawler that taps into Twitter's Streaming API [107]. We first collect 20 seed Twitter accounts from the public timeline [111]. For each of these 20 accounts, we also crawl their followers and followings. We then repeat this process by collecting another 20 seed Twitter accounts from the timeline. For each account, we collect its 40 most recent Tweets and the URLs in the tweets. Due to the large amount of

redirection URLs used in Twitter, we also follow the URL redirection chain to obtain the final destination URL. This resulted in the collection of nearly 500,000 Twitter accounts which posted over 14 million tweets containing almost 6 million URLs (see Table 4.1).

Item	Accounts	Followings	Followers	Tweets	URLs
Number	485,721	791,648,649	855,772,191	14,401,157	5,805,351

Table 4.1: Twitter accounts crawling information.

Next, we use a relatively strict strategy to collect Twitter spammers. More specifically, we focus on those Twitter spammers, who post URLs linking to malicious content with an intention to compromise other users' computers or privacy, as mentioned in The Twitter Rules. We target at this type of spam accounts due to their excessively hazard and prevalence on Twitter. Thus, unlike other related work (e.g., [65]), we do not necessarily consider advertisers in Twitter as spammers, unless they post malicious content. To label Twitter spam accounts, we first utilize two methods to detect malicious or phishing URLs in the tweets: *Google Safe Browsing [49]* and *URL honeypot*. GSB is a widely used and trustable blacklist to identify malicious URLs, which is fast but may miss labeling malicious links. Thus, we also build a high-interaction client-side URL honeypot based on Capture-HPC [52], which will emulate a real person to click the URL in the browser in a virtual machine. The honeypot detects a link as malicious, if the visit of the linked website will modify sensitive data (e.g., process, files and registries) in the virtual machine. We define a Tweet that contains at least one malicious or phishing URL as a *Spam Tweet*. In this way, we collect 3,051 accounts by using GSB and 9,634 accounts by using honeypot,

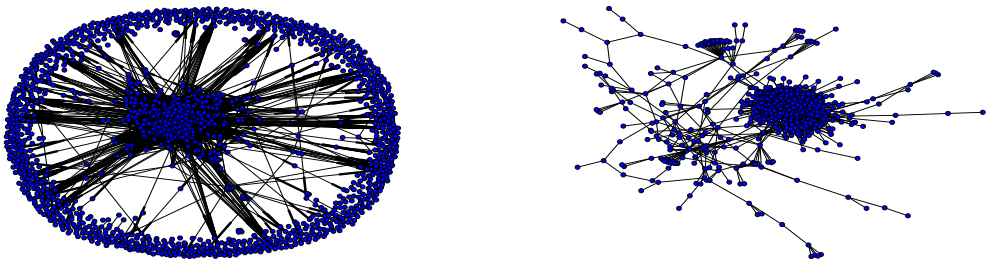
who at least post one Spam Tweet. For each account, we define its *spam ratio* as the ratio of the number of its *spam tweets* that we detect to the total number of its tweets that we collect. In this way, we extract 2,933 Twitter accounts with spam ratios higher than 10%. In order to further decrease false positives, our group members spend several days on manually verifying those 2,933 accounts by viewing whether their tweets are useful and meaningful. Finally, we obtain 2,060 identified spam accounts. Based on this dataset, we build and analyze a sample criminal account community, which is composed of those 2,060 identified spam accounts.

4.2 Inner Social Relationships

In this section, we empirically analyze inner social relationships in our sample criminal account community by visualizing its relationship graph and revealing its relationship characteristics.

4.2.1 Visualizing Relationship Graph

If we view each criminal account as a node v and each follow relationship as a directed edge e, we can view inner social relationships in the criminal account community on Twitter as a directed graph, named as the criminal relationship graph $G = (V, E)$. In our dataset, the criminal relationship graph consists of 2,060 nodes and 9,868 directed edges (see Figure 4.2(a)). By further breaking down the graph, we can obtain 8 weakly connected components containing at least three nodes and 521 isolated nodes. (Since we can partially crawl the whole Twitters space and utilize a relatively strict way of identifying criminal accounts, the number of isolated accounts may be somewhat overestimated.) The giant connected component contains 954 nodes (see Figure 4.2(b)).



(a) Relationship graph (b) Connected component

Figure 4.2: Criminal relationship graph. Each "*dot*" represents a criminal account and each "*line*" connects a pair of following and follower criminal account. The more relationships an account has, the more central it is positioned in the graph.

4.2.2 Revealing Relationship Characteristics

After visualizing our sample criminal relationship graph, we analyze this graph by utilizing graph theoretical knowledge and obtain the following two main findings.

Finding 1: Criminal accounts tend to be socially connected, forming a small-world network. From Figure 4.2(a), we can observe that criminal accounts tend to socially connect with each other. To quantitatively validate this finding, we measure three graph metrics: graph density, reciprocity, and average shortest path length.

Graph density is the proportion of the number of edges in a graph to the maximal number of edges, which can be computed as $\frac{|E|}{|V|\cdot(|V|-1)}$. This metric measures how closely a graph is to being a complete graph. A higher value implies that the graph is denser. After calculating the graph density for both our sample criminal relationship and a public entire Twitter snapshot[63] containing 41.7 million users and 1.47 billion edges, we find that the graph density of our sample criminal relationship graph, which is 2.33×10^{-3}, is much higher than that of the Twitter snapshot, which is 8.45×10^{-7}. This shows that the criminals have closer relationship than regular Twitter users.

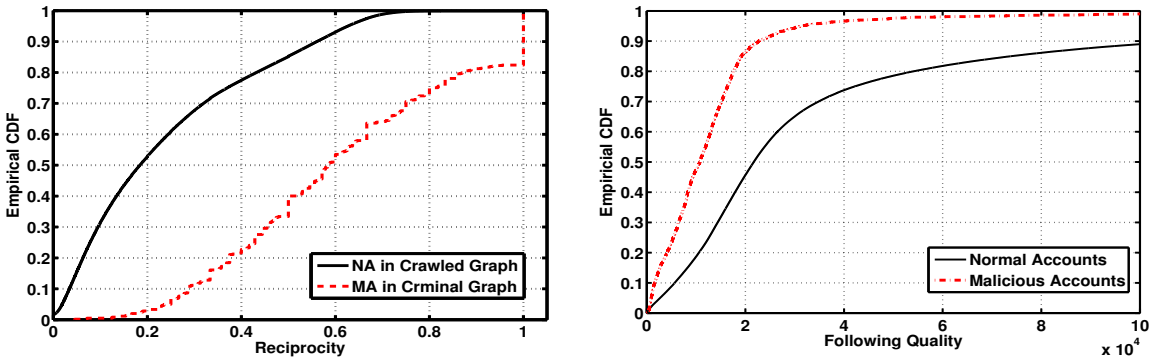
Reciprocity is represented by the number of bi-directional links[1] to the number of outlinks. We find that criminal accounts have higher reciprocity in the criminal relationship graph, but lower reciprocity in our Twitter snapshot graph (containing around $500K$ nodes). Specifically, around 5% of criminal accounts' values of reciprocity in the criminal graph are lower than 0.2, while around 45% of normal accounts and 75% of criminal accounts in our crawled graph have such values (See Figure 4.3(a)). Also, around 20% of criminal accounts' values of reciprocity in the criminal graph are nearly 1.0, i.e., other criminal accounts followed by these 20% of criminal accounts also follow them back. This observation implies that criminal accounts have stronger social relationships in the criminal account community.

Average Shortest Path Length is defined as the average number of steps along the shortest paths for all possible pairs of graph nodes. It can be used to measure the efficiency of information flow on a graph. Compared with the average path length of a sample data set with 3,000 legitimate Twitter accounts [63], which is 4.12, the average shortest path length of the criminal relationship graph is even smaller, which is 2.60. This implies that the criminal account community is also a small-world network. As an important property, a small-world network contains a giant connected component, which can be verified in Figure 4.2(b).

From the above analysis, we can find that criminal accounts have strong social connections with each other. Then, the next question we try to answer is: *what are the main factors (criminal accounts' actions) leading to that structure?*

Finding 2: Compared with criminal leaves, criminal hubs are more inclined to follow criminal accounts. To validate this finding, we examine whether criminal hubs' followings are more likely to be criminal accounts. For better description, we term a criminal account's following account as a "criminal-following", if this following

[1]There is a bi-directional link between two nodes, if they reciprocally link to each other.



(a) Reciprocity (b) Following Quality

Figure 4.3: The comparison of the criminal accounts and normal accounts.

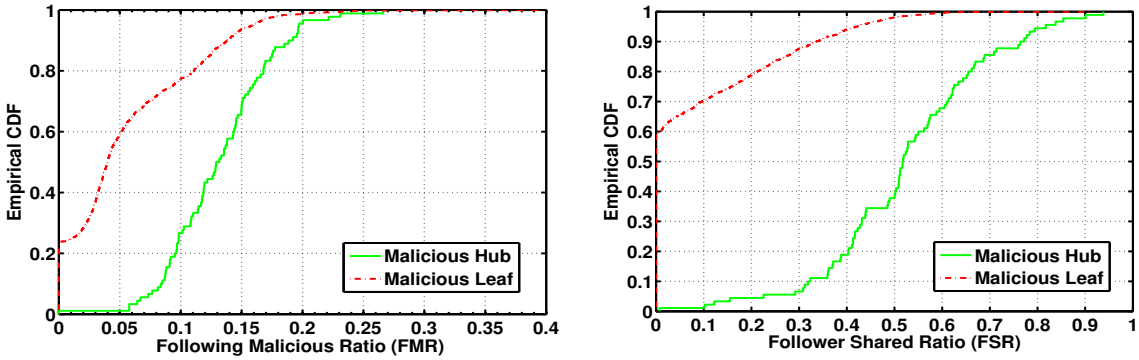
account is also a criminal. Then, we design a metric, named Criminal Following Ratio (CFR), which is the ratio of the number of an account's criminal-followings to its total following number. A higher CFR of an account implies that this account is more inclined to follow criminal accounts. From Figure 4.4(a), we can find that criminal hubs' CFRs are much higher than that of criminal leaves. Specifically, around 80% criminal hubs' CFRs are higher than 0.1, while only 20% of criminal leaves' CFRs are higher than 0.1. Also, almost no criminal hubs' CFRs are lower than 0.05, while around 60% of criminal leaves' CFRs are lower than 0.05. This observation validates that criminal hubs tend to follow more criminal accounts than leaves do. Similar to Finding 1, we next provide and validate possible explanations to Finding 2.

We make the following possible explanation for this finding: *Criminal hubs tend to obtain followers more effectively by following other criminal accounts.* Although criminal accounts could obtain followers by randomly following any account and expecting it to follow back, this method is still not very effective, due to the low chance of successfully alluring legitimate accounts to follow back. However, through

following criminal accounts, hubs can automatically acquire those criminal accounts' followers' information (Username or Account ID). Then, there is a bigger chance for criminal hubs to successfully allure other criminal accounts' followers to become their own followers, since these followers are already proved to be more susceptible to follow criminal accounts, which many legitimate accounts may not choose to do. In this way, criminal hubs can obtain followers more effectively.

To validate this explanation, we examine whether criminal hubs' followers are highly shared with their criminal-followings. Specifically, we design a metric, named Shared Follower Ratio (SFR), which is the percentage of an account's followers, who is also a follower of at least one of this account's criminal-followings. A high SFR of an account implies that most of this account's followers are also its criminal-followings' followers, i.e., this account tends to share common followers with its criminal-followings. We find that criminal hubs' SFRs are higher than criminal leaves'. Around 80% of criminal hubs' SFRs are higher than 0.4, while around 5% of criminal leaves have such values (see Figure 4.4(b)). This observation reflects that compared with criminal leaves, criminal hubs' followers share more follower information with their criminal-followings. This indirectly implies that criminal hubs could obtain followers by knowing their criminal-followings' followers' information, once these hubs follow other criminal accounts.

From these two findings, we can roughly draw a picture on how criminal accounts obtain followers on Twitter. Similar to the Bee Community, in the criminal account community, criminal leaves, like bee workers, mainly focus on collecting pollen (randomly following other accounts to expect them to follow back); criminal hubs in the interior, like bee queens, mainly focus on supporting bee workers and acquiring pollen from them (following leaves and acquiring their followers' information).



(a) Criminal Following Ratio (b) Shared Follower Ratio

Figure 4.4: The comparison between criminal hubs and criminal leaves.

4.3 Outer Social Relationships

If malicious accounts mainly build social relationships within themselves, malicious accounts can be detected by using existing sybil attack detection approaches such as Sybil Guard [136] and Sybil Infer [31]. However, many Twitter malicious accounts have already utilize several tricks to obtain followers outside the malicious account community and mix well into the whole Twittersphere [88]. Thus, those accounts, outside the malicious community who have close "follow relationships" with malicious accounts, defined as *malicious supporters*, essentially help malicious accounts avoid detection and spread malicious content [121].

However, we have little knowledge about the characteristics of those malicious supporters. Thus, in this section, we conduct the first analysis of outer social relationships between malicious accounts and their supporters including extracting malicious supporters and characterizing them. By doing this, we can reveal typical characteristics of malicious supporters and understand more on how malicious accounts can mix into the Twitter space.

4.3.1 Extracting Malicious Supporters

We first design a *Malicious Relevance Score Propagation Algorithm (Mr.SPA)* to extract malicious supports. Specifically, Mr.SPA will assign a malicious relevance score (MR score) to each Twitter account, measuring how closely this account follows malicious accounts. A higher MR score implies a closer "follow relationship" to malicious accounts. Then, we measure the MR score based on three heuristics: (1) the more malicious accounts that an account has followed, the higher score this account should inherit; (2) the further an account is away from a malicious account, the lower score the account should inherit; (3) the closer the support relationship between an account and a malicious account is, the higher score the account should inherit.

To formalize the above intuitions, we build a *malicious relevance graph* $G = (V, E)$ to model the support relationship. In this graph, we consider each Twitter account i in our dataset outside the malicious community as a node V_i. There is a directed edge e_{ij} from the node V_i to the node V_j, if the account i follows the account j. The weight W_{ij} of the edge e_{ij} is determined by the closeness of the relationship between i and j. We next introduce our malicious relevance score propagation algorithm including: initializing MR score and propagating MR score.

MR Score Initialization: Before propagating MR score, we first assign an initial score M_i^0 to each node V_i. If we denote $C = \{C_i | C_i \text{ is a malicious account}\}$, then each malicious account $C_i \in C$ is assigned a non-zero score ${m_i}^2$. For other accounts, the score is initialized to zero.

MR Score Propagation: To propagate a MR score M_i to each node V_i after the initialization phase, we make the following three score-assigning policies according

[2]In our preliminary experiment, we set $m_i = 1$.

to the above three heuristics:

- *Policy 1: MR Score Aggregation.* An account's score should sum up all the scores inherited from the accounts it follows. As Figure 4.5(a) illustrates, when A follows both malicious accounts C_1 and C_2, the score of A is the sum of the malicious scores of C_1 and C_2.

- *Policy 2: MR Score Dampening.* The amount of MR score that an account inherits from other accounts should be multiplied by a dampening factor of α according to their social distances, where $0 < \alpha < 1$. As Figure 4.5(b) illustrates, when A_1 is one hop away from a malicious account C, we assign it a dampening factor of α, where $0 < \alpha < 1$. When A_2 is two-hop away, A_2 will get a dampening factor of $\alpha \cdot \alpha = \alpha^2$.

- *Policy 3: MR Score Splitting.* The amount of MR score that an account inherits from the accounts it follows should be multiplied by a relationship-closeness factor W_{ij}, which is the weight of the edge in our *malicious relevance graph*. Specifically, we use the number of followers of an account to reflect the closeness of the relationship between this account and its followers. (The intuition is that if an account has more followers, the closeness of the relationship between this account and each of its followers will become weaker.) As Figure 4.5(c) illustrates, if A_1 and A_2 have followed the same malicious account C, the relationship-closeness factor of each account to C is 0.5. Thus, according to this policy, the score of a node V_i can be computed as $M_i = W_{ij} \cdot M_j,\ if\ (i,j) \in E$.

Before presenting our mathematical model of propagating MR score, we first introduce some notations. Let n be the number of nodes in the *malicious relevance graph*. We use the indication function $I_{ij} = \{0, 1\}$ to indicate whether $(i,j) \in E$ (i.e.,

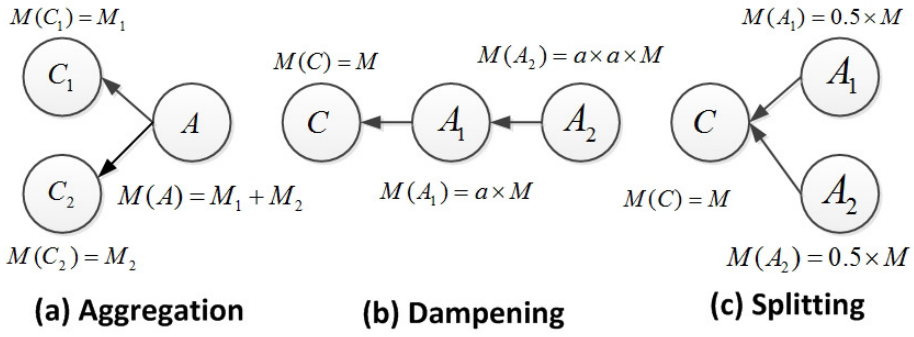


Figure 4.5: The policies of assigning MR scores.

if $(i, j) \in E$, $I_{ij} = 1$; otherwise, $I_{ij} = 0$). If we use $numIndegree(j)$ to denote the number of the indegree of the node j, then from *MR Score Splitting* policy, we can obtain that $W_{ij} = \frac{1}{numIndegree(j)}$. We use $\mathbf{I}$ to denote the column-vector normalized adjacency matrix of nodes (i.e., $\mathbf{I}_{ij} = I_{ij} \cdot W_{ij}$, if $numIndegree(j) \neq 0$; $\mathbf{I}_{ij} = \frac{1}{n}$, if $numIndegree(j) = 0$). Let $\overrightarrow{\mathbf{M}^0}$ be initial MR Score vector for all nodes and let $\overrightarrow{\mathbf{M}^t}$ be malicious score column vector for all nodes at the step t.

According to those three policies and our notations, at each step, for each node V_i, its simple MR score M_i can be computed using Eq.(4.1).

$$M_i = \alpha \cdot \sum_{j=1}^{n} I_{ij} \cdot W_{ij} \cdot M_j \tag{4.1}$$

In addition, with the consideration of each node's historical score record, at each step $t(t > 0)$, we add an initial score bias $(1 - \alpha) \cdot M_i^0$ to its simple MR Score. (In our experiment, we set $\alpha = 0.85$, since it is widely used in the random-walk model.) Thus, we can compute the MR Score column-vector $\overrightarrow{\mathbf{M}^t}$ for all nodes at the step $t(t > 0)$ by Eq.(4.2).

$$\overrightarrow{\mathbf{M}^t} = \alpha \cdot \overrightarrow{\mathbf{I} \cdot \mathbf{M}^{t-1}} + (1 - \alpha) \cdot \overrightarrow{\mathbf{M}^0} \ (t > 0) \tag{4.2}$$

When the score vector converges after several propagation steps, we can obtain final MR scores for all nodes. Once all MR scores have been calculated, a threshold is needed to determine which accounts have *sufficiently* close friend relationships to their malicious counterparts. To find an acceptable threshold, we first use x-means algorithm [83] to cluster accounts based on their MR scores. In this way, accounts with similar scores will be grouped together indicating they have similar follow relationships with malicious accounts. Then, we observe that most accounts have relatively small scores and are grouped into one single cluster. That is mainly because most accounts do not have very close follow relationships with malicious accounts. With this observation, we choose the highest score of the account in that cluster as the threshold. Then, we output 5,924 malicious supporters, whose MR scores are higher than the threshold.

4.3.2 Characterizing Malicious Supporters

After extracting malicious supporters, according to our empirical studies, we observe three representative categories of supporters (social butterflies, social promoters, and dummies) according to our defined thresholds. (Since we aim at showing preliminary and basic insights of malicious supporters' characteristics, the thresholds that are used to characterize them can be tunable according to how strictly to reflect their behavioral characteristics.)

Social Butterflies are those accounts that have extraordinarily large numbers of followers and followings. Like social butterflies in our real life, these accounts build a lot of social relationships with other accounts without discriminating those accounts' qualities. To qualitatively define social butterflies, we use 2,000 following as a threshold in terms of Twitter's *Following Limit Policy* [113], which can be an efficient number to distinguish whether the account is socialized. In this way, we can

find 3,818 social butterflies.

We present our hypothesis that *the reason why social butterflies intend to have close friendships with maliciouss is mainly because most of them usually follow back the users who follow them without careful examinations.* Especially, some public software and services[115] can help users automatically follow back other users who have followed them. In this way, these social butterflies would unintentionally follow back malicious accounts upon requests.

To validate this hypothesis, we first sign up 30 accounts without any tweets and any personal information. Then we use 10 accounts to follow 500 accounts (each account follows 50 accounts) that are randomly selected from those 3,818 butterflies. Meanwhile, we use another 10 accounts to follow another randomly selected 500 normal accounts without any tweets, and the other 10 accounts to follow another randomly selected 500 identified malicious accounts. To minimize the influence generated by our experiment, we close our signed-up accounts after 48 hours. During this timespan, we find 47.8% of those butterflies follow back to our signed-up accounts, while only 1.8% of those normal accounts and 0.6% of those malicious accounts follow back. The fast speed in which these social butterfly accounts followed our accounts back validates our hypothesis that these accounts may automatically follow back any accounts that follow them. Such a low value for those malicious accounts validates that our identified malicious accounts are not social butterflies. And they usually will not follow back other accounts, since this behavior will not increase their follower numbers and influence. This experiment also shows that even though those Twitter accounts with many followers are usually popular and trustable, we cannot totally trust their friends' quality.

Social Promoters are those Twitter accounts that have large following-follower ratios (the ratio of an account's following number to its follower number), larger

following numbers and relatively high URL ratios. The owners of these accounts usually use Twitter to promote themselves or their business. We extract those social promoters whose URL ratios (the ratio of the number of URLs to the number of tweets) are higher than 0.1, and following numbers and following-follower ratios are both at the top 10-percentile of all accounts in our dataset. In this way, we obtain 508 social promoters.

We make our hypothesis that *the reason why social promoters intend to have close friendships with malicious accounts is probably because most of them usually promote themselves or their business by actively following other accounts without considerations of those accounts' quality.* Thus, promoters may become malicious supporters by unintentionally following malicious accounts.

For this type of supporters, we use a heuristic method to validate our hypothesis. Since the goals of these promoters are promoting themselves or their business, they usually repeat posting URLs with the same domain names, which link to the webpages containing their promotion information. Thus, the purity of domain names in promoters' posted URLs are higher, leading a lower domain name entropy. With this intuition, to calculate domain name entropy for each social promoter, we extract each promoter's posted domain names in the final URLs, which are obtained through following URL redirection chains, since many URLs on Twitter are shortening URLs. Then, we can compute its domain name entropy by using $-\sum_{i=1}^{N} p_i \ln p_i$, where N denotes the number of distinct domain names and p_i denotes the ratio of the occurrences of the i-th distinct domain name to the total number of domain names.

From Figure 4.6, we can find around 40% social promoters' domain name entropy are zero, which implies that all their URLs have the same domain names. Also, social promoters' domain name entropy are lower than that of other accounts.

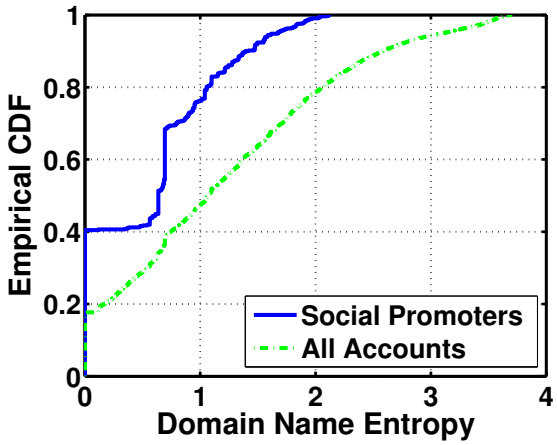


Figure 4.6: The entropy of the domain names.

Specifically, around 80% social promoters' domain name entropy are lower than 1.0, whereas around 45% of all accounts in our dataset have such values. The observation heuristically validates our hypothesis that supporters tend to use Twitter to promote themselves by actively following other accounts, leading to close relationships with malicious accounts. One case study for a social promoter can be seen in Figure 4.7(a). The owner of this promoter mainly utilizes Twitter to promote an online book selling website.

(a) Social Promoter (b) Dummy

Figure 4.7: Case studies for malicious supporters.

Dummies are those Twitter accounts who post few tweets but have many followers. Since in Twitter, legitimate users intend to follow those accounts that share more useful information, it is relatively weird that these dummies with close relationships with malicious accounts also have high follower numbers while sharing little information. In particular, we extract intriguing dummy accounts who post fewer than 5 tweets[3] and whose follower numbers are at the top 10-percentile[4]. In this way, we obtain 81 dummies.

We make our hypothesis that *the reason why dummies intend to have close friendship with malicious accounts is mainly because most of them are controlled or utilized by cyber criminals.* To validate this, we analyze these dummy accounts several months after the data collection. Then, we find that one account has been suspended by Twitter, and 6 accounts do not exist any more (closed), and 36 accounts begin posting malware URLs labeled by Google Safe Browsing, and 8 accounts begin posting (verified) phishing URLs. A case study of one dummy account, who posted no tweets at the time when we crawled its profile, starts to post malicious tweets later (see Figure 4.7(b)). The owner of this dummy account steals victims' email addresses by claiming to help people make money.

From the above experiment, we can find that unlike social butterflies and promoters, "dummy" accounts are a special type of supporters. Since they initially do not post any malicious URLs, they are not considered as malicious. However, those dummy accounts extracted by Mr. SPA could evolve to malicious accounts. The generation of this discrepancy is mainly because our work provides a static view of the ecosystem. Thus, we do not argue whether dummies are supporters or malicious accounts. This observation also implies that Mr.SPA could be applied as an early

[3]None of these tweets contain URLs that are labeled as malicious by GSB or honey client.
[4]According to [8], less than 10% of the Twitter accounts' follower numbers are higher than 100.

monitoring algorithm to catch those highly suspicious accounts, which may evolve to be malicious.

Through analyzing outer social relationships between malicious accounts and their supporters, we can understand more on how malicious accounts can mix into the whole Twitter space by achieving malicious supporters. Also, *once we extract these supporters, we can warn legitimate users not to make friends with these supporters so as to avoid exposure to malicious accounts.*

4.4 Inferring Malicious Accounts

Considering the huge number of Twitter accounts, it is impractical to make in-depth checks on every account whether it is a malicious account at the same time. A lightweight sampling or inference algorithm, to guide to more suspicious accounts instead of scanning or analyzing all accounts given limited resources or time, is indeed needed. As malicious accounts tend to be socially connected, a spontaneous and practical strategy is to first check those accounts that are connected with known malicious accounts by using Breadth First Search (BFS) algorithm. In this section, we propose a *maliCious account Inference Algorithm (CIA)* to selectively sample and infer more malicious accounts by exploiting malicious accounts' social relationships and semantic coordination.

4.4.1 Design of CIA

In brief, our malicious account inference algorithm (CIA) propagates malicious scores from a seed set of known malicious accounts to their followers according to the closeness of *social relationships* and the strength of *semantic coordinations*. If an account accumulates sufficient malicious score, it is more likely to be a malicious account.

The intuition of CIA is based on the following two observations: (1) malicious

accounts tend to be socially connected; (2) malicious accounts usually share similar topic/keywords/URLs to attract victims, thus having strong semantic coordinations among them. The first observation has been shown and discussed in Section 4.2. The second observation has also been analyzed in existing work such as [51, 43], which validates the existence of shared semantic topics among different malicious campaigns.

In general, our CIA integrates the first observation by referring to *Mr.SPA* designed in Section 4.3 to quantify the closeness of social relationships. To integrate the second observation, we use semantic similarity (SS) score to measure the semantic coordination for each pair of accounts. A higher SS score between two accounts implies that they have stronger semantic coordinations.

With the above intuitions and notions, we then describe the design of CIA in details. To infer malicious accounts in a set of U Twitter accounts, we first start from a known seed set of M malicious accounts. Then, similar to *Mr.SPA*, we build a malicious relevance graph by using these (M+U) accounts, denoted as $G = (V, E)$. In this graph, each account denotes a vertex in V and each follow relationship denotes a directed edge in E. Then, unlike *Mr.SPA*, we assign a weight for each edge $e_{ij} \in E$, by using a semantic weight assignment function $WS(i, j)$, to reflect the semantic coordination between each pair of accounts. The basic intuition of designing this function is based on that if an account has higher SS scores (stronger semantic coordination) with its followings, it should inherit more malicious score from its followings. With this intuition, for each account j, we calculate SS score between itself and each of its follower account i, denoted as SS_{ij}. Then, the weight $WS(i, j)$ of the edge e_{ij} can be calculated as: $WS(i, j) = \frac{SS_{ij}}{\sum_{e_{kj} \in E} SS_{kj}}$.

Then, similar to *Mr.SPA*, for each malicious account, we assign a non-zero malicious score and propagate this score by using the semantic weight assignment function

$WS(i, j)$. In this way, we can see that an account's malicious score can be proportionally distributed to its followers according to the closeness of social relationships and strength of semantic coordinations. When the score vector converges after several propagation steps, we infer those accounts with high malicious scores as malicious accounts.

4.4.2 Evaluation of CIA

We evaluate our malicious account inference algorithm (CIA) based on two different datasets – Dataset I and Dataset II. Dataset I refers to the one we use for the previous analyses. Dataset II contains another new crawled $30K$ accounts by starting from 10 newly identified malicious accounts and using breath-first search (BFS) strategy.

To evaluate the effectiveness of our CIA, similar to [138] that uses the number of hits in top list, we use the number of correctly inferred malicious accounts and malicious affected accounts, denoted as CA and MA, respectively. (Even though these malicious affected accounts may not be real malicious accounts, they still pollute Twitter with malicious URLs and create a risk for innocent users.) Thus, a higher number of CA and MA indicates that the algorithm is more effective to infer malicious accounts.

Note that as a lightweight *inference and ranking* algorithm aiming at magnifying suspicious accounts from a small seed set, we do *not* position CIA as a full *detection* algorithm. Thus, we adopt similar metrics to "Hit Count" used in [138] to measure CIA's effectiveness rather than using false positive and false negative rate. However, CIA could definitely be incorporated into an actual detection system by combining with other detection features.

4.4.2.1 Evaluation on Dataset I

We first design six experiments to evaluate the effectiveness of our CIA based on Dataset I:

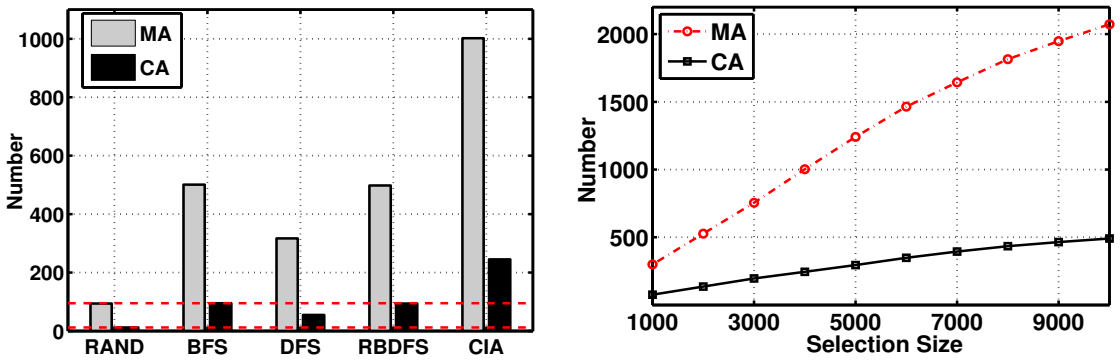
- ***Different Selection Strategies.*** In this experiment, we start from the same seed set of N identified malicious accounts, which are randomly selected from 2,060 identified malicious accounts. Then, starting from this seed set, we use the following five strategies to select five different account sets with the same selection size of k from the dataset[5]: random search (RAND), breath-first search (BFS), depth-first search (DFS), random combination of breadth-first and depth-first search (RBDFS)[6], and CIA. From Figure 4.8(a), we can see that CIA can outperform all the other selection strategies. Specifically, CIA can infer 20.42 times as many CA and 10.66 *times* as many MA as that of using random selection strategy. Also, CIA can infer 2.58 times as many CA and around 2.00 times as many MA as that of using BFS, which can infer the second most CA. Also, CIA can perform much better than the naive algorithm that considering all accounts are possible malicious accounts. Specifically, CIA can correctly predict around 0.0625 malicious accounts and over 0.25 malicious affected accounts by selecting 1 account. However, the naive algorithm can only correctly predict 0.004 malicious accounts and 0.02 malicious affected accounts by selecting 1 account.

- ***Different Selection Sizes.*** In this experiment, we start from 100 identified malicious account seeds and use CIA to infer malicious accounts by choosing different selection sizes of accounts, i.e., we evaluate our CIA by changing the

[5]In this experiment, we choose $N = 100$ and $k = 4,000$.

[6]Specifically, when RBDFS traverses to an account, it will have a probability of 50% to make a breath-first or a depth-first search in the next step.

values of k in the previous experiment. From Figure 4.8(b), we can see that when we select more accounts, we can infer more CA and MA, and the increase of CA and MA is sub-linear with the increase of the selection size.

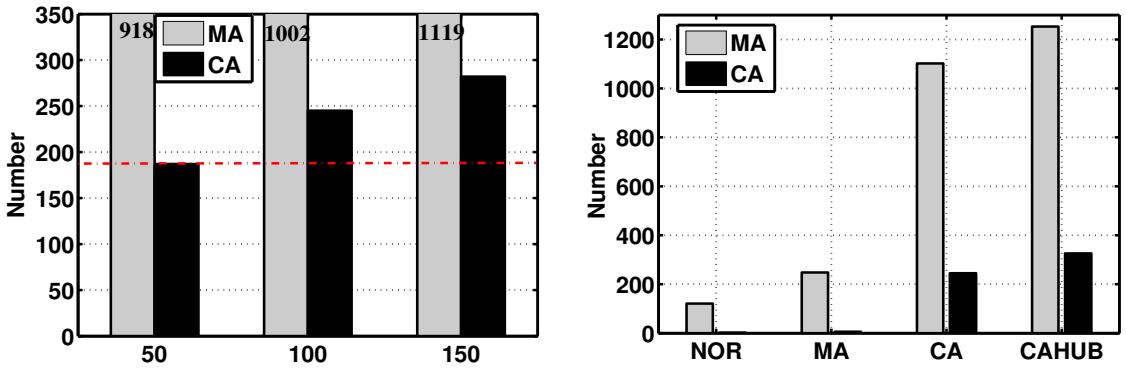


(a) Selection Strategies (b) Selection Sizes

Figure 4.8: Using different selection strategies and setting different selection sizes of accounts.

- ***Different Sizes of Seed Sets.*** In this experiment, we evaluate CIA by starting from different sizes of malicious seeds, i.e., we set different values of N. In this experiment, we also set $k = 4,000$. From Figure 4.9(a), we can see that when we increase the number of seeds, we can infer more malicious accounts while selecting the same size of accounts. This is because when we use more malicious seeds, we have more knowledge about the relationships among the malicious account community.

- ***Different Types of Seeds.*** In this experiment, we evaluate CIA by using different types of accounts as the seeds. Specifically, we start from the same number (100) of randomly selected normal accounts (NOR) (posting no malicious tweets), malicious affected accounts (MA), malicious accounts (CA), and



(a) Seed Size (b) Seed Type

Figure 4.9: Striating from different sizes of seed sets and different types of seeds.

malicious hubs (CAHUB) and use CIA to select the same amount of 4,000 accounts. From Figure 4.9(b), we can find that starting from CAHUB and CA can output much more CA and MA. Specifically, using CA we can infer 245 CA and 1,102 MA, while using MA we can infer 6 CA and 248 MA, and using NOR we can infer 2 CA and 121 MA. This observation also validates that malicious accounts have stronger social relationships and semantic coordinations among themselves. Thus, it will be more effective to use known malicious accounts other than normal accounts as seeds to infer other malicious ones. We can also find that using CAHUB can even infer more CA and MA than using CA. That is also mainly because these malicious hubs have even more social relationships with other malicious accounts than malicious leaves.

- ***Multiple Round Recursive Inference.*** In this experiment, we initially start from a small set of randomly selected 50 identified malicious accounts to recursively run CIA to infer malicious accounts. Specifically, during each round, we will combine previous round's seeds and identified malicious accounts correctly inferred in the previous round as new seeds to run CIA again. From

Figure 4.10, we can find that even when we start from a small number of malicious accounts (50, which is around 2.4% of all CA in the dataset) within running 3 rounds of CIA, we can infer around 9 times more malicious accounts (500, which is around 22.3% of all CA). This observation shows that we can use CIA to recursively infer more malicious accounts by adding newly correctly inferred malicious accounts into the existing seed set.

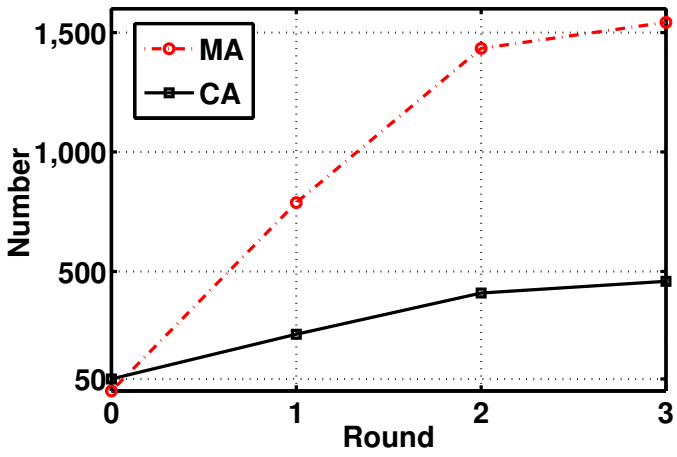


Figure 4.10: Evaluation of multiple round recursive inference.

- **Performance.** In this experiment, we examine the time used by CIA to infer malicious accounts. Our CIA mainly contains three steps: Generating Social Graph (Step 1), Calculating Coordination Weight (Step 2), and Propagating Malicious score (Step 3). SInce the propagation of malicious score essentially require the computation of a sparse matrix, we implement our propagation algorithm based on SparseLib++ [86], which is an open library for efficient sparse matrix computations. Specifically, we examine the time used for obtaining malicious scores by starting from 100 randomly selected known malicious

accounts. Table 4.2 shows the time used for each step to output the final malicious score.

Step	**Step 1**	**Step 2**	**Step 3**	**Total**
Time	5.42 second	126.38 second	27.47 second	159.27 second

Table 4.2: The time (in second) used for each step in CIA to output malicious score.

From this table, we can see that the total time is less than 160 seconds, which shows the efficiency of our CIA algorithm. The most time consuming step is calculating coordination weight, which needs to calculate the semantic similarity.

4.4.2.2 Evaluation on Dataset II

To decrease the effect of possible sampling bias in our analyzed dataset and to show the fact that the performance of CIA are reproducible, we also test CIA on another newly crawled dataset. Also, to guarantee the correctness of identifying malicious accounts, we first use Google Safe Browsing, a trustable blacklist, to collect malicious affected accounts. Then, we manually identify malicious accounts from those malicious affected accounts[7]. Then, we examine the effectiveness of CIA on newly crawled dataset by comparing different account selection strategies. Specifically, we start from only 10 identified malicious accounts and select 4,000 accounts by using each strategy. From Figure 4.11, we can also find that CIA can generate the best results. CIA can infer 13 more malicious accounts than that of using RAND.

Through the above experiments, we can find that our malicious account inference algorithm (CIA) can be used to effectively infer unknown malicious accounts. Also,

[7]We acknowledge that the numbers of CA and MA are the low bound of real numbers in the dataset, because we can not detect all CA and MA by simply using GSB itself.

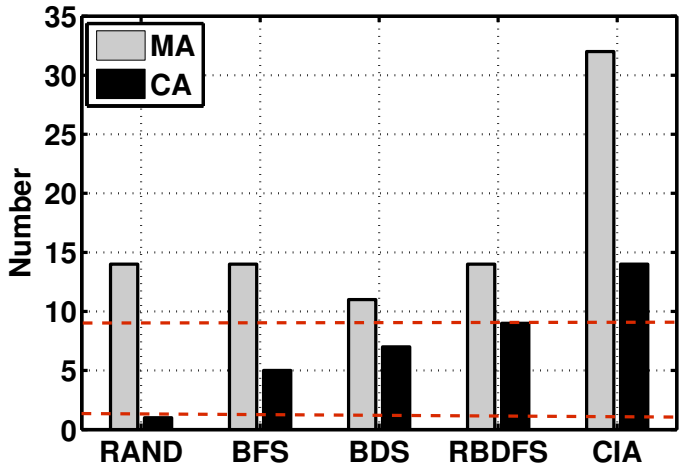


Figure 4.11: Evaluation on Dataset II.

unlike most current work on detecting Twitter spammers based on machine learning techniques, which require extracting many features from all the accounts in the dataset, CIA mainly focuses on those accounts that have strong social relationships with existing known malicious accounts. In addition, CIA can be utilized to work as an early-stage monitoring and ranking algorithm to monitor those highly suspicious accounts, which may evolve to be malicious accounts later.

4.5 Limitation

We acknowledge that our analyzed dataset may contain some bias. Also, the number of our analyzed malicious accounts is a lower bound of the actual number in the dataset, because we only target on one specific type of malicious accounts due to their severity and prevalence on Twitter. However, it is extremely challenging to obtain an ideal, unbiased dataset with perfect ground truth. Especially, to reduce possible data sampling bias, we crawled two datasets at very different time to evaluate the performance of our CIA. We also believe that even though the exact values of some metrics used in our work may vary a little bit when using different sample

datasets, our major conclusions and insights will likely still hold. Also, our analysis is mainly based on a snapshot of Twitter space, which only provides a static view.

We also acknowledge that our validations on some possible explanations proposed in this work may be not absolutely rigorous, due to the difficulties in thoroughly obtaining malicious ' social actions or motivations. However, we believe that our first-in-its-kind analysis of those phenomenon still provides great values and opens a door to better understand the cyber criminal ecosystem on Twitter.

4.6 Summary

We have presented an empirical analysis of the cyber criminal ecosystem on Twitter, including an in-depth analysis of the inner and outer social relationships among malicious accounts. We observed that malicious accounts tend to be socially connected, and malicious accounts in some particular malicious campaign tend to have strong semantic and timing coordinations. We also observed three categories of accounts that have closed social relationships with malicious accounts. Based on these findings, we designed an inference algorithm to selectively sample more likely malicious accounts based on a known seed set by analyzing their social relationships and semantic coordinations with other accounts. We evaluated the algorithm's inference capabilities in a pre-crawled dataset by using different Selection Strategies, setting different values of selection sizes and seed sizes, and using different types of seeds. To prove that the performance of the inference algorithm is reproducible, we also evaluated the algorithm by using a newly crawled dataset from a very small seed of known malicious accounts by using different crawling strategies. Our experience demonstrates that the algorithm can be effectively used to sample more likely spam accounts from a seed set of known malicious accounts. Our analysis of the cyber criminal ecosystem on Twitter is also the first in-depth analysis of the social rela-

tionships among malicious OSN accounts. We hope that our analysis can inspire more studies in this research direction.

5. REVERSE ENGINEERING TWITTER SPAMMERS

We have described our analysis of the social relationships among OSN spammers. Through understanding the social relationships among OSN spammers, we find three major categories of accounts that have close social relationships with spam accounts. We also design two inference algorithms to infer unknown spam Twitter accounts by starting from a seed set of known criminal ones and exploiting the properties of their social relationships and semantic coordinations with other criminal accounts. However, the spammers begin to evolve more evasive and to increase the success chance of obtaining victims, by choosing specific accounts as spamming targets instead of choosing random accounts. In this chapter, we present a novel defense insights against those OSN spammers by reversing engineering the strategies used by spammers to select their spam targets.

Restricted by OSNs' anti-spam measures, many OSN spammers have evolved to launch *Targeted Social-Media Spamming* (i.e., spammers selectively choose their spamming targets by analyzing those targets' behaviors [101]). Twitter users have undergone the following experience: once they write some big brand names such as "Ipad" or "Best Buy" in their tweets, they will receive a slew of tweets offering "free" products or gift cards related to the brands [69, 93]. Such observations indeed imply an obvious interaction between users' social behaviors and spammers' actions (as illustrated in Figure 5.1).

The benefit for spammers to use this strategy to find targets is straightforward. Through selectively choosing targets to initialize unsolicited friend requests or send unsolicited messages, spammers could significantly decrease the risks of being detected under current OSNs' policies. (According to our observation, a Twitter ac-

Figure 5.1: Illustration of interactions between users' social behaviors and spammers' actions.

count, who constantly follows more than 50 accounts per day, will highly possibly be suspend by Twitter within a week.) Furthermore, after knowing targets' tastes or social friend-circles, spammers could significantly increase their chances of successfully spamming, either by actively pushing spam messages related to targets' tastes (e.g., on Twitter) or pretending to be in the same social friend-circle (e.g., on Facebook). In this way, social spammers could garner victims more effectively by launching customized actions based on their targets' social behavior characteristics. Thus, this is different from the scenario for traditional email spam or web spam, in which attackers usually know nothing about their targets and can merely blindly send spam.

However, we still know little about basic insights of the interactions between users' behaviors and spammers' actions, which could be used to catch spam accounts. Also, such insights may further facilitate us to understand common questions such as "Why do I get spam friends? [89]", "Why do I receive spam messages? [96]" and "How do spammers find their targets?". The desire of addressing such questions, and thus *obtaining insights for defending against social spammers*, forms one motivation of this work.

Furthermore, although many existing studies rely on social honeypots (or even manual identification) to collect likely spam accounts (aiming at further analyzing them to generate defense insights), such strategies are still not very efficient in terms of collecting a large-scale of spam accounts from the huge Twittersphere. In particular, the technique of social honeypots is relatively passive and typically requires a long time to attract many spam accounts. The strategy of manually labeling spam accounts is tedious, time-consuming, and very difficult to scale. Thus, given limited resources/time, a light-weight strategy to selectively sample more likely spam accounts from the huge Twittersphere is strongly desired.

Through reverse engineering spammers' tastes (their preferred targets to spam), this chapter provides guidelines for designing effective social honeypots, and designing lightweight and guided strategies to actively sample more likely social spam accounts. To achieve this goal, we use Twitter as a case study due to its great popularity and publicity. Specifically, to reveal which behaviors tend to incur spammers' contact, we implement 96 "benchmark" Twitter social honeypots with 24 diverse fine-grained social behavior patterns to trap spam accounts. After launching our social honeypots for five months, we successfully garner around 600 spam accounts. Using these data, we analyze spammers' tastes (how spammers find their targets), through comparing the effectiveness of social honeypots with different behavior patterns. Based on these analyses, we design and implement 10 more effective ("advanced") honeypots to trap Twitter spammers. Within the same time period, using those advanced honeypots can trap spammers around 26 times faster than using "traditional" honeypots. To further understand spammers' tastes, we also design an algorithm to extract semantic topic terms, which may highly attract spammers' attentions.

We design two guided approaches to prioritize the *active* sampling of more likely

spam accounts from Twittersphere. These two approaches are designed by the analysis results obtained through reverse engineering spammers' strategies of selecting targets. They are an effective complement to existing *passive* social honeypots.

We make the following major contributions:

- We present a deep analysis of spammers' tastes: spammers tend to contact with accounts that tweet messages and follow accounts related to specific topics.
- We deploy "advanced" (more effective) honeypots based on our provided guidelines, which can trap spammers around 26 times faster than using "traditional" honeypots.
- We design two lightweight, guided approaches to prioritize the sampling of more likely Twitter spam accounts in the huge Twittersphere. According to our evaluation, our samplers can efficiently collect over 17,000 Twitter spam accounts in a short time with a considerably high "Hit Rate" (correctly collect 0.6 spam account per sampled account).

In Section 5.1, we further clarify the problem that we are targeting. In Section 5.2, we detail the procedure of our collection and analysis of spammers' interests. In Section 5.3, we describe the motivation and algorithm design of our two lightweight samplers to infer more likely Twitter spam accounts. In Section 5.4, we present our evaluation results of those two samplers. We discuss our limitation in Section 5.5.

5.1 Problem Statement

We next introduce the research scope of this work. Our research goal is to understand the characteristics of one special type of Twitter spammers, who launch targeted social-media spamming in Twitter, and further to gain new defense insights against them by reverse engineering their spamming tastes. Particularly, we use a

relatively strict/conservative view (similar to existing work [27] and Twitter rules [108]) to consider an account to be a spam account, if it meets one of the following criteria: (1) tend to post spam or malicious URLs in the tweets; (2) tend to post scam words in the tweets; (3) repeatedly post duplicate tweets; (4) repeatedly send "@mention" messages to other accounts with few useful content.

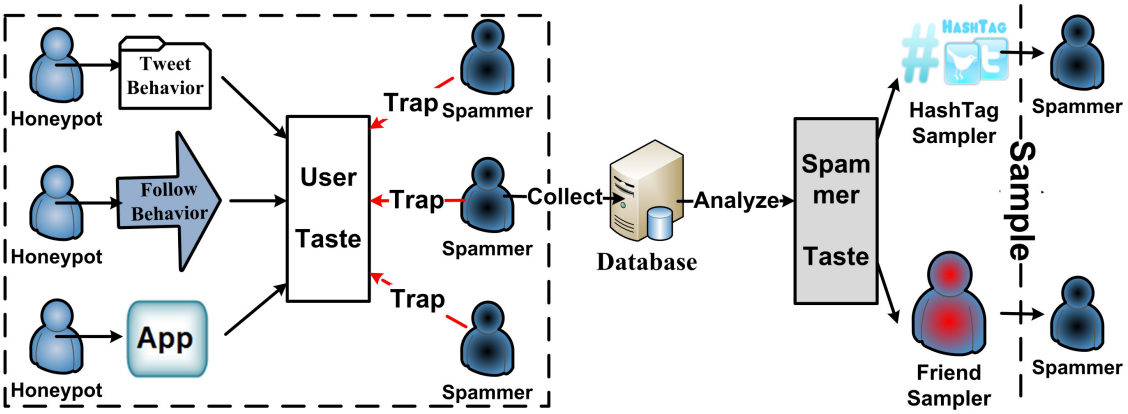


Figure 5.2: Illustration of the analysis flow.

To achieve our research goal, we first design 96 social honeypots with diverse social behavior patterns to garner spammers (see Figure 5.2). Based on the functions provided by Twitter, these social behavior patterns mainly vary in terms of tweeting behaviors, following behaviors, and application usage. Particularly, the content posted by users, the famous accounts followed by users and the applications used by users may display users' tastes, incurring spammers' contact. Then, these social honeypots could trap spammers by receiving spammers' unsolicited messages and obtaining spam followers. All of social honeypots' behavior and their trapped spammers' actions (sending unsolicited messages or building unsolicited friendships) will be saved in a local database.Next, after deeply analyzing spammers' tastes by

comparing the effectiveness of honeypots with different social behavior patterns, we can provide guidelines to build effective honeypots. Finally, through reverse engineering spammers' strategies of selecting targets, we design two lightweight, guided strategies (Hashtag Sampler and Friend Sampler) to prioritize the sampling of more likely Twitter spam accounts from the huge Twittersphere. More specially, Hashtag Sampler is designed to catch spammers that target on specific accounts if they tweet specific hashtags. Friend Sampler is designed to catch spammers that target on specific famous accounts' followers.

5.2 Reverse Engineering Spammers

In this section, we describe our methodologies of extracting and analyzing social spammers' tastes. Specifically, we design and launch multiple social honeypots with diverse fine-grained behavior patterns to garner spammers. Next, through analyzing intrinsic properties of the interactions between users' social behaviors and spammers' actions, we could better understand the following questions: Who do spammers spam? How do spammers find their victims? Through these analyses, we further provide guidelines of building more attractive social honeypots.

5.2.1 Collecting Spammers' Tastes

To analyze the interactions between users' behaviors and spammers' actions, we need to endow social honeypots with diverse fine-grained social behavior patterns to show diverse users' tastes. As a Twitter account mainly has three categories of social behaviors (posting tweets, following accounts and installing applications), we design social honeypots based on the variations of these three categories: Tweet Behavior (Tweet), Follow Behavior (Follow) and Application Usage (App) (see Table 5.1).

Index	Category	Sub-Category	Pattern
1-5	Tweet	Frequency	Once per day
6-10	Tweet	Frequency	Twice per day
11-15	Tweet	Frequency	Once per hour
16-20	Tweet	Keywords	Trending Topics
21-25	Tweet	Keywords	Arbitrary Hashtags
26-30	Tweet	Keywords	Current Affairs
31-35	Tweet	Keywords	Bait Words
36-40	Tweet	Keywords	No Hashtags
41-45	Tweet	Topic (Twice per day)	Entertainment
46-50	Tweet	Topic (Twice per day)	Expertise
51-55	Tweet	Topic (Twice per day)	Sports
56-60	Tweet	Topic (Twice per day)	Economics
61-62	Tweet	Topic (Once per hour)	Entertainment
63-64	Tweet	Topic (Once per hour)	Expertise
65-66	Tweet	Topic (Once per hour)	Sports
67-68	Tweet	Topic (Once per hour)	Economics
69-70	Follow	Two accounts per day	Entertainment
71-72	Follow	Two accounts per day	Expertise
73-74	Follow	Two accounts per day	Sports
75-76	Follow	Two accounts per day	Economics
77-81	App	NA	Twitpic
82-86	App	NA	Instagr
87-91	App	NA	Twiends
92-96	Default	NA	NA

Table 5.1: Summary of 96 "benchmark" social honeypots with 24 fine-grained social behavior patterns.

5.2.1.1 Tweet Behavior

The content tweeted by users (and tweet frequency) may directly expose users' interests. Particularly, the keywords/topics posted by users may reveal their tastes, which could be utilized by spammers to find targets. In fact, users' real experience has shown that different tweet keywords may behave very differently in terms of incurring spammers [93]. Accordingly, we divide our social honeypots' tweet behaviors into three sub-categories: Tweet Frequency, Tweet Keywords and Tweet Topics.(To

reduce possible effects to other users, our social honeypots will not post any links and "@ mentions".)

Tweet Frequency refers to how often post one tweet. We divide tweeting frequency into the following three patterns: 1 tweet per hour, 2 tweets per day, and 1 tweet per day. For each pattern, we use 5 honeypot accounts to send tweets according to the specific tweet frequency. Those tweets are randomly selected from the dataset containing around half million Twitter accounts and 14 million tweets, which was collected from Apr. 2010 to Aug. 2010 by using Twitter Stream APIs.

Tweet Keywords refer to special words or terms in the tweets, which may represent specific semantic topics. We divide tweet keywords into four patterns: popular trending topics, arbitrary hashtags, current affairs, bait words, and no hashtag tweets:

- "Popular trending topics" refer to those hot Twitter trending topics [109], which are widely used by Twitter users to express their opinions or experience on specific topics or events. For each day, we collect top (the most widely used) 10 trending topics. Then, we use 5 honeypot accounts to post these 10 trending topics. Each of them will post 2 trending topics.

- "Arbitrary hashtags" refer to those tweet terms with the tag of "#". The tweets containing the same hashtag will be grouped together by Twitter and searched out by users from Twitter Search [112]. These hashtags are also randomly selected from the pre-collected dataset. For each day, we use 5 honeypot accounts to send 10 tweets with hashtags. Each of them sends 2 tweets, which are randomly selected from our collected dataset.

- "Current affairs" refer to important social events happened each day. To extract those events, we implement a web crawler to crawl the top 10 headlines

from CNN.com. Then, we use 5 honeypots to post those headlines (each posts two headlines).

- "Bait words" refer to those keywords that are mainly used by spammers in their scam webpages or messages to trap victims (e.g., "giftcard"). We use a list of 200 bait words, mainly obtained through feeding queries such as "scam word lists" to Google.com. Then, we also use 5 honeypots to send 2 messages containing bait words per account per day.

- "No hashtags" refer to tweets without any hashtags. To make the comparison with other social patterns, we also use 5 honeypots to post 10 tweets without any hashtags on each day. Each of them will post 2 tweets, which are randomly selected from the dataset.

Tweet Topics refer to specific semantic topics in the tweets. Since these tweets are closely related with specific semantic topics, they will explicitly reveal users' tastes. Particularly, we focus on the following four topics: *Entertainment, Expertise, Sports and Economics*. Entertainment contains those semantic topics related to TV media, music, books and arts; Expertise contains topics related to IT technology, Science, Fashion and Household; Sports contains topics related to golf, NBA, NCAA, NFL and NHL; Economics contains topics related to business, finance and charity. To use our honeypots to tweet those semantic topics, we first collect tweets related with those topics by searching topic terms (e.g., "NBA") on Twitter. Then, for each topic, we use 5 honeypots to send one tweet per day. To compare, we also use 2 honeypots to send 1 tweet per hour.

5.2.1.2 Follow Behavior

Besides the content posted by users, users' followings (especially those famous people or companies' official accounts) may also reveal their tastes. For example, if an account follows "Lady Gaga", the owner of the account may like music or live concert. Thus, this kind of following tastes might be utilized by spammers.

To extract spammers' such tastes, we use our honeypots to follow "verified accounts", whose tweets are related with four major topics mentioned above: Entertainment, Expertise, Sports and Economics. Specifically, for each topic, we manually collect 400 verified accounts from Twitter. These verified accounts are typically owned by famous people or organizations with high reputation, such as sports stars and official business accounts. Thus, through following those verified accounts, our honeypots explicitly show their interests to those topics. Particularly, for each topic, we use 2 social honeypots to follow 2 verified accounts per day. (To reduce possible effects, each account will follow 30 verified accounts at most.)

5.2.1.3 Application Usage and Default

Users who install specific Twitter applications (e.g., multimedia sharing tools and online games) may also reveal their specific tastes and thus become spammers' targets. In our test, we choose three very popular Twitter social applications: Twitpic [106], Instagr [54] and Twiends [110]. (Twitpic and Instagr are popular photo and video sharing tools, and Twiends is an online Twitter friend-making tool.) For each application, we use five honeypots to install and use it.

As a comparison, we also use five honeypots with default account registration configuration, which neither post any tweets nor follow any accounts.

5.2.1.4 Deployment of Honeypot

In summary, as seen in Table 5.1, we design 96 honeypots with 24 diverse fine-grained behavior patterns to garner spammers. Since the aim of designing these social honeypots is to understand which specific social behaviors tend to incur spammers rather than to trap more spammers, we refer these 96 honeypots as "benchmark" honeypots.

To implement those "benchmark" honeypots, we develop a realtime Twitter application, named social honeypot app (SHP), which has three major operations: write, follow, and read. As illustrated in Figure 5.3, write operation is utilized to implement diverse tweet-behaviors by posting tweets on honeypots' timelines; follow operation is utilized to implement diverse follow-behaviors by following other accounts; read operation is utilized to collect spam accounts and spam tweets trapped by our social honeypots, through reading honeypots' followers and "@mentions".

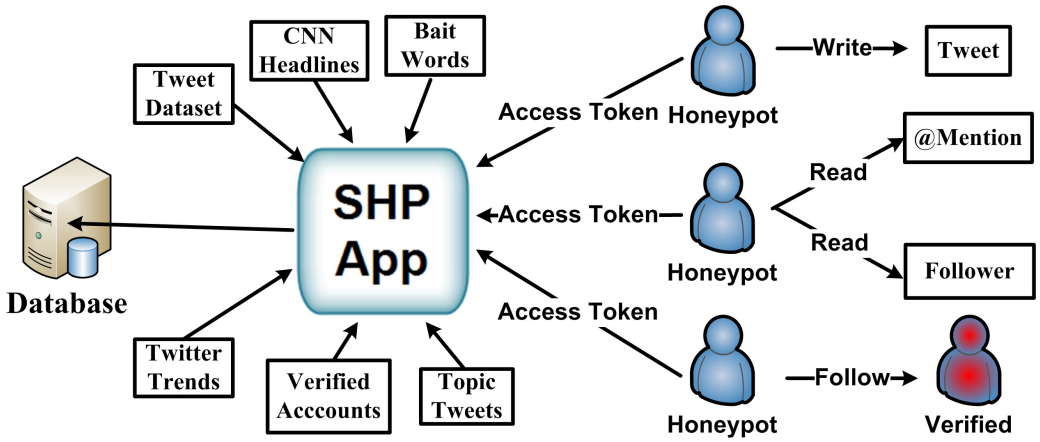


Figure 5.3: The implementation of social honeypots.

More specifically, the application obtains each honeypot's access token to auto-

matically make the corresponding operations (write, follow and read) on the account to perform its designed social behaviors according to the protocol of OAuth 2.0. All the auxiliary data such as our collected tweet dataset, popular trending topics and bait words are loaded into the app to implement corresponding operations. Finally, the app will record each honeypot's social behaviors, and its received "@mentions" and followers into a local database everyday. In this way, we can collect the interactions between honeypots' behavior patterns and their trapped spammers' actions.

Particularly, to make our honeypots to be more likely to be real accounts (i.e., to decrease the chance of being identified as honeypots by spammers), we register our honeypot accounts by using real human names (e.g., *Tracy Thompson*) and valid email addresses. Also, we will initialize the friendships among those honeypots. We admit that smart shammers might still recognize our social honeypots by deeply analyzing those honeypots' behaviors, because these honeypots are designed with a set of scheduled tasks. However, many normal accounts (e.g., some official company accounts) are also customized to post particular messages/notifications in a scheduled way. Thus, it is not that trivial for spammers to distinguish honeypots from normal accounts. Also, this limitation is common for all this line of studies, which rely on deploying automated honeypot accounts.

5.2.2 Analyzing Spammers' Tastes

We next show the results of spammers collected by "benchmark" honeypots, and analyze those spammers' tastes.

5.2.2.1 Data Collection Result

We implemented those 96 "benchmark" honeypots and run them for five months. We collected $1,077$ unique accounts that at least follow one of our social honeypots, and 440 unique accounts that at least send one "@mention" to one honeypot. In

total, there are 1,512 unique accounts.

To extract spammers' tastes, we need to identify spam accounts from those 1,512 accounts. We first found out 303 accounts that have been suspended by Twitter due to their violations to the Twitter Rule. Furthermore, following the definition of our target spam accounts' described in Section 5.1., we identified additional 278 spam accounts by manually examining accounts timeline and checking those accounts' posted URLs. In total, we obtain 578 spam accounts.

Note that the number of spam accounts trapped by our "benchmark" honeypots seems a little smaller than other earlier social honeypot studies (e.g., [65]). We believe this is due to the following reasons. First, those studies were conducted in early days when Twitter has relatively loose policies to identify/mitigate spammers. However, since 2009, Twitter has taken significant anti-spam efforts to actively filter/mitigate a lot of spam accounts [23]. In addition, in this work, to guarantee the correctness to analyze spammers' interests, we use a relatively strict way to consider an account to be spam. While there could be a few spam accounts missed in our data collection with this relatively strict spammer identification strategy, we believe that our major findings/conclusion in this research will still hold.

5.2.2.2 Analysis of Spammers' Tastes

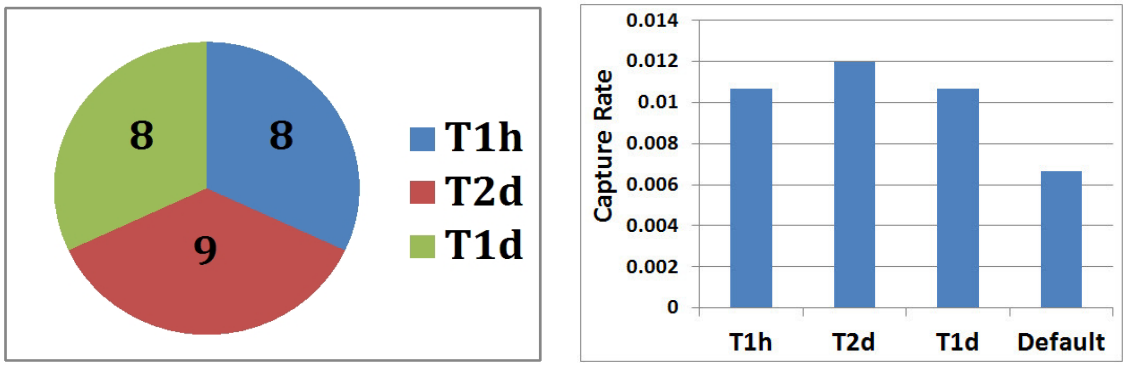
In this section, we provide our analyses of spammers' tastes based on 578 trapped spam accounts. To better measure the effectiveness of social honeypots with different behavior patterns, we define a metric named *Capture Rate (CR)*, which is the average number of spam accounts trapped by a honeypot per day. Thus, a higher value of *CR* of honeypots with a specific pattern implies this pattern is more effective to trap spammers. As mentioned before, we try to answer several questions about the social behavior interactions between users and spammers (e.g., "Who do spammers

spam?"). Then, our analysis and measurement results are presented in the question-answer format.

Q1: Do spammers tend to find targets by randomly selecting accounts from Twitter public timeline? Empirical Answer: No. One possible way for spammers to find targets is to send requests to the public timeline, which will return the 20 most recent tweets per request. However, according to our observation, it is not the case now. As seen in Figure 5.4(a), we can find that even though we diversify tweeting frequencies (once per hour – `T1h`, twice per day – `T2d`, and once per day – `T1d`), the performances of these three patterns are similar (garnering similar numbers of spammers). Particularly, although `T1h` honeypots post more tweets than `T2d` and `T1d`, `T1h`'s CR, which is 0.011, is even slightly smaller than that of `T2d`, which is 0.012 (see Figure 5.4(b)). Thus, this observation shows that if an account posts more arbitrary tweets, it does not necessarily increase the chance of attracting spammers' more attentions, even though this behavior will bring in a better chance for the account to be shown on the Twitter public timeline.[1] Nevertheless, we can see that these three types of honeypots can trap more spammers than `Default`, because tweeting content essentially may reveal honeypots' tastes.

Q2: If an account posts more tweets related with specific semantic topics, does it tend to attract more spammers' attention? Empirical Answer: Yes. Another possible way to find targets for spammers is to analyze targets' tweet content, which may show targets' interests on specific topics. Then, through actively pushing spam related to those topics to those targets, attackers may achieve a higher chance of success. As seen in Figure 5.5(a), posting messages related with specific topics (Entertainment – `TEn`, Sports – `TSp`, Economics – `TEc`, Expertise – `TEx`), will incur more

[1]To reduce the possible interference to Twittersphere, we did not test with extremely high tweeting frequencies. Thus, we do not deny the possibility that our conclusion might be somehow different if some accounts post tweets with an extremely high frequency.

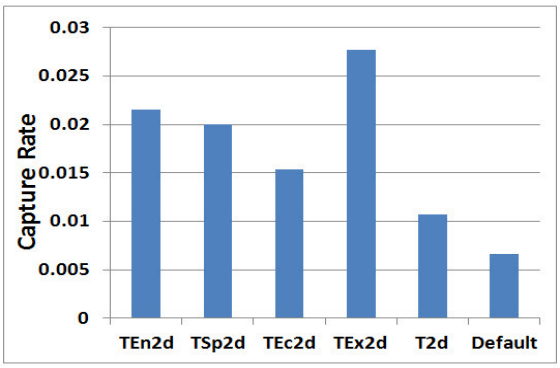


(a) # of Trapped Spammers (b) Capture Rate

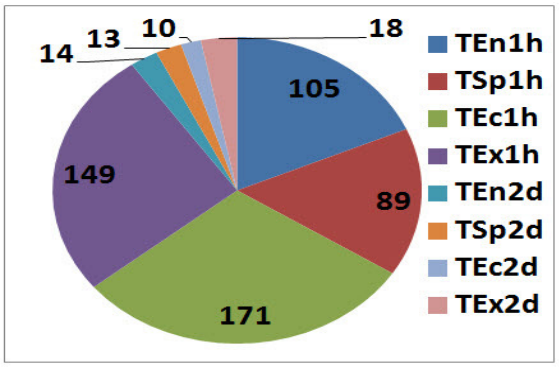
Figure 5.4: Comparison of different tweet frequencies.

spammers' contact than posting arbitrary messages even with the same tweeting frequency (twice per day). More specifically, `TEx2d`'s CR (the highest for tweeting topic twice per day) is around 3 times as that of `T2d`, and `TEc2d`'s CR (the lowest) is around 1.5 times as that of `T2d`. In addition, when we increase the frequency from twice per day to once per hour (e.g., from `TEn2d` to `TEn1h`), honeypots can trap more spammers (See Figure 5.5(b)). And the average values of CR for these four topics can be increased around 22.35 times (from 0.021 to 0.494). Thus, unlike the observation under the pattern of tweet frequency, we can find that honeypots can trap more spammers through tweeting more messages related with certain semantic topics.

Q3: Do accounts that tweet more special terms (e.g., "Trending topics") tend to attract more spammers' contact? Empirical Answer: Yes. As seen in Figure 5.6(a), the values of CR for tweeting trending topics (`Trend`), arbitrary hashtags (`Hashtag`), and bait words (`Bait`), are all higher than that of `Nohash` (arbitrary tweets without hashtags) and `Default`. This observation indicates that posting special key terms may also incur spammers' contact. This because these key terms usually represent semantic topic meanings, which could be utilized by spammers to find targets (similar



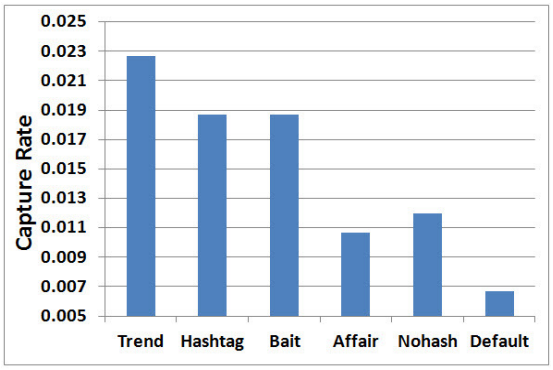
(a) Tweet Topics



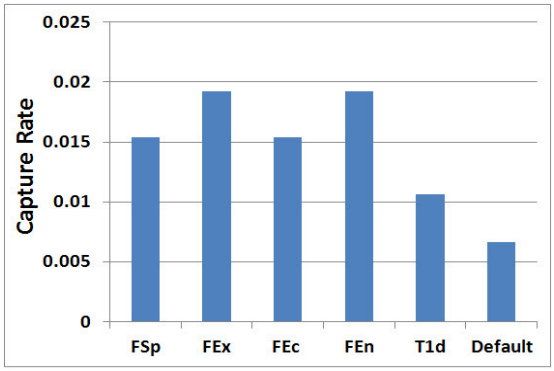
(b) # of Trapped Spammers

Figure 5.5: The effectiveness of tweet topics.

to tweeting topics). In addition, we can find that `Trend` is more effective than `Hashtag`. This might because trending topics are more timely and popular than arbitrary hashtags.



(a) Tweet Keywords



(b) Follow Behavior

Figure 5.6: The effectiveness of tweet keywords and follow behavior.

Q4: Do users' following behaviors tend to expose them to spammers? Empirical Answer: Yes. As seen in Figure 5.6(b), similar to tweet topics, the values of CR for following verified accounts related with the topics of Entertainment (`FEn`), Sports

(FSp), Economics (FEc), Expertise (FEx), are all higher than that of T1d and Default. This observation implies that the behavior of following those famous ("verified") accounts could be utilized by spammers to find their targets. As a case study, we find one spam account, which mainly posts spam about TV media (the URLs in the tweets have been identified as suspicious by the URL shortening service), shares 19 followings (most of them are related the topic of art or TV media) with one honeypot.

Q5: Do accounts with the usage of social apps tend to be contacted by more spammers? Empirical Answer: No. According to our data, the capture rates of honeypots with the usage of Instagram, Twitpic and Twiends, are 0.008, 0.008, and 0.009, respectively. These values are lower than that of most of other social patterns and similar to *Default*. Thus, using social apps do not help much in terms of attracting spammers. This might either because spammers have not use this strategy to find targets or the selections of applications used by our honeypots are not representative.

According to the above analyses, we can find that many spammers indeed selectively choose spamming targets, rather than random selections. By doing this, spammers can increase the chance of success, while avoiding being suspended due to excessive contacts with others.

5.2.2.3 Guidelines for Designing Effective Honeypots

According to the above analyses, we could summarize the following guidelines for designing more effective social honeypots to trap Twitter spammers: **(1) post tweets related with specific topics; (2) post tweets containing special keywords such as Trending topics; (3) follow famous accounts related with specific areas.**

To evaluate the effectiveness of those guidelines, we denote *96 "benchmark" hon-*

eypots as GE, and 51 honeypots of them[2] that *meet at least one guideline* as GU. We find that GU's capture rate (0.083) is over two times as that of GE (0.040). This observation indicates that GU (that meet guidelines) is more effective to attract spammers than GE.

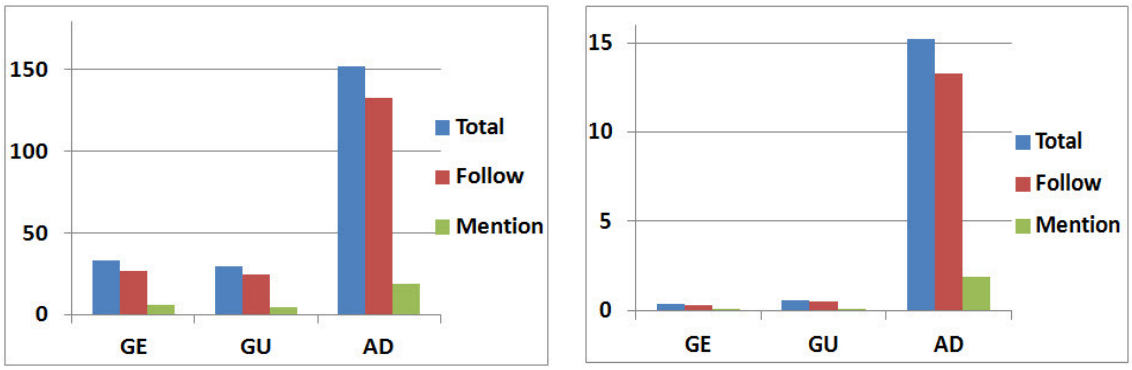
To further evaluate the effectiveness of our guidelines, we deploy another 10 "advanced" honeypots (AD) with more guided social behaviors for a week right after finishing the 5-month running of "benchmark" honeypots. Specifically, in each day, each of them will behave the following social patterns[3]: (1) post one topic tweet per hour related with each of those four topics; (2) post one tweet containing one trending topic per hour; (3) post one tweet containing one arbitrary hashtag per hour; (4) post one tweet containing one bait word per hour; (5) Follow 5 experts related with each of four topics per day. Then, as seen in Figure 5.7, we compare the performance of AD with GE and GU by collecting data in the same week. We can find that AD is much more effective than GE and GU in trapping more spammers. Particularly, AD's capture rate (2.17) is 25.5 times as that of GU (0.085), and 45.2 times as that of GE (0.048). Although this comparison result may contain some bias due to a relatively short period time of data collection, such a huge difference could still validate the effectiveness of our guidelines for designing better social honeypots.

5.2.2.4 Extracting Spammers' Interested Topic Terms

To better understand spammers' tastes, it is meaningful to extract those specific key terms, which usually contain semantic meanings and tend to be used by spammers to find targets. Even though this could be achieved by manually analyzing those hashtags in the tweets, a more generic and automated approach to extract key

[2]51 accounts are 16-25, 31-35, 41-76 as labeled in Table 5.1.

[3]To prevent spammers identifying our accounts as honeypots based on the temporal patterns, some random delays are inserted before posting each tweet.



(a) # of Spammers (b) per Honeypot

Figure 5.7: The effectiveness of advanced honeypots.

terms (not limited to hashtags only) is still needed, because a large amount of tweets do not contain hashtags. Thus, in this section, we design an approach to extract those topic terms through analyzing the data collected by our honeypots.

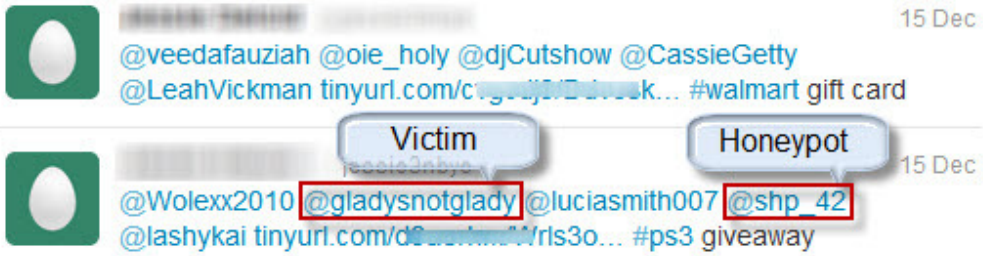


Figure 5.8: One real case study of potential victims.

We first introduce the intuition of our algorithm. As shown in Figure 5.8, we find that many spammers send illicit "@mentions" not only to our honeypots but also to other accounts (e.g., "gladynotglady" in this example), which denoted as "potential victims" in this work. We denote each pair of potential victims and honeypots in one illicit "@mention" as a "victim relationship". Thus, we believe that there should be

some common social behavior patterns between our honeypots and those potential victims in each victim relationship, which essentially incur spammers' contacts. An intuitive method is to extract the common terms used by both our honeypots and those victims, which may represent spammers' tastes. However, this approach will extract many widely-used common words, which are not representative for spammers' tastes. Even though we could use a big stop-word list to filter some common words, it could not help much to achieve this goal, because many words tweeted by users are not even spelled completely or in a standard form.

Thus, we first use the Latent Dirichlet Allocation (LDA)[4] [127] algorithm to extract topic terms, which are better to represent semantic topics, from the tweets posted by our honeypots and potential victims. Then, we output those topic terms that are highly/frequently shared by the pairs of victims and honeypots. In this way, these topic terms may highly attract spammers' contact.

Due to the page limitation, we next briefly introduce our algorithm to extract topic terms based on LDA instead of presenting its details. For each honeypot $\{SHP_i|i = 1, 2, ..., I\}$, we extract its received "@mention" messages $\{M_{im}|m = 1, 2, ..., M\}$ sent from spam accounts. If the message contains other potential victims $\{PV_j|j = 1, 2, ..., J\}$, we put $\{SHP_i$ and $\{PV_j|j = 1, 2, ..., J\}$ into an account set SMA, and record each victim relationship $vr = (SHP_i, PV_j)$ into a victim relationship set VR. Then, for each unique account $SMA_k \in SMA$, we extract its NT_k top ranked topic terms by using LDA. Next, for each relationship $vr \in VR$, we extract the shared topic terms among the honeypot and the potential victim, and save them into a semantic topic term set ST. Then, we output the most frequent FT topic terms shown in all victim relationships as spammers' interesting topic terms, because

[4]LDA is a generative probabilistic topic modeling (clustering) algorithm, which could cluster terms in the large volumes of unlabeled text into several semantic topics by identifying the latent topics words in the text.

these terms with strong semantic topic meanings are highly shared among our social honeypots and their corresponding potential victims.

Specifically, our honeypots receive 449 "@mention" messages from spammers and form 5,716 victim relationships with 275 unique potential victims. Then, we extract $1,500$ and 600 topic terms for each honeypot and potential victim, respectively. We finally output the top 500 as semantic topic terms. (Due to the page limitation, we skip to show those topic terms.) Furthermore, through extracting semantic topic terms, we could examine whether there is tweet similarity between spammers and their targets. Particularly, we extract semantic topic terms for 278 (manually) identified spam accounts by using LDA. Then, we extract all pairs of honeypots and spam accounts, if the spam accounts either follow or "@mention" the honeypot. Then, we find 81.69% of 360 pairs of two accounts share at least one semantic term. This observation indicates a relatively strong semantic similarity between spammers and their targets.

5.2.2.5 Ethical Considerations

The technology of deploying social honeypots on real OSNs may raise ethical considerations: whether such social honeypots will generate big effects to OSNs. In terms of our work, both "benchmark" and "advanced" honeypots are designed to neither send any messages to other users nor post any URLs/spam in the tweets. Also, those accounts only follow a relatively small number (several hundreds) of "verified" accounts. Thus, we believe our designed honeypots will generate very limited effects to other normal users. In addition, the technique of social honeypots has been commonly used to capture spammers [102, 65] or to understand the security vulnerability [14] on OSNs. Furthermore, as advocated in a recent study on the ethics of security vulnerability [74], such studies served as social functions are neither

unethical nor illegal.

5.3 Prioritizing the Sampling of Likely Spammers

In this section, we design two guided approaches to actively sample more likely Twitter spam accounts from Twittersphere, based on the observation that many spammers find their targets based on targets' social behaviors.

5.3.1 Motivation

The collection of spam accounts is usually the first step to analyze spammers' behaviors and to further generate defense insights. However, given the limited time/resource (especially for academic researchers), it is not trivial to collect a large-scale of spam accounts in the huge Twittersphere. Existing studies mainly rely on the following three strategies to collect (likely) spam accounts: implementing social honeypots[65, 66, 102], collecting suspended accounts [105, 57], and manual identification [65, 116]. However, all these three strategies have certain limitations. The honeypot approach is a *passive* one, requiring time (and luck) to wait for spammers' contacts. Collecting suspended accounts requires to develop a robust crawler and takes a considerable long time (typically several months) to crawl Twitter and to wait for collected accounts to be suspended by Twitter. Manual identification could achieves a high accuracy, which requires tedious human work and is not scalable.

Motivated by the limitations of existing strategies to collect (likely) spam accounts, we design two lightweight, guided strategies (called samplers in this dissertation) to prioritize the active sampling of more likely spam accounts in Twittersphere: *Hashtag Sampler and Friend Sampler*. These two samplers are designed to be able to *efficiently* collect/sample a considerable number of targeted social-media spammers (spam accounts) in a short time period with a relatively high hit rate.

5.3.2 Hashtag Sampler

5.3.2.1 Basic Intuition

Spammers tend to follow those accounts that post spammers' interested keywords (hashtags). According to this intuition, an account might be suspicious if it follows many accounts that share some spammers' interested hashtags in their tweets. At the high level, Hashtag sampler is designed to preferentially sample likely spam accounts through checking *common followers* of multiple accounts that share/post *similar, multiple* hashtags as spammers do.

5.3.2.2 Detailed Strategy

As illustrated in Figure 5.9, Hashtag Sampler has three steps to sample likely spam accounts from Twitter: (1) collecting spammers' hashtags; (2) searching potential spammers' targets; (3) sampling suspicious hashtag followers.

Particularly, in Step 1, Hashtag Sampler collects keywords/hashtages that spammers are potentially interested in (i.e., hashtags in spam accounts' tweets) through identifying hashtags ("#") from tweets posted by our trapped spam accounts; in Step 2, for each hashtag, Hashtag Sampler searches the recent M tweets[5] that contain hashtags, through exactly querying the hashtag from Twitter Search. Then, we consider an account to be a potential spammers' *target*, if they send tweets containing that particular hashtag. Accordingly, through extracting the senders of those tweets, Hashtag Sampler searches out all the potential targets; in Step 3, for each potential spammers' target, we could obtain its followers through using Twitter API. After extracting all targets' followers, we denote those followers with high occurrences as *suspicious hashtag followers*. These hashtag followers essentially follow many other

[5]For each query term, Twitter limits to return 1,500 tweets as a maximal. Thus, in our experiment, we set $M = 1,500$

accounts, who post that spammers' hashtag. Finally, Hashtag Sampler outputs those accounts as spam accounts, if they are sampled as suspicious hashtag followers with the usage of multiple different hashtags, i.e., they are considered as suspicious hashtag followers with a high occurrence by using different hashtags.

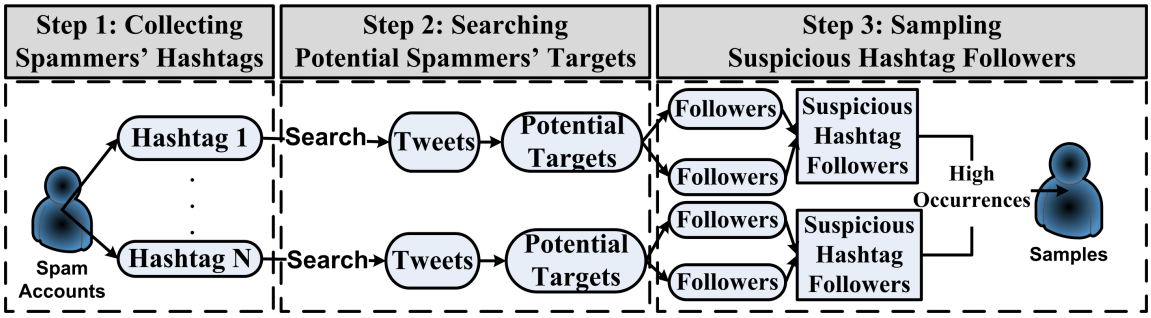


Figure 5.9: Illustration of Hashtag Sampler.

5.3.3 Friend Sampler

5.3.3.1 Basic Intuition

Spammers tend to select famous accounts' followers as their targets. In fact, those Twitter accounts (especially famous accounts) followed by a user could also reveal this user's taste, which could be utilized by spammers to find their potential spamming targets. According to this intuition, Friend Sampler is designed to preferentially sample likely spam accounts through checking those accounts that excessively follow multiple famous accounts' followers, i.e., examining *common followers of the followers* of some famous accounts.

5.3.3.2 Detailed Strategy

As illustrated in Figure 5.10, Friend Sampler first randomly selects M verified (famous) accounts from those 400 verified accounts used in Section 5.2. Then, for

each account, Friend Sampler collects its N followers (if available), which could be considered as spammers' potential targets. Then, we further examine extracted followers of those potential targets, and save them in a dataset with their numbers of occurrences, denoted as suspicious account set.(For example, if an account follows two potential targets, its number of occurrences is 2.) Finally, Friend Sampler outputs N_{fd} accounts in the suspicious account set as spammers with the top numbers of occurrences.

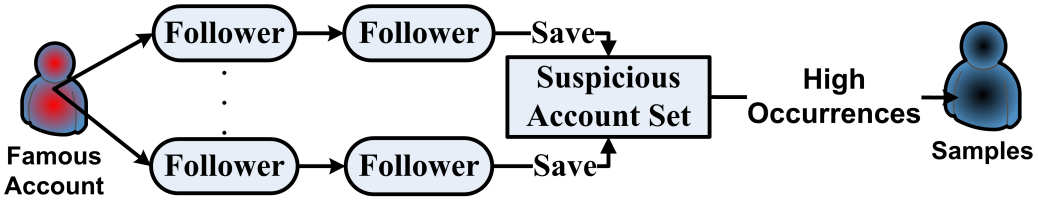


Figure 5.10: Illustration of Friends Sampler.

5.4 Evaluation of Samplers

In this section, we mainly describe our evaluation methodologies and evaluation results for two samplers in selectively sampling likely spam accounts.

5.4.1 Ground Truth and Evaluation Metrics

5.4.1.1 Ground Truth

To evaluate the effectiveness of two samplers, we require a ground truth for those accounts collected by our samplers. However, as a common challenge for all OSN data analysis work, it is difficult to obtain perfect ground truth for a large-scale dataset.

It is straightforward that an account can be considered as spam if it is sus-

pended by Twitter. However, only considering suspend accounts as spam accounts will miss many other spam accounts, which have not been identified/suspended by Twitter. Thus, for the rest *unsuspended* accounts output by our samplers, we rely on a state-of-the-art machine-learning classifier to further examine whether they are spam accounts. This classifier is implemented based on `Random Forest` and uses the same feature set designed in [65]. Then, the classifier is trained by using 2,000 suspended accounts and 20,000 normal accounts (none of them post malicious URLs). The accuracy and false positive rate of this classifier is 99.2% and 0.97% respectively based on the training datasets through *10-fold cross validation* tests. Note that we use the machine-learning technique to help to estimate the ground truth rather than to detect spam accounts. Also, we acknowledge that any machine learning classifier may not be absolutely accurate (especially it may induce some false positives). However in our evaluation, we are mainly interested in getting the *estimation* of the accuracy, instead of absolute values. Furthermore, such a strategy is a common practice for similar studies on accuracy estimation of large scale unlabeled datasets [124].

5.4.1.2 Evaluation Metrics

To measure the effectiveness of sampling strategies with the goal of collecting more likely spam accounts, two metrics are typically considered. The number of collected spam accounts, denoted in our work as "Hit Count (N_{hit})"; and the ratio of Hit Count to the total number of sampled accounts (N_{sample}), denoted as "Hit Ratio (H_r)". Thus, a higher value of Hit Count and Hit Ratio indicates that we can catch more spam accounts and more accurately, respectively. Motivated by the limitations of traditional ways of collecting spam accounts as described in Section 5.3, our two samplers are designed as lightweight, guided strategies to efficiently and effectively

prioritize the sampling of more likely spam accounts instead of(otherwise) analyzing all accounts in the huge Twittersphere. Thus, our two samplers are *not* designed to find/uncover all types of spam accounts, and thus should *not* be considered as spammer detectors. Accordingly, we use those two evaluation metrics (N_{hit}, H_r) instead of false positives/negatives in our evaluation. Particularly, many existing studies [138, 55] similarly use these two metrics to measure the effectiveness by outputting the number of hits in a top list.

With such notions, if we denote the number of suspended accounts as N_{sus} and the number of spam accounts output by the machine-learning classifier as N_{mal}, we could calculate Hit Count and Hit Ratio as follows[6]: $N_{hit} = N_{sus}+N_{mal}$; $H_r = N_{hit}/N_{sample}$.

5.4.2 Implementation

To implement Hashtag Sampler, we use 3,246 unique hashtags/keywords posted by 278 identified spammers. For each hashtag, Hashtag Sampler outputs $SF = 500$ (if available) suspicious hashtag followers. By using each spam account's hashtags, Hashtag Sampler samples $M = 500$ suspicious hashtag followers (if available) with the top occurrences as spammers. To implement Hashtag Sampler, we randomly select $M = 40$ verified (famous) accounts (10 accounts for each of four topics). For each verified account, we examine its $N = 5,000$ followers, which are retrieved by sending one "get-follower" request to Twitter. Then, for each follower, Friend Sampler continues to examine its followers, and samples $N_{fd} = 1,000$ top ranked accounts as spam accounts. Using these implementation parameters, we run our two samplers for four days to sample more likely spam accounts. After one month, we further examine whether those sampled accounts are suspended by Twitter.

[6]Since we could not obtain ground truth for those protected and nonexistent accounts output by our samplers, we do not count such accounts in N_{sample}.

5.4.3 Effectiveness of Hashtag Sampler

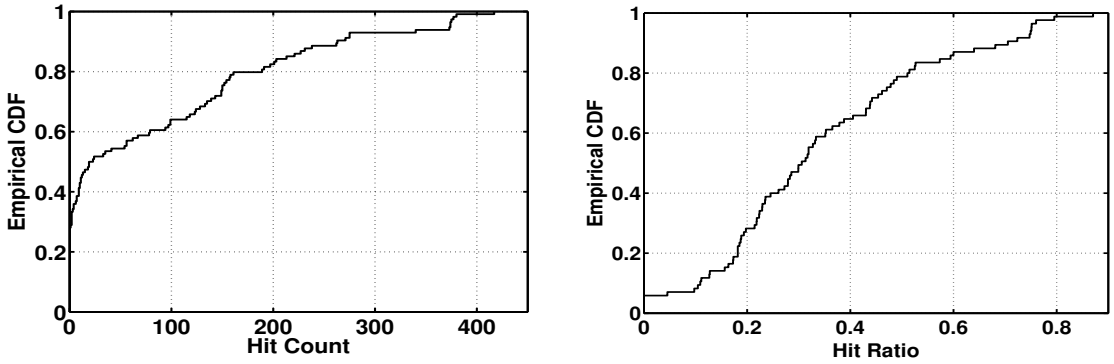
As seen in Table 5.2, Hashtag Sampler outputs 8,983 unique accounts to be likely spam accounts. Among them, 262 accounts have been suspended, and 4,665 others are output as spam accounts by the classifier. Thus, the hit count is 4,927 and the hit ratio is 0.5489, which implies that Hashtag Sampler could correctly collect one spam account by sampling less than two accounts.

Item	N_{sus}	N_{mal}	N_{hit}	N_{sample}	H_r
Value	262	4,665	4,927	8,983	0.5489

Table 5.2: The effectiveness of Hashtag Sampler.

Also, we further examine hit count and hit ratio by using each spammer's hashtags. As seen in Figure 5.11(a), over 40% spam accounts' hashtags can be used to collect over 100 spam accounts by sampling 500 accounts. This observation shows that Hashtag Sampler can effectively collect spam accounts by focusing on spammers' tastes. Also, we find that around 30% spammers' hashtags can not be used to correctly collect spam accounts. The reason is mainly because Twitter Search does not index every tweet, due to its resource constraints. According to our observation, we could crawl very few (or even no) tweets by using those spammers' hashtags.

As seen in Figure 5.11(b), Hashtag Sampler could obtain reasonable hit ratios by using around 60% spammers' hashtags, which are higher than 0.3 (sampling 3 accounts will correctly collect 1 spam account).



(a) Hit Count (b) Hit Ratio

Figure 5.11: Collection results of Hashtag Sampler by using individual spammers' hashtags.

5.4.4 Effectiveness of Friend Sampler

As seen in Table 5.3, Friend Sampler outputs 21,686 unique accounts to be likely spammers. Among these accounts, 4,000 have been suspended, and 9,781 others are output as spam accounts by the classifier. Thus, the hit count is 13,781 and hit ratio is 0.6355. According to our evaluation, over 50% famous accounts used by Friend Sampler could achieve a hit ratio higher than 0.5.

Item	N_{sus}	N_{mal}	N_{hit}	N_{sample}	H_r
Value	4,000	9,781	13,781	21,686	0.6355

Table 5.3: The effectiveness of Friend Sampler.

5.4.5 Diversity and Complementarity

Next, we analyze the diversity and complementarity of using these two samplers. Essentially, between these two algorithms, we examine the number of spam accounts correctly sampled only by one algorithm, which can not be found by using the other

one. If this number of each algorithm is high, it implies that these two samplers are very complementary. Thus they could be combined together to find more spam accounts. Specifically, to measure the diversity, we design a metric, named "Exclusive Ratio (E_r)", which is the ratio of the number of spam accounts that are exclusively sampled by one sampler (not sampled by the other one) to the total number of spam accounts sampled by this sampler.

As seen in Table 5.4, We can find that both two samplers can obtain relatively high exclusive ratios (over 77%). The ratio for Friend Sampler is even higher than 90%. This observation shows that these two samplers are indeed complementary. And thus, they can be used together to collect more (likely) spam accounts.

Algorithm	**Hashtag Sampler**	**Friend Sampler**
Exclusive Ratio	77.69%	90.57%

Table 5.4: Exclusive ratios between two samplers.

Particularly, as shown in Table 5.5, according to our collected dataset, the combination usage of these two algorithms could correctly collect 17,416 spam accounts.

Item	N_{sus}	N_{mal}	N_{hit}	N_{sample}	H_r
Value	4,249	13,936	18,185	29,239	0.6219

Table 5.5: Result of combining two algorithms.

Among them, 3,480 accounts have been suspended by Twitter, and 13,936 other accounts are classified as spam accounts by our classifier. Thus, the hit ratio of combining these two algorithms is 0.6023. Compared with the dataset used for the

purpose of building an effective machine learning classifier [12], which contains 355 manually identified spam accounts from 8,207 randomly crawled accounts (i.e., a hit ratio of only 0.04), this value of hit ratio is considerably high in terms of effectively crawling likely spam accounts.

5.4.6 Comparison with Existing Strategies

As the research motivation described in Section 5.3, we next compare the efficiency and effectiveness of using our two samplers with existing strategies to collect spam accounts.

5.4.6.1 Our Samplers VS Collecting Suspended Accounts

To compare the strategy to collect spam accounts by collecting suspended accounts from a pre-crawled dataset, we calculate the number of suspended accounts from those 1.2 million accounts. Whether an account is suspended can be automatically known by issuing a query to Twitter. Accordingly, we get 14,226 suspended accounts from those 1.2 million accounts, and achieve a hit ratio at 1.19%, which is the number of suspended accounts to the total number of crawled accounts. However, our two samplers have a much higher hit ratio of 14.53% in terms of suspended accounts only (more than 10 times higher). In terms of the speed, it only takes about four days for our samplers to collect 4,249 suspended accounts, while to collect 14,226 suspended accounts, it takes around 7 months to crawl the dataset (not to mention the long time lag to wait for Twitter to suspend the accounts). The advantage of our samplers in terms of effectiveness and efficiency is clear.

5.4.6.2 Our Samplers VS Social Honeypots

We next compare our samplers with two existing social honeypot studies. As shown in Table 5.6, the honeypots used in [65] collected 500 spam accounts in one

month; the honeypots used in [102] collected 11,699 spam accounts in 11 months.

Strategy	[65]	[102]	**Samplers**
Time	1 month	11 months	4 days
# of Spammers	500	11,699	18,185

Table 5.6: Comparison with existing social honeypots.

Since different studies use different numbers of social honeypots, and collect spam accounts at different time periods with different definitions of the spam account, we do not intend to conclude which honeypots are better. However, we could easily see that compared with the strategy of using (passive) social honeypots, our samplers can *actively* collect likely spam accounts much more *efficiently*.

5.5 Limitation

We acknowledge our manually identified spam accounts may contain some bias, and our machine-learning classifier may not be absolutely accurate. However, it is challenging to obtain a perfect ground truth, and our strategies have been widely used in this line of studies [123, 27, 65, 124]. In addition, even though some values may vary according to different datasets, we believe that our major findings and insights are still valid independent of the datasets.

It is possible that our advanced honeypots may also attract a few benign accounts' contacts. However, this highly depends on the goal of honeypots – trapping more spam accounts or obtaining spam accounts only, for which we believe the former is more important. According to our data collection results, our advanced honeypots could trap *significantly more* spam accounts.

We note that our samplers are not designed to collect/cover all (types of) spam-

mers in Twittersphere. In addition, we note that the number of collected spam accounts by our samplers is restricted by the number of inputs, e.g., hashtags and famous accounts. Our result is also limited by Twitter Search API: one request could only obtain the recent 1,500 search results, and not even to mention not all tweets are indexed by Twitter Search. Thus, if our samplers are implemented by Twitter (without many restrictions), they could find more spam accounts.

5.6 Summary

While spam accounts become more evasive and evolve to launch target attacks, in this chapter, we performed a deep measurement study on how some Twitter spammers choose their spamming targets, by building social honeypots with diverse social behavioral patterns. We provided principled guidelines for building more effective (attractive) social honeypots, based on the intuitions that the accounts tend to attract more spammers' contact, if they post tweets related with specific topics, or post tweets containing special keywords such as Trending topics, or follow famous accounts related with specific areas. We designed two light-weight and effective samplers to guide the sampling of more likely Twitter spam accounts by reverse engineering spammers' tastes of finding their spamming targets. We reported an experimental evaluation of our two samplers by actively crawling online data from Twitter, and showed that our two samplers can be effectively and complementarily used to actively find spam accounts.

6. UNDERSTANDING ANDROID MALWARE ECOSYSTEM

Similar to the analysis of the cyber criminal ecosystem on OSN platform, to better understand how Android malware authors spread malicious Android apps, this chapter presents an in-depth analysis of the market-level and network-level behaviors of the underground Android malware ecosystem.

We empirically perform the first comprehensive measurement study on analyzing the market-level and network-level behaviors of the Android malware ecosystem. We have crawled and analyzed over 82,000 Android apps[1] and 28,000 Android market accounts[2] from multiple representative markets (including both official GooglePlay [48] and third-party ones such as SlideMe [100] from USA, Anzhi [70] from China, and Tapp [104] from Russia). After further analysis, we obtain a dataset of over 9,700 malicious Android apps, and another dataset of over 3,500 malicious market accounts that distribute at least one malicious app to the market. To facilitate the analysis of the network-level behaviors, we extract networking attempts made by Android apps, through running them in a customized Android runtime environment and developing a UI fuzzing tool to add random UI events. In total, we obtain over 239,000 unique URLs leading to over 25,000 unique remote servers.

In the phase of analyzing the market-level behaviors, we investigate whether there are any special characteristics of those market accounts that distribute malware. We investigate whether specific metrics, such as the location of the market and the popularity of the app, are effective indications to the quality of Android apps. In particular, we investigate whether malicious accounts have specific temporal behavioral patterns in submitting malware samples. In the phase of analyzing

[1]Each app is uniquely counted by its value of MD5.
[2]Each market account is uniquely counted by the author name registered in the Android market.

the network-level behaviors, we investigate whether there are any special networks mainly utilized by Android malware authors to host their remote servers and whether there are any large communities among Android malware.

We propose a lightweight algorithm to infer malicious apps based on our analysis of the market-level and network-level behaviors of Android malware ecosystem. Our inference algorithm is positioned as a *complementary*, lightweight strategy to quickly find those more suspicious apps.

We make the following contributions:

- We present an in-depth look at the market-level and network-level behaviors of the Android malware ecosystem, based on a detailed analysis of over 9,700 malicious apps, collected from a large corpus of over 82,000 Android apps from multiple markets.

- Through analyzing the **market-level behaviors**, we find that: (1) Neither the location of the market nor the popularity of the apps has a strong correlation with the quality of the apps; (2) The public Android anti-virus blacklist is too slow at identifying new Android malware, allowing around 90% of malicious apps submitted to the markets before they are seen by the blacklist; (3) The same malware authors tend to submit multiple malicious apps, and within a short time period. This represents an interesting spatial-temporal behavioral pattern.

- Through analyzing the **network-level behaviors**, we find that: (1) There is a strong provider locality property in the Android malware's remote servers hosting infrastructure; Android malware authors tend to use cloud vendors to host remote servers to communicate with their malware samples; (2) Existing IP/domain blacklists are not effective to be used to find Android malware;

(3) A few malware communities (sharing common authors or remote servers) contribute to a large portion of Android malware.

- We design a novel algorithm (`AMIA`) to infer more malicious apps by exploiting their community relationships. AMIA is designed by exploiting the properties of the community relationships among Android malware, which requires neither the disassembling of Android apps nor the deep domain knowledge of the Android system. By using a small seed set of known malicious apps, AMIA can effectively find another extra 20 times of malicious apps, while maintaining a considerable accuracy higher than 94%.

In Section 6.1, we introduce the problem background and the overview of our analysis. In Section 6.2, we present our data collection methodology. We detail the procedure of our analysis of market-level behaviors in Section 6.3 and network-level behaviors in Section 6.4. In Section 6.5, we describe the design and evaluation results of our inference algorithm to sample more likely malicious Android apps. We discuss our limitation in Section 6.6.

6.1 Background and Overview

In this section, we first introduce the major actions taken by Android malware authors to spread their malware to achieve malicious goals, and then describe the overview of our analysis.

6.1.1 Background

Before the time when a victim accesses the Android market to install a malicious Android app on his smartphone, an Android malware author typically need to take a sequence of actions to underpin a successful download. We present the flow of actions taken by Android malware authors to spread their malware in Figure 6.1.

The flow begins with Android malware authors developing Android malware (①). To achieve malicious goals such as compromising victims' privacy and remotely hijacking victims' phones, malware authors typically need to build remote servers (②) to communicate with (or control) their malware samples. Next, malware authors require to register Android market accounts (③) to launch malware on specific Android markets (④). The registration of the market accounts typically requires developers to use valid email accounts or even paying registration fee (e.g., in GooglePlay). After successfully attracting victims' attention and obtaining their trust, the malware will be downloaded and further installed on victims' smartphones (⑤, ⑥). Once the victims' phones are infected, the malware typically communicate with the remote servers, to send out private/system information (⑦), or even to further receive instructions to communicate with other remote servers (⑧). Once victims' phones are fully controlled by the malware, any variety of other malware can be installed. Finally, malware authors obtain profits by selling victims' sensitive data or stealthily charging victims' mobile bills (⑨).

Throughout this process, we can clearly see that after developing malware, Android malware authors typically require two types of behaviors to lure their victims and obtain profits from the victims: utilizing Android markets to spread malware and building remote servers to communicate with malware. Thus, our research goal is to provide the first empirical analysis of the characteristics of the market-level and network-level behaviors of the Android malware ecosystem, and provide new defense insights against Android malware.

6.1.2 Analysis Overview

To achieve our research goals, as illustrated in Figure 6.2, our analysis procedure contains three major steps: collecting data, empirically analyzing market-level and

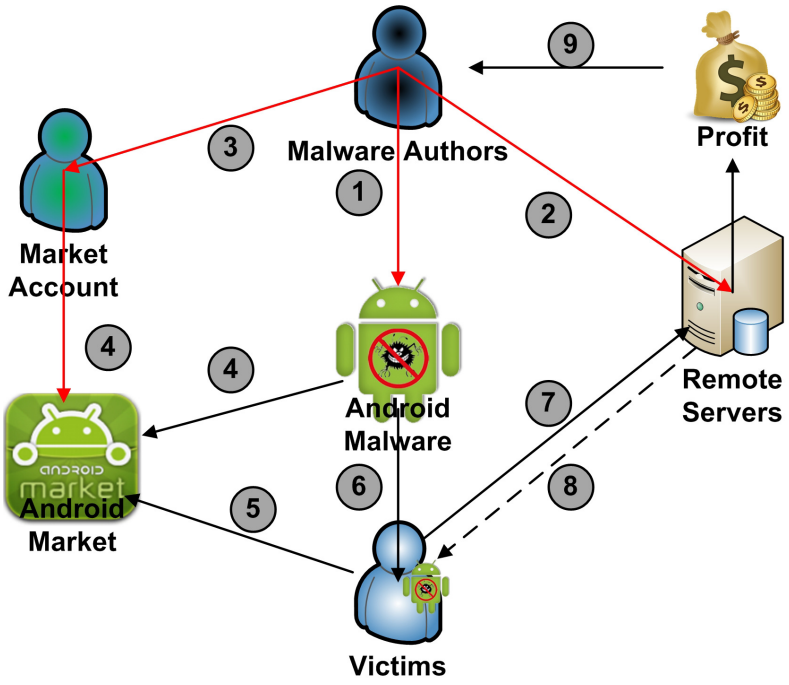


Figure 6.1: The flow of actions taken by Android malware authors to spread Android malware.

network-level behaviors, and generating defense insights.

In the data collection step, besides crawling Android apps, our crawler also collects those apps' corresponding market information (e.g., author, downloading number, and submission time) from the official Android market (GooglePlay) and three representative third-party Android markets (SlideMe, Anzhi, and Tapp). Then, we identify Android malware from our crawled dataset, and extract the remote servers (IP addresses and domains) visited by crawled Android apps.

In the phase of analyzing the market-level behaviors, we first uncover whether the apps that are hosted in American markets or highly downloaded are more trustable; we next measure the effectiveness of using the Android malware blacklist to stop malware authors submitting their malware samples to the markets; we also examine which app categories Android malware tend to masquerade themselves to belong

to; we finally uncover the behavioral characteristics of those malicious accounts that submit malware to the markets.

In the phase of analyzing the network-level behaviors, we mainly analyze the IP address spaces, special networks, and cloud hosting services used by Android malware; we also examine the effectiveness of using existing IP/domain blacklists to find Android malware; we finally extract and measure Android malware communities.

Finally, spurred by the above analysis, we design a new algorithm to infer more malicious Android apps.

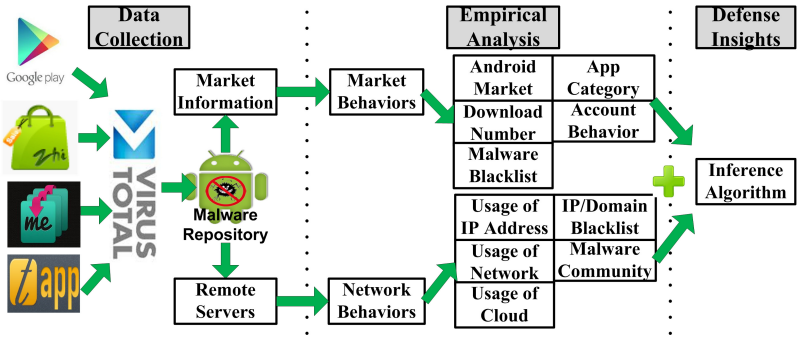


Figure 6.2: The analysis overview.

6.2 Data Collection

6.2.1 Crawling Android Apps

We crawled Android apps from four representative Android markets: the official Android market (*GooglePlay*) announced in 2008 and three third-party Android markets (*SlideMe* from USA created in 2008, *Anzhi* from China created in 2010, and *Tapp* from Russia created in 2012). The crawling process for GooglePlay was harvested during a two-months period, from August 23rd, 2012 through October 23rd, 2012. The crawling process for those third-party markets was mainly achieved from

June 3rd, 2012 to June 15th, 2012. During the crawling process, besides downloading Android apps, our crawler also recorded those apps' market information (e.g., author, submission time, downloading number, and app category). Our crawler downloaded all free apps that were available in the third-party markets at the time when the crawler was launched. Due to the crawling rate limit and the large amount of apps in GooglePlay, our dataset of official apps were randomly sampled from all 33 app categories in GooglePlay. As summarized in Table 6.1, in total, we collected 82,966 free Android apps, where around 22% of the apps (18,751) were collected from GooglePlay, and the remaining 78% (65,232) were harvested from the third-party markets.

	GooglePlay	**SlideMe**	**Anzhi**	**Tapp**
Location	U.S.A	U.S.A	China	Russia
Creation Time	2008	2008	2010	2012
Number of Unique Apps	18,751	15,109	38,458	11,822
Total (Unique)	18,751 (22%)	65,232 (78%)		
	82,966			

Table 6.1: Summary of crawling Android apps.

6.2.2 Identifying Android Malware

Next, we identified malicious apps from our Android app corpus by searching their values of MD5 to VirusTotal [114], which is a free anti-virus blacklist service providing the scanning reports from over 40 different anti-virus products. For each app, if it has been seen by VirusTotal, we obtained its full scanning report, which includes the first and the last time the app was seen, as well as the results from each individual virus scan. We consider an app to be malicious, if it is labeled as

malware by at lease one anti-virus product. It is also worth noting that we searched our crawled apps' reports from VirusTotal on Oct. 11th, 2013, over one year after we finished crawling those apps, in order to give enough time for those commercial anti-virus tools to have updated signatures and achieve more accurate scanning results of those apps.

As seen in Table 6.2, we finally obtained 9,712 unique malicious apps, where around 16% of them (1,593) were collected from GooglePlay, and around 84% (8,229) were harvested from three third-party markets. We term this dataset of malicious apps as `MalApps`. Apart from the dataset of 9,956 adware (`AdwareApps`), we term the dataset of the rest 63,298 apps as `RestApps`. Note that since a few antivirus tools consider those non-malicious apps that use certain advertisement libraries as adware, to better guarantee the accuracy of our measurement results, we distinguish the adware from those truly malicious apps.

	GooglePlay	**SlideMe**	**Anzhi**	**Tapp**
MalApps	1,593	1,946	4,840	1,450
Total(Unique)	1,593 (16%)	8,229 (84%)		
	9,712			
AdwareApps	1,037	1,247	6,764	994
Total(Unique)	1,037 (12%)	8,977 (88%)		
	9,956			
RestApps	16,121	11,916	26,854	9,378
Total(Unique)	16,121 (29%)	38,726(71%)		
	63,298			

Table 6.2: Summary of collecting Android malware.

Similar to other measurement studies, our analyzed dataset may contain some bias or noise. For example, there could be sampling bias in our crawling. To reduce

possible data sampling bias, we have crawled several representative large Android app markets (instead of one) in difficult countries, and we crawled all the apps that are hosted in the third-party markets and randomly sample apps from all 33 app categories in GooglePlay. It is also true that even we use VirusTotal, a state-of-the-art anti-virus service that combines the scanning reports from over 40 commercial Anti-Virus products, it is still possible that a small number of apps in `MalApps` might be actually benign, and `RestApps` might still contain a few malicious apps. Essentially, it is extremely challenging to obtain an ideal, unbiased dataset with perfect ground truth for a large-scale dataset of Android apps. We believe that even though the exact values of some metrics reported in our work may vary a little bit when using different sample datasets or ground truths, our major conclusions and insights obtained in our analysis will likely still hold.

6.3 Analyzing Market-level Behaviors

Different from desktop malware authors, who typically have to build their own platforms/websites to spread malware, Android malware authors can spread malware more effectively by utilizing popular Android markets. However, no existing studies have been done to deeply analyze how malware authors utilize those Android markets. In this section, we first describe our results of collecting market accounts, and then provide our detailed analysis of the market-level behaviors in a question-and-answer fashion.

6.3.1 Collecting Market Accounts

To facilitate the analysis of the market-level behaviors, we collected market accounts (uniquely identified by the author name) from those three representative markets. Next, we extracted malicious accounts that at least submitted one malicious app. As seen in Table 6.3, we collected 28,496 market accounts, where 35%

of the accounts (10,064) were collected from GooglePlay, and the remaining 65% (18,432) were harvested from three third-party markets; 3,517 of those 28,496 market accounts are identified as malicious, where 25% of the accounts (883) were from GooglePlay, and the remaining 75% (2,634) were from three third-party markets.

	GooglePlay	**SlideMe**	**Anzhi**	**Tapp**
# of Accounts	10,064	3,896	9,665	4,871
Total (Unique)	10,064 (35%)	18,432 (65%)		
	28,496			
# of Malicoius Accounts	883	432	1,493	709
Total(Unique)	883 (25%)	2,634 (75%)		
	3,517			

Table 6.3: Summary of collecting market accounts.

6.3.2 Detailed Analysis

Since the official market (GooglePlay) is located in America, many users prefer to choose apps from GooglePlay or other American markets. Accordingly, our first market-level analysis is to examine the relationship between the app quality and the app market.

6.3.2.1 App Quality VS App Market

Question 1: Are the apps from American markets more trustable? Our Empirical Answer: No. The quality of the American third-party market is not better than that of the Chinese one, where malicious apps have been reported to be widely hosted in. Also, although the quality of the official market is a little better than those third-party markets as we expect, it still has much room to improve.

To answer this question, for each market, we calculate its percentage of malicious

apps in the total apps crawled from the market, termed as Market Malicious Ratio. Accordingly, the lower ratio one market has, the better quality it is. As seen in Table 6.4, the ratio of GooglePlay, which is around 8.5%, is lower than that of all three third-party markets, which are 12.87%, 12.58%, and 12.27%, respectively.

	GooglePlay	**SlideMe**	**Anzhi**	**Tapp**
Location	U.S.A	U.S.A	China	Russian
Market Malicious Ratio	8.50%	12.87%	12.58%	12.27%

Table 6.4: The comparsion of market quality.

This observation is mainly because the official market has more strict security rules (e.g., Google Bouncer [71]) to better guarantee the quality of the apps than those third-party markets, as we expect. However, the difference of the ratio between GooglePlay and those third-party markets is not that significant. This observation implies that there is still a large space for both the official market and third-party markets to improve their quality.

In addition, the ratio for the American third-party market (SlideMe) is even slightly higher than that of the Chinese one (Anzhi). Although the quality of the Chinese market is not that good due to the fact that Chinese customers can not access GooglePlay and have to rely on third-party markets to download Android apps, which motivates many malware authors to spread Android malware in Chinese markets, the quality of the American third-party market is not necessarily better. This observation implies that apps that are hosted in the American markets are not necessarily more trustable.

6.3.2.2 Effectiveness of Android Antivirus Blacklist

As an important security aspect, besides analyzing the detection rate of the blacklists [68, 141], existing studies have also analyzed the detection timelag of the blacklists in different security scenarios (e.g., social network spam study [51] and X86 malware study [30]). Similar to these studies, we next examine the detection timelag in the Android malware scenario.

Question 2: Is it effective to use the Android anti-virus blacklist to filter Android malware, before they are submitted to Android markets? Our Empirical Answer: No. The public Android anti-virus blacklist is too slow at stopping malware authors from submitting Android malware to the market.

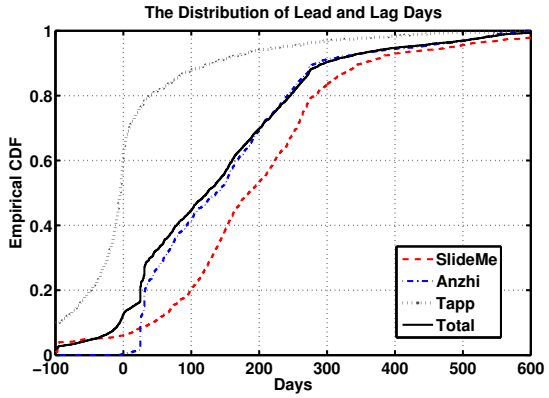
To answer this question, we use one of the most well-known (representative) Android anti-virus blacklists (VirusTotal) as the case study. VirusTotal automatically retrieves the scanning reports (alerts) for Android apps from over commercial 40 anti-virus products. In the report, it will show the malware categories of the malware sample labeled by each anti-virus product, as well as the dates when the sample is seen for the first time and the last time. Accordingly, using historical data from VirusTotal, we can measure the blacklist lag period, which is the time period delay between one malicious app's submission to the market (submission date) and its first appearance on VirusTotal (the firstly-seen date). (Note that when one malware sample is firstly seen by VirusTotal, it may not be successfully identified as malicious by those anti-virus products. Thus, our measured blacklist lag period is only the lower bound of the actual detection lag period)

For cases where malicious apps seen by the VirusTotal prior to appearing on the markets, we say that the blacklist of VirusTotal leads the markets. Conversely, VirusTotal lags the markets if malicious apps are submitted to the markets before

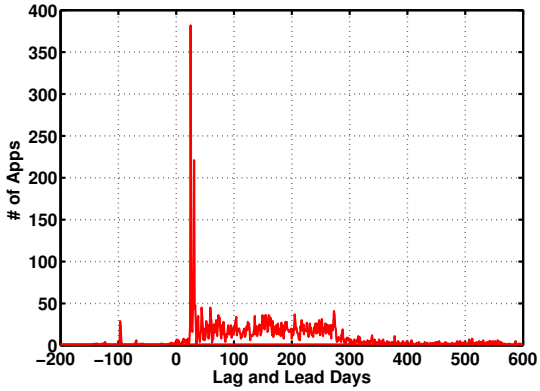
becoming examined by VirusTotal. Lead and lag times can indicate the efficiency of using the VirusTotal blacklist to stop malware authors submitting their malware samples to the markets.

We begin measuring blacklist delay by gathering the timestamps for each Android malicious app that is submitted to the market and that is firstly seen by VirusTotal. For each malware sample submitted by multiple accounts (or in multiple markets, or at different timestamps), we consider each submission as a unique, independent event. Due to the fact that GooglePlay does not provide the exact time when the app is firstly uploaded, we test this experiment based on the malware crawled from those three third-party markets.

Figure 6.3(a) shows the lead and lag periods for Android malware, where we see that around 90% of collected malware samples appear on markets prior to being seen by VirusTotal. More specifically, over 99% malware samples from Anzhi and 93% from SlideMe appear on markets prior to being being seen by VirusTotal. Although the Russian market (Tapp) is founded much later and much less active than Anzhi and SlideMe, around 40% of malware samples from it can not be identified by the blacklist before their appear on the market. In addition, the average lag period is over 167 days, which is much longer than that (from 2 to 27 days) for detecting desktop malware by using existing anti-virus products as stated in one recent study[30]. A more extensive presentation of blacklist lag days can be seen in Figure 6.3(b), showing the volume of malware samples per lead and lag day. Through this observation, we find that the Android anti-virus blacklist in fact lags behind Android malware's appearance on the markets.



(a) Distribution of Lag Period



(b) Volume per Lead and Lag Day

Figure 6.3: Lag period between the submission date and the firstly-seen date.

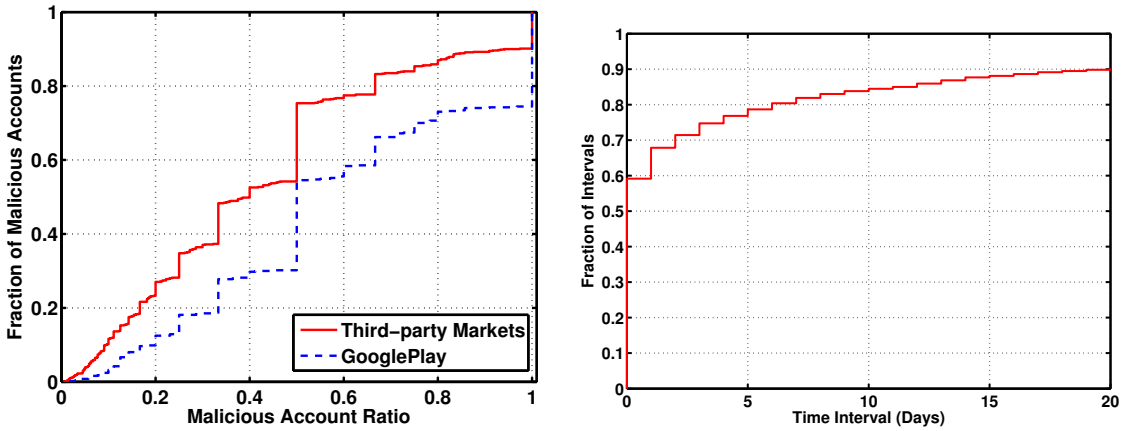
Rank	**GooglePlay**		**SlideMe**	
1	Personalization	357 (22.41%)	Game	788 (40.51%)
2	Entertainment	249 (15.63%)	Wallpapers	271 (13.93%)
3	Arcade and Action	97 (6.09%)	Entertainment	265 (13.62%)
Total	703 (44.13%)		1,324 (68.07%)	
Rank	**Anzhi**		**Tapp**	
1	Entertainment	2830 (58.47%)	Game	713 (49.17%)
2	Lifestyle	508 (10.50%)	Wallpapers	362 (24.97%)
3	Wallpapers	398 (8.23%)	Entertainment	149 (10.28%)
Total	3,736 (77.19%)		1,224 (84.41%)	

Table 6.5: The categories of apps tend to be malicious.

6.3.2.3 Common Behaviors Among Malicious Accounts

Question 3: Are there any common behavioral characteristics among malicious market accounts? Our Empirical Answer: Yes. There is a spatial and temporal locality property in terms of malware authors' submissions of their malware samples. Malicious authors tend to repeatedly use the same market accounts to post multiple malicious apps, and within a short time period.

To answer this question, for each of 3,517 malicious market accounts, we first



(a) Account Malicious Ratio (b) Time Interval

Figure 6.4: The distribution of account malicious ratios, and the time intevals between two consequent malware submissions from the same malicious account.

calculate its account malicious ratio, which is the percentage of malware samples in all of its submitted apps. As seen in Figure 6.4(a), around 50% of malicious accounts from third-party markets and around 70% of malicious accounts from GooglePlay have a malicious ratio higher than 0.5 (i.e., at least one malicious app in two submissions). This observation indicates that malware authors tend to repeatedly use the same malicious accounts to submit malicious apps. Also, we can find that malicious accounts in GooglePlay typically have a higher ration than those malicious accounts in the third-party markets. That might because GooglePlay requires higher cost for registering market accounts (e.g., valid Google accounts and registration fee). Thus, it will bring a much higher cost for malware authors to create a large number of GooglePlay accounts to spread malware.

We also examine the time interval (in days) between two consequent submissions of malicious apps for the same malicious accounts. As seen in Figure 6.4(b), over 60% of time intervals are zero (i.e., those two consequent submissions happened in the same day), and around 80% of time intervals are less than 5 days. This observation

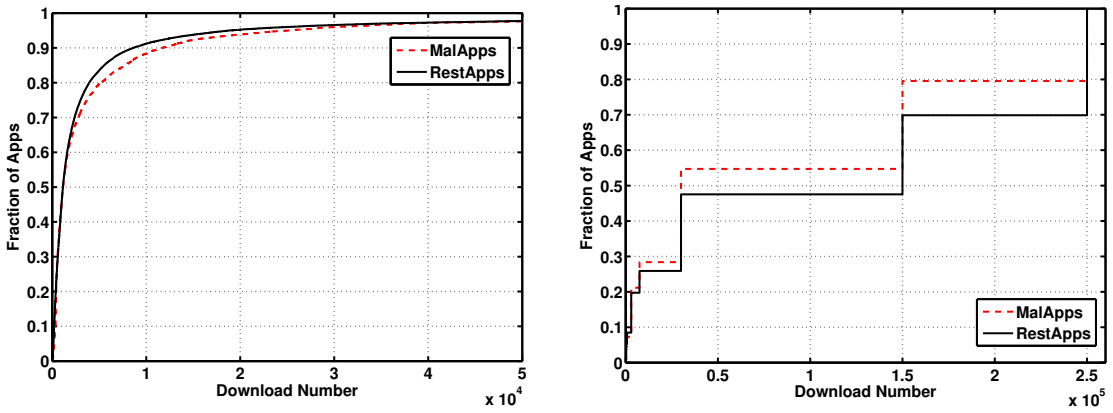
indicates that malware authors tend to submit multiple malware samples within a short time period, an interesting spatial and temporal locality property. Utilizing this property, we could find more malicious apps by checking those apps whose author names are the same with known malicious ones', especially when they are submitted within a short time period.

6.3.2.4 App Quality VS App Popularity

Without a deep knowledge about Android app security, many users prefer to trust those popular apps with high downloading numbers. *Question 4: Are those apps with higher downloading numbers more safer?* Empirical Answer: No. An app's downloading number has not necessarily a strong correlation with its quality. Many malicious apps have also been downloaded for a great number of times. To answer this question, we need to calculate our crawled apps' downloading numbers. In particular, unlike third-party markets, which provide the accurate value of each app's downloading number, GooglePlay only provides the interval of the downloading number (e.g., 5,000-10,000). Also, once the downloading number is higher than 250,000, GooglePlay only shows the interval as "$> 250,000$". Thus, we use an approximate way to count the downloading number for the apps in GooglePlay. More specifically, if the number is smaller than 250,000, we use the median value of the interval; otherwise, we directly use 250,000. Then, we compare the downloading numbers between the dataset of `MalApps` and `RestApps` for third-party markets and GooglePlay, respectively.

As seen in Figure 6.5, in both third-party markets and GooglePlay, we can find that the distributions of the downloading number between `MalApps` and `RestApps` are similar. Specifically, in the third-party markets, we can find that the percentage of the apps in `MalApps` that have been downloaded more than 10,000 times is

around 12%, which is even slightly larger than that (10%) in `RestApps`. Also, we can find that around 20% of apps in `MalApps` have been downloaded more than 5,000 times, whereas around 80% of apps in `RestApps` have been downloaded less than 5,000 times. Similarly, in GooglePlay, around 45% of apps in `MalApps` have been downloaded more than 150,000 times, whereas around 50% of apps in `RestApps` have been downloaded less than 150,000 times. Thus, this observation indicates that many popular apps with high downloading numbers are still malicious, i.e., an app's popularity is not an effective indication to its quality.



(a) Third-party Markets (b) GooglePlay

Figure 6.5: The comparison of the downloading numbers for `MalApps` and `RestApps` in the third-party markets and GooglePlay.

6.3.2.5 Common App Categories Among Malicious Apps

Question 5: Are there any specific app categories that Android malware authors tend to masquerade their malware samples to belong to? Our Empirical Answer: Yes. Malware authors tend to register their malware samples into a few specific categories (e.g., Game, Entertainment, and Wallpapers).

To answer this question, in each market, we rank its app categories according to the number of malware samples belonged to them. Table 6.5 lists the top three categories for each market.

From this table, we can find that the majority of the malicious apps belong to a few categories (e.g., Game, Entertainment, Wallpapers.). More specifically, the categories of Entertainment and Game ranks in the top three in all these three markets. (Note that in GooglePlay, the apps belonged to"Arcade and Action" are mainly action games, which are typically categorized as "Game" in the third-party markets; in the market of Anzhi, game apps are registered with the category of Entertainment instead of "Game".) In addition, in all three third-party markets, top three categories are used by more than over 60% of malware samples. Even in GooglePlay, which categorizes apps in a more fine-grained way (i.e., apps are registered into multiple more fine-grained categories), the top three categories covers more than 40% of malware samples.

This observation indicates that some categories are highly used by malware authors. This phenomenon might because malicious apps may obtain more users' attention, when they are registered to belong to those hot categories such as "Entertainment" and "Game". The apps belong to those hot categories typically have a large downloading number. In addition, since the programming logic of those benign Wallpaper apps is typically simple, compared with developing completely new malicious apps, it is easier for malware authors to develop malicious apps by inserting malicious payloads into those benign Wallpaper apps.

6.4 Analyzing Network-level Behaviors

6.4.1 Extracting Remote Servers

To facilitate the analysis of the network-level behaviors, we need to extract remote servers (domain names and IP addresses) communicated by Android malware, i.e., we need to extract the URLs, domain names and IP addresses that are visited by Android malware.

A naive approach is to extract those URLs that are hard-coded in the apps by using existing Android app static analysis tools (e.g., [45]). However, this approach is not effective, due to the fact that many malicious Android apps use diverse techniques to hide visited URLs instead of hard-coding them as constant strings to evade detection. Such techniques range from string encoding, string encryption, string obfuscation, splitting string into segments, to saving URL string segments into XML files used in the apps. Thus, this strategy can not be used to effectively extract remote servers.

To avoid such a limitation, we extract remote servers by running apps in a customized Android runtime environment (Android phone emulator). Before analyzing each app, the emulator will start from a clean snapshot that is saved at the starting point to avoid possible effects (e.g., the changes of SD card) generated by other apps. To trigger an app to execute more networking connections, we also design an Android app UI fuzzing tool to simulate real users' usage of the app by adding random UI events (e.g., click buttons, stretch the views, and type characters).

While running each app, our environment will record its networking attempts by using `Android TCPDump`[46]. Unlike traditional TCPDump, Android TCPDump will only save the networking packets made by the emulator instead of the host. In addition, we also record networking attempts saved in the emulator's runtime log file

by using `Android Logcat`[46]. Android Logcat will both record some sensitive networking attempts (e.g., the links of the advertisements for other Android apps), and those failed networking attempts due to some networking exceptions (e.g., Once the remote server is no longer alive, Logcat will record those failed networking attempts to the remote server with the exception of "java.net.UnknownHostException".) that are not captured in the Android TCPDump. Finally, a parser is developed to automatically extract those networking attempts. To facilitate the following analysis, for each URL, the parser will also extract its domain name; for each IP address, the parser will reverse lookup its domain name, if available.

As seen in Table[3] 6.6, we finally collect 239,582 unique URLs leading to 25,099 unique servers[4] (including 19,342 unique domain names and 5,755 unique IP addresses). More specifically, by running 9,712 malicious apps in the dataset of `MalApps`, we collect a dataset of 34,176 unique URLs and 5,142 remote servers (including 3,980 unique domain names and 1,162 IP unique addresses), named as `MalServers`.

Type	URLs	Domains	IPs	Servers
MalServers	34,176	3,980	1,162	5,142
AdwareServer	35,267	3,112	1,057	4,169
RestServers	176,949	16,580	5,125	21,707
Total	239,582	19,342	5,755	25,099

Table 6.6: The summary of extracting remote servers.

[3]The datasets of `MalServers`, `AdwareServers` and `RestServers` in the table represent the servers extracted from the apps in the dataset of `MalApps`, `AdwareApps` and `RestApps`, respectively.
[4]Each remote server is counted by one of its valid domain names, if aviable. Otherwise, the server is counted by its IP address.

6.4.2 Filtering Benign Servers

Since our goal is to analyze the network-level behaviors of Android malware authors by analyzing the characteristics of the remote servers uniquely used by malware, a data filtering step is required to filter benign servers that are also visited by Android malware. In another word, we need to filter benign servers saved in the dataset of `MalServers` in Table 6.6). We first filter top 10,000 Alexa [2] domain names. Then, we use two conservative strategies to further filter benign servers: (1) filtering all the servers visited by the apps in the `RestApps`, and (2) filtering top frequently used servers by the apps in the `RestApps`.

More specifically, in the former strategy, we generated a filtered dataset named as `FAMalServers` by filtering all the servers in `RestServers` from `MalServers`. In this way, `FAMalServers` contains 2,288 unique servers. Note that due to the possibility that a small portion of apps in `RestApps` might still be malicious (i.e., the false negatives of those AV products), some servers in `RestServers` might still be uniquely visited by Android malware. Accordingly, this strategy may filter more remote servers that are uniquely visited by Android malware. However, this strict strategy could guarantee nearly all of the servers left in `FAMalServers` are unique for malware, especially when the dataset of `RestServers` is sufficiently large.

As a supplement, in the latter strategy, instead of filtering all servers in `RestServers`, we only filter those servers that are highly visited by the apps in `RestApps`. More specifically, we first rank the servers in `RestServers` according to the number of unique apps in `RestApps` that visit them. Then, we empirically filter the top $N = 1,000$ servers[5] from `MalServers` and generate another filtered dataset named

[5]According to our empirical observation, $N = 1,000$ is a proper value for filtering known benign servers while keeping unknown servers. However, this value could still be tuned with the tradeoff between filtering more truly benign servers and keeping more servers uniquely used by malware.

FTMalServers, which contains 4,379 servers. To facilitate the later comparison, we name the dataset of the rest of 20,707 servers as FTRestServers by filtering those top 1,000 servers from RestServers, as seen in Table 6.7.

Dataset	FAMalServers	FTMalServers	FTRestServers
# of Servers (Unique)	2,288	4,379	20,707

Table 6.7: The number of servers in each filtered dataset.

Due to the lack of the perfect ground truth, we clearly acknowledge that even these two filtered datasets (FAMalServers and FTMalServers) may still contain some benign servers, and miss some malware servers. However, note that these two strategies are essentially complementary and we mainly use these two filtered datasets to compare the network-level behaviors between malicious apps and other apps. If our conclusions could hold under the usage of both two datasets, we believe that the same conclusions will also likely hold under the usage of another real dataset, even though the actual values of specific metrics may vary a little bit.

6.4.3 Detailed Analysis

Similar to the analysis of the market-level behaviors, we next perform our detailed analysis of the network-level analysis in a question-and-answer fashion.

To determine which IP address and network spaces are mainly used by malware authors to create their remote servers, our first network-level analysis focuses on examining the network addresses visited by Android malware.

6.4.3.1 Usage of Network Space

Question 1: Is the distribution of remote servers' IP address space visited by Android malware different from those visited by other apps? And are there any special networks that are mainly utilized by Android malware authors? Our Empirical Answer: Yes. We find that a few concentrated portion of IP address spaces are highly visited by malware samples rather than other apps.

We compare the ASes visited by malicious apps and other apps. More specifically, for each remote server, we extract the AS it belongs to. Then, in each dataset, we rank the ASes according to the number of unique apps that visit them. We extract the IP addresses of the remote servers in each of the four datasets (`RestServers`, `MalServers`, `FTMalServers` and `FAMalServers`). In each of the four datasets, for each individual IP network prefix (i.e., each /8 subnet from `0.0.0.0/8` to `255.0.0.0/8`), we calculate the number of the unique apps (N_{subnet}) that visit the subnet. Then, for each subnet, we calculate the ratio of N_{subnet} to the total number of the malware samples in the dataset, termed as Subnet Malicious Ratio. Accordingly, a higher ratio for a subnet implies that the subnet is visited by more malicious apps.

As seen in Figure 6.6, although most IP address ranges that are used by a significant amount of malware also used by a lot of other apps, a few subnets are significantly used by more malware than other apps. For example, in terms of the subnets of `23.0.0.0/8` and `54.0.0.0/8`, the ratios of all three malware datasets (`MalServers`, `FTMalServers` and `FAMalServers`) are much higher than that of `RestAppServers`. This characteristic implies that malware authors may tend to utilize a few special subnets to host remote servers to communicate with their malware. To further analyze the usage of ASes and domains for malware, we also compare the top ten ASes

for RestServers, MalServers, and FAMalServers as seen in Table 6.8, and show the top ten domain names for FTMalServers and FAMalServers as seen in Table 6.9.

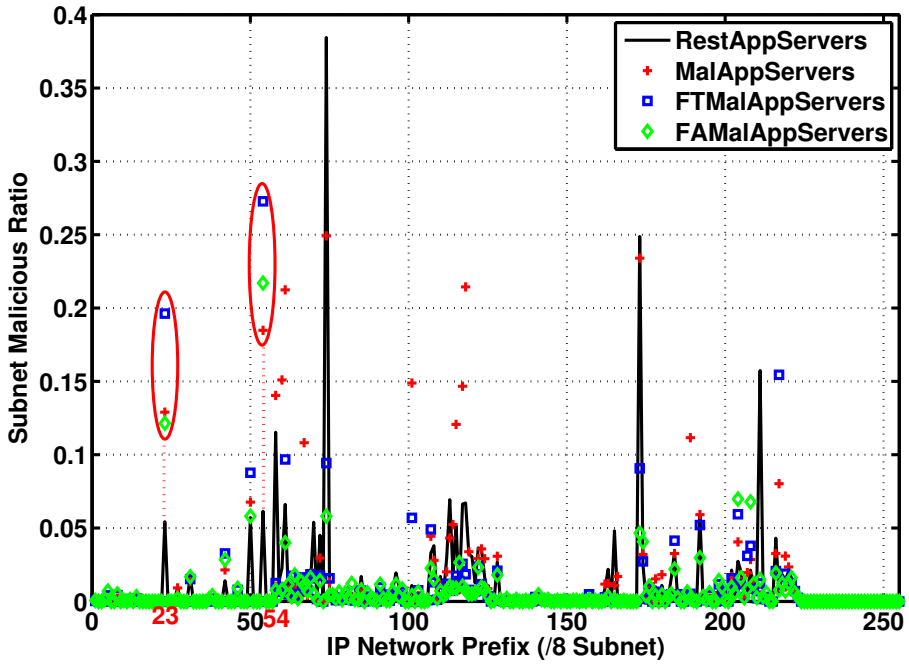


Figure 6.6: The comparision of the distribution of the usage of IP address between malicious apps and other apps.

Table 6.8 shows the top ten most frequently visited ASes in the datasets of RestServers and MalServers. From this table, we can find that in RestServers, only one of the top ten ASes belongs to the cloud vendor and it ranks the sixth. However, in MalServers, four of the top ten ASes belong to cloud vendors.

Next, we show the top ten most used domain names by malware samples in the datasets of FTMalServers and FAMalServers. As seen in Table 6.9 and 6.10, we can find that in the dataset of FTMalServers, seven of the top ten most used domain

	RestServers		MalServers	
Rank	**AS Number**	**AS Name**	**AS Number**	**AS Name**
1	AS15169	Google	AS9308	Abitcool
2	AS4134	Chinanet	AS4134	Chinanet
3	AS9308	Abitcool	AS15169	Google
4	AS4808	China169 Beijing	AS17964	Beijing Dian-Xin-Tong (**Cloud**)
5	AS24400	Shanghai Mobile	AS23724	IDC, China (**Cloud**)
6	AS14618	Amazon (**Cloud**)	AS14618	Amazon (**Cloud**)
7	AS22577	Google	AS24400	Shanghai Mobile
8	AS4837	China169 Backbone	AS17431	Beijing TONEK
9	AS17431	Beijing TONEK	AS3549	Global Crossing
10	AS20645	PurePeak Limited	AS33494	IHNetworks(**Cloud**)

Table 6.8: The top ten ASes for RestApps.

names belong to the cloud vendors; one domain name is mainly used as C&C Servers to receive victims' private information; the other two domain names are used for the abusive mobile advertisement. Also, in the dataset of `FTMalServers`, all of the top ten domain names belong to the cloud vendors.

	FTMalServers	
Rank	**Domain Names**	**Usage**
1	ad.leadboltapps.net	Abusive Advertisement
2	d36hc9ptsltjmz.cloudfront.net	Cloud
3	m.airpush.com	Abusive Advertisement
4	ec2-54-225-174-248.compute-1.amazonaws.com	Cloud
5	client.139vps.com	C&C Server
6	ec2-23-21-95-12.compute-1.amazonaws.com	Cloud
7	ec2-54-225-131-82.compute-1.amazonaws.com	Cloud
8	ec2-54-235-138-219.compute-1.amazonaws.com	Cloud
9	ec2-204-236-218-179.compute-1.amazonaws.com	Cloud
10	ec2-54-243-171-43.compute-1.amazonaws.com	Cloud

Table 6.9: The top ten most frequently used in `FTMalServers`.

	FAMalServers	
Rank	**Domain Names**	**Usage**
1	ec2-54-225-174-248.compute-1.amazonaws.com	Cloud
2	aec2-204-236-218-179.compute-1.amazonaws.com	Cloud
3	ec2-54-243-171-43.compute-1.amazonaws.com	Cloud
4	ec2-23-21-67-42.compute-1.amazonaws.com	Cloud
5	ec2-174-129-232-156.compute-1.amazonaws.com	Cloud
6	208.43.117.142-static.reverse.softlayer.com	Cloud
7	ec2-23-21-51-117.compute-1.amazonaws.com	Cloud
8	ec2-23-21-251-51.compute-1.amazonaws.com	Cloud
9	ec2-54-243-36-7.compute-1.amazonaws.com	Cloud
10	ec2-50-112-100-234.us-west-2.compute.amazonaws.com	Cloud

Table 6.10: The top ten most frequently used in `FAMalServers`.

Also, in the dataset of `FAMalServers`, the IP addresses of three cloud hosts (the first, third and ninth) belong to the subnet of 54.0.0.0/8; the IP addresses of three cloud hosts (the fourth, seventh and eighth) belong to the subnet of 23.0.0.0/8. This observation also explains why these two subnets are highly used by Android malware as discussed in the previous question.

From the above analysis, we can find that Android malware authors tend to host their remote servers in the cloud vendors. Motivated by this observation, we further analyze the common characteristics among malware samples, which communicate with those servers hosted in the cloud vendors.

6.4.3.2 Usage of Cloud Vendors

Question 2: Are there any common charaterisitcs of Android malware samples, which communicate with the servers hotsed in the cloud vendors? Empirical Answer: Yes. Malicious Android apps, which communicate with the same cloud server/subnet, are very likely to belong to the same malware families.

To answer this question, we use the popular cloud vendor (AmazonEC2) as the

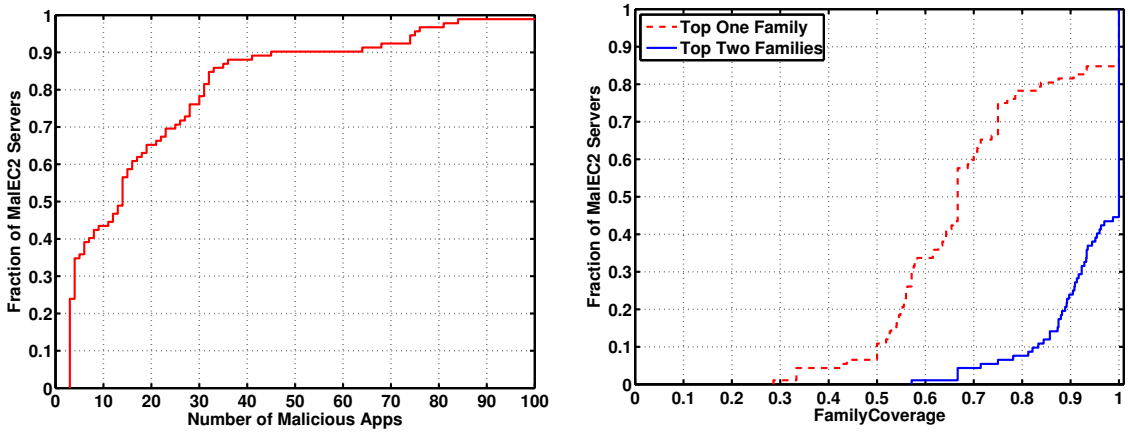
case study. We first extract AmazonEC2 servers[6], termed as `MalEC2Servers`, visited by malicious apps. Then, for each `MalEC2Server`, we extract its `MalEC2Families`, which are the malware families of the malware samples that visit MalEC2Server[7].

We first analyze the variation of the malware families in the MalEC2Servers. More specifically, we first extract 92 different MalEC2Servers that have more than two MalEC2Families. Figure 6.7(a) shows the distribution of the number of unique apps in those 92 MalEC2Servers. From this figure, we can find that around 58% of those MalEC2Servers are visited by more than 10 different malicious apps (i.e., these MalEC2Serverstend to be visited by multiple malicious apps.).

Then, for each MalEC2Familiy in each MalEC2Server, we calculate its value of FamilyCoverage, which is the percentage (coverage) of the malware samples belonged to the MalEC2Family in all malware samples that visit that MalEC2Server. Next, in each MalEC2Server, we rank its MalEC2Families according to their values of FamilyCoverage. Figure 6.7(b) shows the distribution of the FamilyCoverage of the top MalEC2Family, and the sum of the top two MalEC2Families' values of FamilyCoverage, among those 92 MalEC2Servers. From this figure, we can find that in over 90% of MalEC2Servers, the top MalEC2Family's FamilyCoverage is higher than 0.5 (i.e., in over 90% of MalEC2Servers, more than half malware samples belong to the same family.). While considering the sum of the top two families, the coverage is increased to 0.85 (i.e., in over 90% of MalEC2Servers, the top two families coverages over 85% of malware samples.) Also, we can see that in over 50% of MalEC2Servers, the sum of the top two families is 1.0 (i.e., in over half of MalEC2Servers, there are only two different families.). These observations imply that the malware samples that visit the same AmazonEC2 server tend to belong to the same family.

[6]These servers are identified by their domain names, which start with "ec2" and end with "amazonaws.com"

[7]This is achieved by analzying each malware sample's scanning report received from VirusTotal.



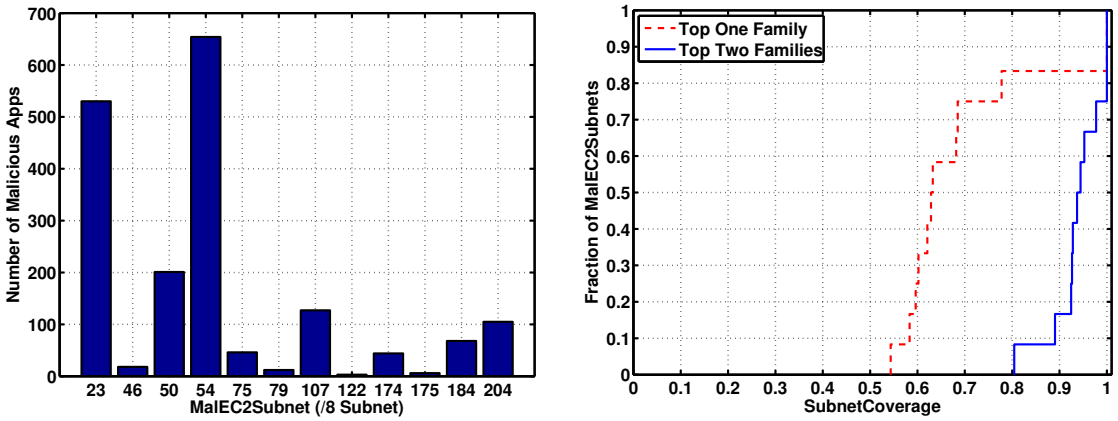
(a) Number of Malicious Apps (b) FamilyCoverage

Figure 6.7: The distributions of the number of malicious apps, and the famliy coverages among MalEC2Servers.

Next, we group MalEC2Servers into 12 different /8 subnets according to their IP addresses. We name these subnets as MalEC2Subnets, and further examine the variation of malware families in these subnets. Figure 6.8(a) shows the number of unique malware samples in each MalEC2Subnet. Similar to the above experiment, for each MalEC2Family in each MalEC2Subnet, we calculate its value of FamilyCoverage, which is the percentage (coverage) of the malware samples belonged to the MalEC2Family in all malware samples that visit the severs belonged to that MalEC2Subnet.

Figure 6.8(b) shows the FamilyCoverage of the top family and the sum of the top two values of FamilyCoverage for all MalEC2Subnets. From this figure, we can find that the FamilyCoverage of the top family in all MalEC2Subnets are higher than 0.5 (i.e., for each MalEC2Subnet, over 50% malware samples that visit that MalEC2Subnet belong to the same family.) Also, this value increases to 0.8, while considering the top two families (i.e., for each MalEC2Subnet, over 80% malware samples that visit that MalEC2Subnet belong to two families.) This observation

implies that malware samples that visit the same MalEC2Subnet are very likely to belong to the same malware family.



(a) Number of Malicious Apps (b) FamilyCoverage

Figure 6.8: The distributions of the number of malicious apps, and the famliy coverages among MalEC2Subnets.

From the above experiments, we can find that malware samples that visit the same cloud server/subnet are very likely to belong to the same malware family. This server vendor locality property (i.e., malware authors tend to rent servers from the same cloud vendor) is interesting. This shows that with the popularity of cloud hosting services, malware authors begin to use cloud machines as remote servers. That is mainly because comparing with deploying personal servers, it will cost less time/money to use a cloud vendor, and the anonymity is also better preserved. From defense point of view, once we find specific cloud servers that are used by malware authors to communicate with their malware samples, we may likely to find more such servers in the same cloud hosting network that belong to the same malware authors.

6.4.3.3 Effectiveness of IP/Domain Blacklists

Since malware samples tend to use specific IP addresses and networks, we raise the following question. *Question: Is that effective to use existing IP (or domain name) blacklists to find Android malware?* Empirical Answer: Not very effective. Even after the malware samples in our dataset are submitted to the market for a long time (more than one year), existing IP (or domain name) blacklists can still only find a small portion of them.

In this experiment, we extract the servers in `MalServers` that are labeled as malicious by using the following four popular domain name and IP blacklists: VirusTotal [114] (`VT`), WhatIsMyIPAddress [122] (`IPB`), Malware Domain List [73] (`MDL`) and Malware Domain Blocklist [33] (`MDB`). More specifically, VirusTotal can also be used to check malicious IP addresses and domain names. WhatIsMyIPAddress is a powerful IP blacklist, which integrates results from 78 different blacklist servers. Both Malware Domain List and Malware Domain Blocklist are blacklists for identifying malicious domains. Table 6.11 shows the number of identified unique remote servers and the number of affected apps (malicious apps) by using each blacklist.

BlackList	**VT**	**IPB**	**MDL**	**MDB**
Remote Servers	213	510	5	4
Total	659			
Affected Apps	735	2,718	22	7
Total	3,210			

Table 6.11: The number of identified remote servers and affected malicious apps.

From this table, we can find that both MDL and MDB can only find a very small number (less than 25) of malicious apps from the dataset of 9,712 malicious apps.

Also, even though our malicious apps are collected with the usage of VirusTotal, the IP/domain name blacklist from VirusTotal can only be used to find 735 (around 7%) malicious apps. Even combining those four popular blacklists, we can identify only 3,210 (around 33%) malicious apps, not even considering the long time lag of these IP/domain blacklists. This observation implies that although existing IP/domain blacklists can be used to facilitate finding malicious apps, this strategy is neither efficient nor effective to catch a high coverage of the malware.

6.4.3.4 Malware Community

Due to the observation that malware samples frequently share the same authors and remote servers, we next analyze whether the submissions of the malicious apps are more likely to be organized activities or isolated actions. *Question 4: Are there any Android malware communities?* Our Empirical Answer: Yes. A few large malware communities contribute to a great amount of malicious apps.

In this experiment, we cluster malicious apps into communities according to their community relationships. More specifically, we consider there is a *community relationship* between two malicious apps, if they share the same author name or at least one malicious server. The intuition behind this is that if two apps share the same author name (market account), they essentially belong to the same author, thus belong to same organization (community). If two apps share the same remote servers, they are also very likely to belong to the same organization, who use those remote servers to achieve their malicious goals.

To model such community relationships among malicious apps, we build a community relationship graph $G = (V, E)$. In this graph, each node (v_i) is represented as a two-tuple (app, author name), in which the author name is the concatenation of the account name and the market name. There is an edge e_{ij} between node v_i

and v_j, if these two nodes share the same app (i.e., the same value of MD5) or the same author name, or at least one malicious server in the dataset of `FTMalServers`. Accordingly, our relationship graph contains 9,850 nodes and 621,166 edges, as visualized in Figure 6.9.

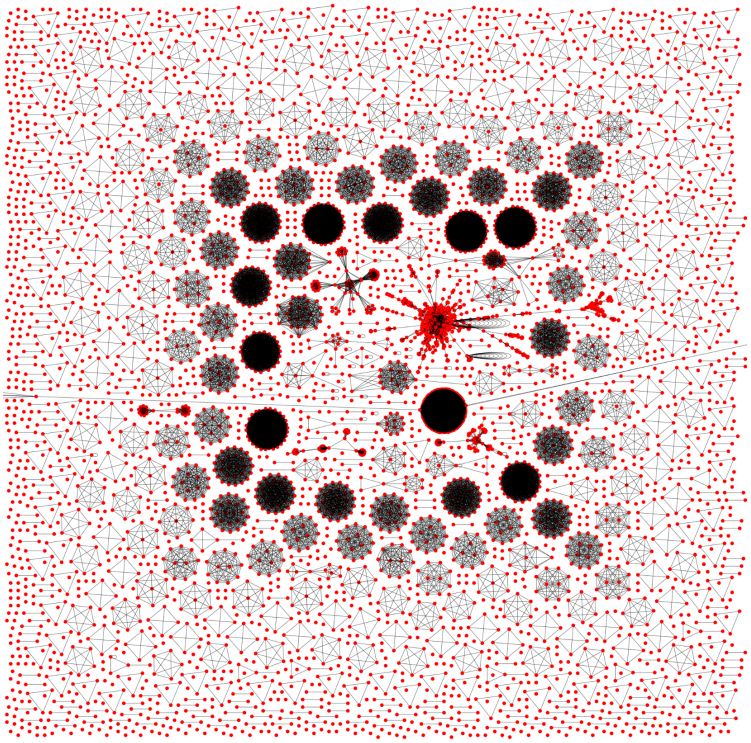


Figure 6.9: The visulazation of the community graph for malicious Android apps.

From this figure, we can clearly find that the majority (80.67%) of the nodes are connected with other nodes. Also, there are a few large subgraphs that are well connected. This observation implies that there are some large malware communities.

We next examine the percentage of the malware samples covered by those large communities in all malware samples. More specifically, we consider each connected subgraph as one community, and thus obtain 847 communities. We next rank those

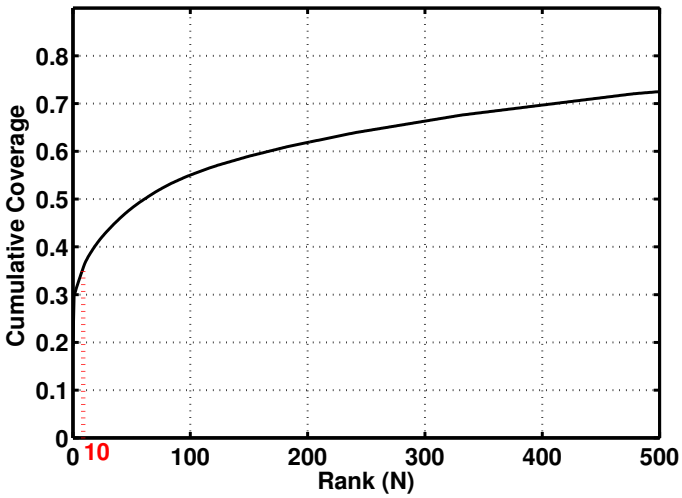


Figure 6.10: The distribution of the cumulative community coverage under different ranks.

communities based on their size. For each community, we calculate its coverage (c_i), which is the percentage of its malware samples in all malware samples. Then, summing up the coverages from the top community to the nth community, we calculate the cumulative coverage as $C_n = \sum_{i=1}^{n} c_i$. Figure 6.10 shows the distribution of the cumulative coverage with the value of n.

From this figure, we can find that the top 10 communities covers over 35% of all malware samples, and the top 100 communities covers covers over 55% of all malware samples. This observation implies that a few communities contribute to a large number of malware samples.

We next make an in-depth analysis of the top three communities. More specifically, we term one app's duration as the time period (in days) between the app's submission time and the date when we crawled it; we term one apps' infection rate as the ratio of its downloading numbers to its duration. Then, we measure those five communities' size, average downloading number, average duration, and average

infection rate (see Table 6.12).

Rank	Size	Downloading Number	Duration	Infection Rate
1	1,895	7,551	117	180
2	1,031	453	152	3.75
3	85	569	99	5.77

Table 6.12: In-depth analysis of the top three communities.

From this table, we can see that the malware samples in all these three communities have been downloaded many times, and last for a long time. More specifically, the average downloading number for the top community is even higher than 7,500; the average duration of all the communities are longer than three months. On each day, the malware samples in all these three communities are downloaded more than 3 times, especially this number increases to 180 in terms of the top community.

We further analyze the inner community properties of these three communities. In the top community, the most frequently used server is "217.65.36.4", shared by 527 malicious apps. Once directly visiting this server by using its IP address, it shows a "404" error. However, this server is one C&C server, located in Israel, used for communicating malware samples belonged to the family of `Plankton.P`. Once victims install this type of malware, the malware will download actual malicious payloads from the C&C server instead of directly executing malicious behaviors. Thus, this type of malware is more stealthy and difficult to detect than other types of malware. That might be one of the reasons why this community of malware could allure a great number of downloads. In this community, we also find one malicious GooglePlay account, named "Antonio Tonev", shared by 30 malicious apps. In fact, this developer has been reported as an notorious malware author to submit multiple

malicious apps in GooglePlay [82].

In both the second and third communities, those apps are connected due to the abusive usage of the Android market by one malicious market account. In the second community, one malicious account named "hongxiutianxiang" submitted 1,024 malicious apps to the market of Anzhi. This malicious account mainly inserted malicious payloads into benign WallPaper apps and E-book apps. In the third community, one malicious account named "phoneliving" submitted 85 malicious apps to the market of SlideMe. These malicious apps are developed mainly by inserting malicious payloads (including malware downloaders and abusive advertisements) into game apps.

From the above observation, we can find that some malicious apps have strong community relationships. This implies that if we can find a few malicious apps in particular malware communities, we could find more malicious apps belonged to the same communities.

6.5 Combating Malicious Apps

We next discuss a defensive technique that can be used to efficiently catch more malicious apps, as well as to further verify the correctness of the defense insights obtained from the previous analysis.

Considering that there are a large number of submissions to Android markets, not all Android markets have sufficient time/resource/capability to make a deep security analysis of their apps. In fact, due to those practical restrictions, most of current third-party Android markets do not apply any vetting process to examine the quality of their apps. Thus, a lightweight inference algorithm, to guide quickly identifying more suspicious apps instead of analyzing all apps given limited resources or time, is indeed needed, especially for those third-party markets. In this section, we propose a lightweight algorithm (AMIA) to infer malicious apps based on our analysis of

the market-level and network-level behaviors of Android malware ecosystem. Note that our inference algorithm is positioned as a *complementary*, lightweight strategy to quickly find those more suspicious apps. The limitation of our approach will be discussed in Section 6.5.3. In practice, we envision AMIA could be combined with existing detection approaches (e.g., the permission and API used by the apps) for more complete protection of the Android market.

6.5.1 Design of Inference Algorithm

In brief, our inference algorithm (AMIA) propagates malicious scores from a seed set of known malicious apps to other apps according to the closeness of their community relationships. If an app accumulates a sufficient malicious score, it is more likely to be a malicious app.

The intuition is based on the two observations found in our previous analysis of the market-level and network-level behaviors: (1) Malware authors tend to use the same market accounts to spread multiple malware samples, and within a short time period. Thus, an app sharing the same author names (market accounts) with known malicious apps are more suspicious, especially when their submission times are close; (2) A few Android malware communities contribute a large portion of Android malware. Thus, by propagating malicious scores from the seed malware samples, we can find out more unknown malware samples with close community relationships with those seeds.

With the above intuitions, we then describe the design of AMIA in details. To infer malicious apps from a set of U unknown apps[8] by starting from a known seed set of M malicious apps, similar to the way of obtaining malware communities, we build a Malicious Relevance Graph by using these $(M + U)$ apps, denoted as $G = (V, E)$.

[8]In our preliminary experiment, we use all collected malicious apps from GooglePlay, SlideMe and Anzhi.

In this graph, each node (v_i) is represented as a two-tuple (app, author name). There is an edge e_{ij} between node v_i and v_j, if these two nodes has the same app (i.e., the same value of MD5) or the same author name, or their apps share at least one remote server[9].

Then, to model the closeness of their community relationships, we assign a weight $w_{ij} \in E$ for each edge $e_{ij} \in E$. As shown in Table 6.13, if two nodes share the same value of MD5 or author name, the weight of the edge will be added 1.0, respectively. If two nodes share n remote servers, then the weight will be added based on the Gaussian Error Function [126] $f(n) = \frac{1}{2}(1 + erf(\frac{n-\mu}{\gamma}))$, which can normalize the weight into the interval of $[0, 1]$. If the time interval between two nodes' submission times is δ days, then a value of $g(\delta) = 1.0/(1.0 + \delta)$ will be added to the weight[10].

Feature	**MD5**	**Author**	**Remote Servers**	**Submission Time**
Weight	1.0	1.0	$f(n)$	$g(\delta)$

Table 6.13: Weights used to build the malicious relevance graph.

Then, for each node whose app is malicious, we assign a non-zero malicious score and propagate this score to other nodes according to the weights of the edges between them by using the PageRank algorithm [128]. When the score vector converges after several propagation steps, we infer the apps in those nodes with high malicious scores as malicious apps.

[9]In our preliminary evaluation, we filter out the top 1,000 frequently visited servers by those $(M+U)$ apps.

[10]The weight functions used in our empirical exmpriment could be further tuned to achieve better performance based on different types of datasets.

6.5.2 Evaluation

Since the goal of our propagation-based algorithm is not to cover all malware samples in the market, an evaluation metric like false negative rate is probably not appropriate in our scenario. Instead, to evaluate the effectiveness of our inference algorithm, we consider both the number of correctly inferred malicious apps, termed as "Hit Number", and the ratio of Hit Number to the total number of inferred apps, termed as "Hit Rate". Thus, a higher Hit Number indicates the algorithm can catch more malicious apps; and a higher Hit Rate indicates the algorithm can infer malicious apps more accurately.

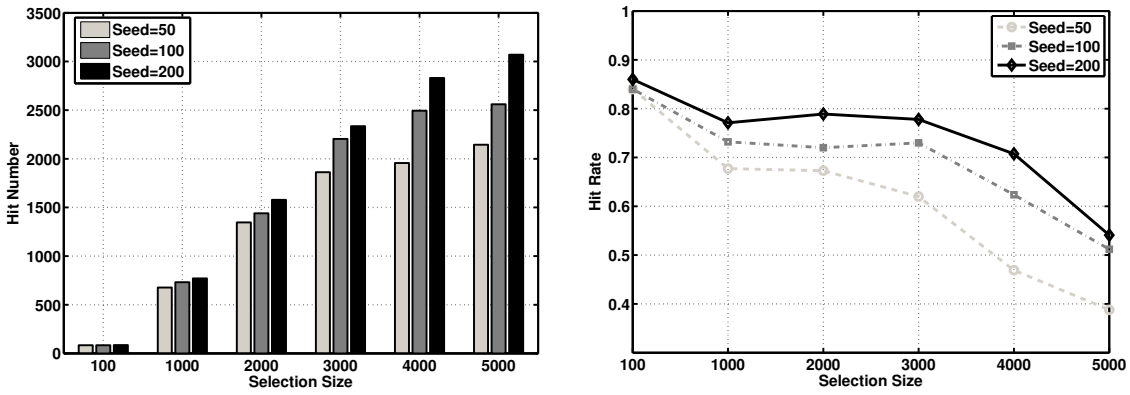
These two metrics have been similarly used in several existing inference-based algorithms (e.g., [138, 137]), where Hit Rate is more reasonable to be used to measure the accuracy than the false positive rate. Next, we provide our evaluation results by varying different selection sizes (i.e., the number of apps inferred in the top list), and different seed sizes.

6.5.2.1 Varying Selection Size

As seen in Figure 6.11(a), while increasing the selection size, more malicious apps could be identified by AMIA. More specifically, starting from 200 seeds and selecting 5,000 apps from the corpus of over 82,000 apps, our inference algorithm could correctly find out 3,070 malicious apps. This implies that our lightweight algorithm can be effectively used to infer more malicious apps. Also, as seen in Figure 6.11(b), the Hit Rate decreases with the increase of the selection size. That is mainly because the apps with higher malicious scores are more likely to be malicious.

6.5.2.2 *Varying Seed Size*

As in seen in Figure 6.11(a) and (b), the more seeds we use, the higher Hit Number and Hit Rate we can achieve by selecting the same size of apps. This is because when we use more malicious seeds, we have more knowledge about the community relationships among malicious apps. Thus, the performance of AMIA could be further improved, when we have more known seed malicious apps. Also, as shown in Figure 6.11(a), starting from 50 known malware samples, AIMA can find over 2,000 malicious apps, many of which do not share the same family(type) with those seeds. This implies although as a complementary and lightweight strategy to quickly find those more suspicious apps, AMIA is not designed to find all malware in the markets, it can still be used to find new types of malware.



(a) Hit Number (b) Hit Rate

Figure 6.11: The hit number and hit rate based on VirusTotal.

6.5.2.3 *Further Analysis*

We further manually analyze those inferred apps that are not labeled as malicious by VirusTotal blacklist. More specifically, while $K = 1,000$, we manually scan the

APK files of those apps, with the usage of multiple most recent Android Anti-Virus tools[11]. By using three different sets of seeds (size=50, 100, 200), 266, 217 and 179 inferred apps that are not identified by VirusTotal blacklist are reported as malicious by other Android anti-virus tools, respectively. This observation also implies that the Hit Number and Hit Rate showed in Figure 6.11 are only the lower bound achieved by our inference algorithm. While $K = 1,000$, by adding the numbers of malicious apps identified by VirusTotal (VT) and reported by other Android anti-virus tools, the actual Hit Number (Actual) can be seen in Table 6.14.

Seed Size	**50**		**100**		**200**	
	VT	**Actual**	**VT**	**Actual**	**VT**	**Actual**
Hit Number	677	943	732	949	771	950

Table 6.14: The actual hit number by using three different sets of seeds.

From this table, we can see that our inference algorithm is also a good complement to existing Android malware blacklist service, (i.e., it can quickly find out more malicious apps missed by the blacklist). Also, by using three different sets of seeds, AMIA can all correctly infer more than 940 malicious apps, while selecting 1,000 apps (i.e., the Hit Rate is higher than 0.94).

6.5.3 Possible Evasions

Malware authors could try to evade our inference algorithms by only submitting a very small number (e.g., only one) of malicious sample per account. However, this strategy will significantly limit the effectiveness of distributing malware. Thus, they may try to create a large number of market accounts and use each account to submit

[11]This is very time-consuming and tedious work. Thus, it is not practical to use this strategy to identify malware from a large-scale (e.g., over 70,000) corpus of apps in our data collection phase.

one malicious app. However, in many Android markets, it will bring a significant cost to register a great amount of market accounts. For example, in GooglePlay, registering each market account requires one valid Google account and paying 25 dollars registration fee. Also, as long as the malware authors use the same groups of remote servers to communicate with their malware samples, they may still be caught due to their shared remote servers.

Malware authors may try to increase the false positives of our inference algorithms by submitting benign apps on the same day when they submit their malware samples. However, we can use a more strict strategy to filter those benign apps, due to the fact that those benign apps do not communicate with those remote servers used by malware.

6.6 Limitation

We note that the number of remote servers extracted by running malware samples in an emulated Android platform environment is restricted by the coverage of the execution paths (i.e, whether the apps meet specific conditions to execute the paths to communicate with remote servers.), a common limitation for all dynamic analysis based approaches. More specifically, it is challenging to trigger the apps to execute all networking connections, even with the usage of the current static/dynamic analysis tools. In our current work, to increase the coverage, we have implemented a UI fuzzing tool to simulate real users' usage of the apps, and inserted hundreds of UI events.

6.7 Summary

Similar to the measurement of the malicious OSN accounts' ecosystem, in this chapter, we described an in-depth empirical analysis of the market-level and network-level behaviors of the Android malware ecosystem. We observed find that neither

existing Android malware blacklists nor IP/domain blacklists are effective to stop malware authors from submitting malicious apps to Android markets. We also observed that malware authors tend to submit multiple malware samples within a short time period, and discovered a few communities that are in charge of a large portion of Android malware. We further proposed an effective algorithm to infer more malicious apps by starting from a seed set of known malicious ones and exploiting the properties of their community relationships. This approach is promising because it requires the disassembling of Android apps nor the deep domain knowledge of the Android system.

7. AUTOMATED MINING MALICIOUS BEHAVIORS IN ANDROID APPLICATIONS

Besides providing an inference-based algorithm to find more malicious Android apps, this chapter further presents a more complete detection scheme by dissembling Android apps and deeply analyzing the programming procedure used in known Android malware.

We introduce **DroidMiner**, a new approach to salably detect and characterize Android malware through robust and automated learning of fine-grained programming logic and patterns in known malware. Specifically, DroidMiner extends traditional static analysis techniques to map the functionalities of an Android app into a two-tiered behavior graph. This two-tiered behavior graph is specialized for modeling the complex, multi-entity interactions that are typical for Android applications. Within this behavior graph, DroidMiner automatically identifies *modalities*, i.e., programming logic segments in the graph that correspond to known suspicious behavior. The set of identified modalities is then used to define a modality vector. DroidMiner then uses common modality vectors to offer a more robust classification scheme, in which variant applications can be grouped together based on their shared patterns of suspicious logic.

We present and implement a prototype of DroidMiner for discovering and automatically extracting malware modalities. While our efforts are primarily focused on identifying and then characterizing malware behavior, aspects of our methodology are also directly applicable to automated characterization of a broad class of Android application behaviors, including the detection of shared security vulnerabilities [22]. We evaluate DroidMiner using 2,466 malicious apps, identified from a corpus of over

67,000 third-party market Android apps, plus an additional set of over 10,000 official market Android apps from *GooglePlay [48]*. Specifically, we measure the utility of DroidMiner modalities with respect to three specific use cases: (*i*) malware detection, (*ii*) malware family classification, and (*iii*) malware behavioral characterization. Our results validate that DroidMiner modalities are useful for classification and capable of isolating a wide range of suspicious behavioral traits embedded within parasitic Android applications. Furthermore, the composite of these traits enables a unique means by which Android malware can be identified with a high degree of accuracy.

We make the following contributions:

- A description of our new two-tiered behavioral graph model for characterizing Android application behavior, and labeling its logical paths within known malicious apps as malicious modalities.
- The design and implementation of DroidMiner, a novel system for automated extraction of Android app modalities, and using machine learning strategies to classify a given app under the modality pattern of a known malware family.
- An in-depth evaluation of DroidMiner with respect to its run-time performance and efficacy in malware detection, family classification, and behavioral characterization.

7.1 Motivation and System Goals

Program analysis techniques (e.g., data flow analysis and control flow analysis) have been widely used to analyze and detect traditional malware. Kolbitsch et al. proposed to detect host-based malware by extracting malware's behavior graph through analyzing the function-call flow [60]. Fredrikson et al. proposed to utilize control flow to extract discriminating specifications to identify a class of malware

[41]. Christodorescu et al. proposed to mine specific malicious behavioral patterns (such as decryption loops) from tracking the data flow and control flow of malware [26, 25].

7.1.1 Case Study

We motivate our system design by introducing the inner working of a real-world malicious Android application. This malware sample (*MD5: c05c25b769919f d7f1b12b4800e374b5*) belongs to the family of ADRD (a.k.a HongTouTou). It attempts to perform the following malicious behaviors in the background after the phone is booted: stealing users' personal sensitive information (e.g., IMEI and IMSI) and sending them to remote servers, sending and deleting SMS messages, downloading unsolicited apps, and issuing HTTP search requests to increase websites' search rankings on the search engine.

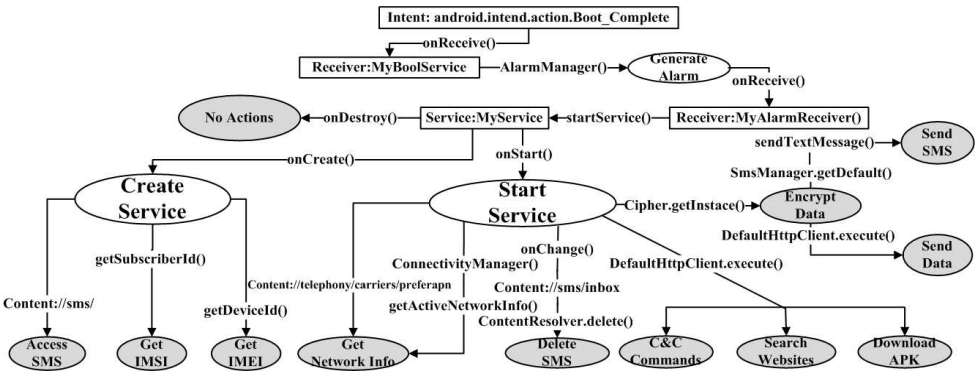


Figure 7.1: Capabilities embedded in malware from the ADRD family.

As illustrated in Figure 7.1, HongTouTou registers a receiver (named "MyBoolService") to receive the boot intent `BOOT_COMPLETED` message. Once the phone is booted, the receiver will send out an alarm every two minutes and trigger another

receiver (named "MyAlarmReceiver") by using three API calls: `AlarmManager()`, `getServiceSystem()`, and `getBroadcast()`. Then, MyAlarmReceiver starts a background service (named "MyService") by calling `startService()` in its lifecycle call `onReceive()`. Once the service is triggered through `onCreate()` or `onStart()`[1], it will read the device ID (`getDeviceId()`) and subscriber ID (`getSubscriberId()`) in the phone, and register an object handler to access the short message database `content://sms/`)). Before sending out sensitive information and communicating with the C&C server, the service obtains network information (e.g., network types such as "CMWAP", "UNIWAP" and "wifi") by invoking two Framework API calls: `ConnectivityManager()` and `getActiveNetworkInfo()`, and reading the content provider `content://telephony/carriers/preferapn`. It then encrypts personal information by using `Cipher.getInstance()`, `Cipher.init()` and `Cipher.doFinal()`, and exfiltrates encrypted data through SMS by using `SmsManager.getDefault()` and `sendTextMessage()`, and issuing HTTP requests. Meanwhile, the service monitors the changes to the SMS Inbox database by calling `onChange()` and deleting particular messages using `delete()`. Finally, it also attempts to download unsolicited APK files (e.g., "myupdate.apk"), to receive C&C commands and data, and to visit search engines by issuing HTTP requests.

The above description motivates an important design premise that when malware authors design malicious apps to achieve specific malicious behaviors, they typically require the use of sets of framework API calls and specific resources (e.g., content providers). More specifically, although attackers may attempt to launch malicious behaviors in a more surreptitious way, they would still have to use those framework APIs or access those important resources.

[1]If the service is triggered as the first time, it will call onCreate and onStart; otherwise, it will only call onStart.

7.1.2 Goals and Assumptions

The goal of DroidMiner is to automatically, effectively and efficiently mine Android apps and interrogate them for potentially malicious behaviors. Given an unknown Android app, DroidMiner should be able to determine whether or not it is malicious. Going beyond just providing a yes or no answer, our system should be able to provide further evidence as to why the app is considered as malicious by including a concise description of identified malicious behaviors. This kind of information is typically considered the hallmark of a good malware detection system. For example, DroidMiner can inform us that a given app is malicious, and that it contains behaviors such as sending SMS messages and blocking certain incoming SMS messages. With such information, an informed analyst could further infer that this is probably a money-stealing app that uses SMS to register for a premium service, spends money, and then suppresses the end-user notification.

The input into our system is an Android application developed with the Android SDK. Currently, we do not analyze native Android applications implemented using the Android Native Develop Kit (Android NDK). According to our observations, an overwhelming majority of Android applications today are developed using the Android SDK. Furthermore, the vast majority of malicious behaviors in Android apps are achieved by using Android SDK rather than Android NDK. Even for those malicious apps that use the NDK to achieve some malicious behaviors, they typically also use certain Android Framework APIs to obtain some auxiliary information. For example, "rooting" malware (e.g., samples in the family of DroidKungFu), which utilizes native code to achieve privilege escalation, still needs to use specific Framework APIs to obtain auxiliary information (e.g., the version of the operating system) to successfully root the phone. Hence, the presence of such APIs in the Dalvik bytecode

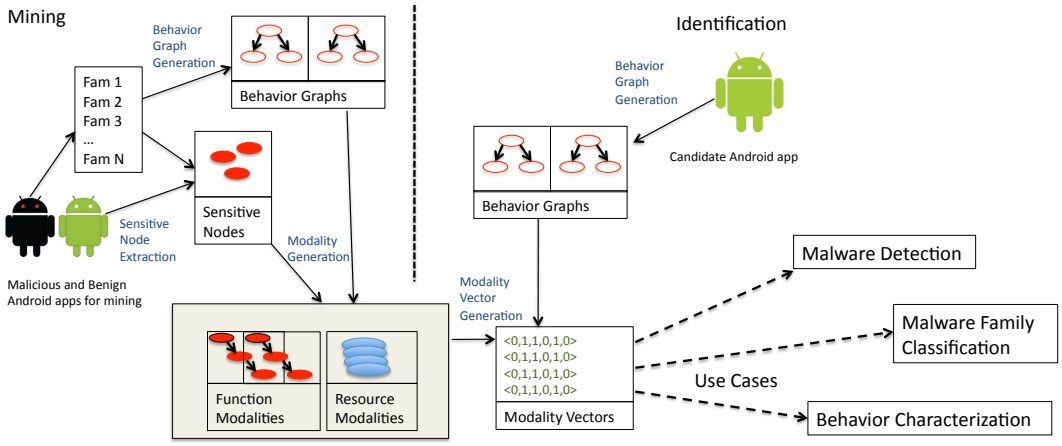


Figure 7.2: DroidMiner System Architecture

could still provide hints for detecting such malware. Extending our system to include complete analysis of native code in Android applications is future work and outside the scope of this dissertation.

7.2 System Design

DroidMiner contains two phases: Mining and Identification. As illustrated in Figure 7.2, in the mining phase, DroidMiner takes both benign and malicious Android apps as input data and automatically mines malicious behavior patterns or models, which we call *modalities*. In the identification phase, our system takes an unknown app as input, extracts a Modality Vector (MV) based on our trained modalities, and outputs whether or not it it is malicious, and which family it belongs to. In addition to a simple yes/no answer, our system can also characterize the behaviors of the app given the Modality Vector representation.

An important component in our system is the Behavior Graph Generator, which takes an app as input and outputs a behavior graph representation. As the analysis of a real-world malicious app shown in Figure 7.1, although Android malware

authors have significant flexibility in constructing malicious code, they must obey certain specific rules, pre-defined by the Android platform, to realize malware functionality (e.g., using particular Android framework APIs and accessing particular content providers). These framework APIs and sensitive content providers capture the interactions of Android apps with Android framework software or phone hardware, which could be used to model Android apps' behaviors. With this intuition, DroidMiner builds a behavior graph based on the analysis of Android framework APIs and content providers used in apps' bytecode.

In the Mining phase, DroidMiner will attempt to automatically learn the malicious behaviors/patterns from a training set of malicious applications. The basic intuition is that malicious apps in the same family will typically share similar functionalities and behaviors. DroidMiner will examine the similarities from the behavior graphs of these malicious apps and automatically extract common subsets of suspicious behavior specifications, which we call *modalities*. From an intrusion detection perspective, these modalities are essentially micro detection models that characterize various suspicious behaviors found in malicious apps. We provide more detailed descriptions in Section 7.2.2.

In the Identification phase, DroidMiner will transform an unknown malicious application into its behavior graph representation (using Behavior Graph Extractor) and extract a Modality Vector (based on all trained modalities), described in Section 7.2.3. Then, DroidMiner can apply machine-learning techniques to detect whether or not the app is malicious. DroidMiner also has a data-mining module that implements Association Rule Mining to automatically learn the behavior characterization of a given Modality Vector, described in Section 7.2.4.

7.2.1 Behavior Graph and Modality

7.2.1.1 Behavior Graph

DroidMiner detects malware by analyzing the program logic of sensitive Android and Java framework API functions and sensitive Android resources. To represent such logic, we use a two-tiered graphical model. As shown in Figure 7.3, at upper tier, the behaviors (functionalities) of each Android app could be viewed as the interaction among four types of components (Activities, Services, Broadcast Receivers, and Content Observers). We represent this tier using a **Component Dependency Graph (CDG)**. At the lower tier, each component has its own semantic functionalities and a relatively independent behavior logic during its lifetime. Here, we represent this independent logic using **Component Behavior Graphs (CBG)**.

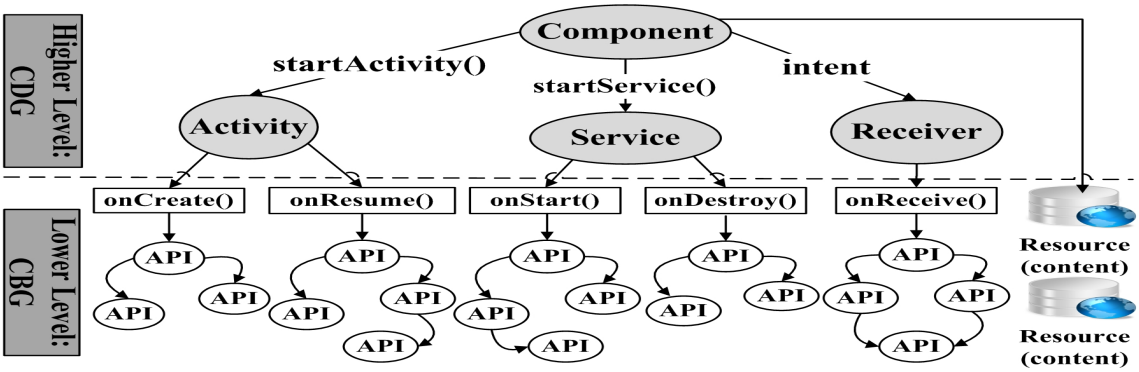


Figure 7.3: Two-tier behavior graph.

The **Component Dependency Graph (CDG)** (upper tier of Figure 7.3) represents the interaction relationships among all components in an app. In particular, each node in the CDG is a component (Activity, Service, or Broadcast Receiver). (Note that multiple nodes could belong to the same type of component.) There is

an edge from one node v_i to another node v_j, if the component v_i could activate the start of component v_j's lifecycle. For example, in terms of the malware sample illustrated in Figure 7.1, since *MyAlarmReceiver* could activate *MyService* by using `startService()`, its CDG has an edge from a broadcast receiver node *MyAlarmReceiver* to a service node *MyService*.

The **Component Behavior Graphs (CBG)** (lower tier of Figure 7.3) represents each component's *lifetime* [2] behavior logic (functionalities), i.e., each CBG represents the control-flow logic of those permission-related Android and Java API functions, and actions performed on particular resources of each component. Specifically, as illustrated in Figure 7.3, a CBG contains four types of node:

- A *root* note (v_{root}), denoting the component itself (e.g., one Activity or one Service).
- *Lifecycle functions* (V_{lcf}), used to achieve the runtime logic of specific type of component (e.g., `onCreate()` in an activity, `onReceive()` in a receiver, and `onStart()` in a service).
- *Permission-related API functions* (V_{pf}), representing those permission-related (Android SDK or Java SDK) API functions (e.g., Java API `Runtime.execute()` or Android API `sendTextMessage()`). For simplicity, in the rest of this chapter, we refer both lifecycle functions and API functions as *framework API functions*.
- *Sensitive resource* (V_{res}), i.e., sensitive data (files or databases) that are accessed by the component. In this dissertation, we consider resources as content providers (e.g., `content://sms/inbox/`), which could be extended to any

[2]Lifetime, as defined by the Android, is time between the moment when the Android OS considers a component to be constructed and the moment when the Android OS considers the component to be destroyed.

other type of sensitive data. The usage of framework API functions and sensitive resources in an app essentially captures the interactions of an app with the Android platform hardware and sensitive data. Hence, the control-flow logic of framework API functions and the actions performed on those sensitive resources reflect an application's range of capabilities.

The edges in CBG represent the control-flow logic of framework API functions and sensitive resources. In terms of framework API functions, we consider that there is a direct edge from function node v_i to v_j in the CBG, if (1) when v_i and v_j are in the same control-flow block, v_j is executed just after v_i with no other functions executed between them; or (2) when v_i and v_j are in two continuous control-flow blocks B_i and B_j respectively (i.e., B_j follows B_i), v_i is the last function node in B_i and v_j is the first node in B_j. Then, we call v_j "is a successor of" v_i. For example, in terms of the malware sample illustrated in Figure 7.1, there is an edge from `smsManager.getDefault()` to `sendTextMessage()`. In terms of sensitive resources, since our work mainly focuses on analyzing the control-flow of sensitive functions rather than the data flow of sensitive data, we simply consider that there is an edge from the root to the resource v_r, if the component uses that sensitive resource[3].

7.2.1.2 Modality

We use the term, *modalities* to refer to malicious behavior patterns that are mined from behavior graphs of Android malware. More specifically, each modality is an ordered sequence of framework API functions (function modality) or a set of sensitive resources (resource modality) in commonly shared in malicious apps' behavior graphs[4], which could be used to implement suspicious activities (e.g., sending SMS

[3]We could also choose to build an edge from a framework API function (that uses that resource) to the resource, which relies on the data flow analysis.

[4]Although modalities described in this dissertation are localized within a CBG, our work could be extended to include cross CBG modalities with the usage of CDG.

messages to premium-rate numbers or stealing sensitive information). As an example, the malware sample illustrated in Figure 7.1 relies on a function modality with an ordered sequence of two framework functions (`onChange()` → `ContentResolver.delete()`), and a resource modality (`content://sms/inbox/`) to partially achieve the malicious behavior of deleting messages in the SMS inbox.

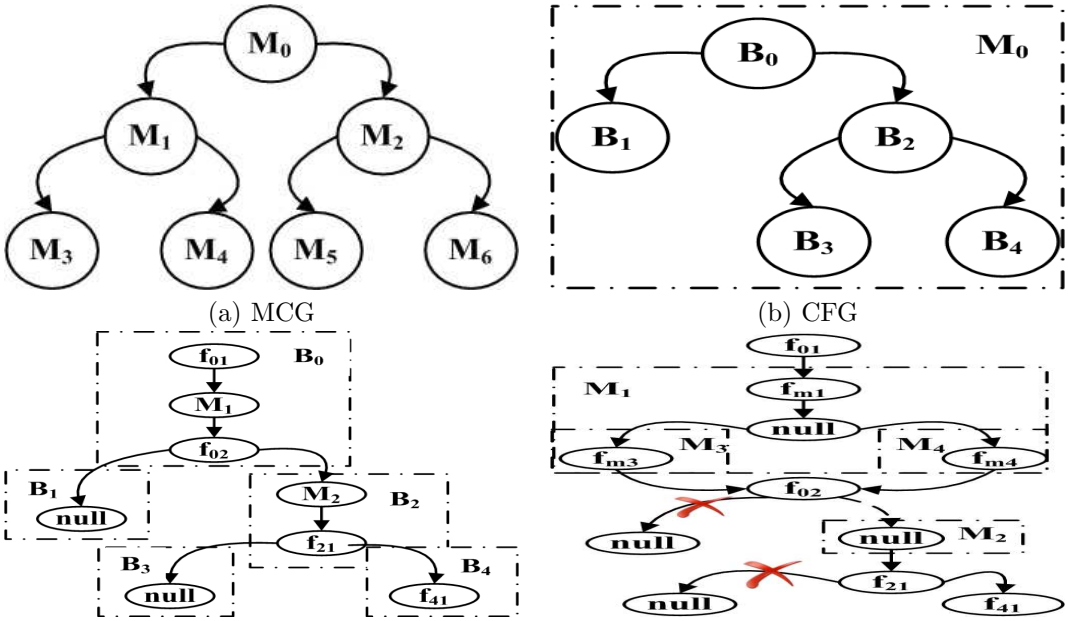
7.2.2 Mining Modalities

Our desire to conduct efficient mining of modalities from large malware corpora calls for an automated approach to mining malicious patterns. We now describe the details of our modality mining process, which involves the following three steps: Behavior Graph Generation, Sensitive Node Extraction, and Modality Generation.

7.2.2.1 Behavior Graph Generation

The generation of the behavior graph of an app contains two phases: generating CDG and generating CBG. The generation of CDG is relatively straightforward. The nodes in an app's CDG are acquired by analyzing activities, receivers, and services registered in its manifest file ("AndroidManifest.xml"). As a special case, DroidMiner extracts runtime the Broadcast Receiver by analyzing instances of `Context.registerReceiver()` instead of parsing the manifest file. Much like [139], DroidMiner acquires the edges of an app's CDG by analyzing the usage of intents in each component. For example, an intent used in `startActivity(Intent)` can activate an activity; an intent used in `startService(Intent)` can start a background service.

Since Android is component driven, and each component has its own lifetime execution logic, the extraction of control-flow logic of framework API functions (rather than the control-flow logic of methods in traditional program analysis) described in the model of our CBG is more complex, which involves the following three steps: Generate Method Call Graph, Generate Control-Flow Graph, and Replace User-Defined



(a) MCG (b) CFG

(c) Transformed CFG (d) CBG with API functions

Figure 7.4: Illustration of generating a CBG with framework API functions.

Methods:

- **Step 1: Generate Method Call Graph.** For each component, our system generates a method call graph (MCG) containing two types of nodes: Android lifecycle functions and user-defined methods. Since each type of component has fixed lifecycle functions (e.g., `onCreate()` in an Activity), DroidMiner extracts lifecycle functions by analyzing method names in the component according to the type. Those user-defined methods could be identified by using a static analysis tool. As illustrated in Figure 7.4(a), there is a directed edge from method M_0 to M_1, which implies M_0 calls M_1.

- **Step 2: Generate Control-Flow Graph.** To extract the programming logical usage of framework API calls, DroidMiner first extract each method's control-flow graph (CFG) via identifying branch-jump instructions in the method's

bytecode (e.g., `if-nez` or `packed-switch`). Each node is a block of Dalvik bytecode without any jump-branch instructions. For example, M_0 with five blocks is illustrated in Figure 7.4(b). There is a directed edge from block B_0 to B_1, if B_1 is a successor block of B_0. Then, each block is represented as ordered sequence of framework API functions and user-defined methods, which are extracted from the Dalvik bytecode with function call instructions (e.g., `invoke-direct`). We label a block as "null", if it does not contain any function call instructions . For example, in the method M_0, if (1) B_0 contains two API functions and user-defined method M_1, with the execution order of f_{01}, M_1 and f_{02}; (2) B_1 and B_3 do not contain any function calls; (3)B_2 contains method M_2 and one API function f_{21}; (4) B_3 contains one API function f_{41}, then the control-flow graph of M_0 could be formed as Figure 7.4(c).

- **Step 3: Replace User-Defined Methods.** As illustrated in Figure 7.4(c), since each leaf in the method-call graph does not call any other user-defined method, the leaf either contains a subgraph of framework API functions or is "null". Then, our approach replaces its position in its parents' control-flow graphs with that subgraph. This process is recursively performed, until all user-defined methods are replaced with framework API functions. For example, if (1) M_1 contains three framework API functions (f_{m1}, f_{m3}, and f_{m4}) and one "null" node after replacing its children methods M_3 and M_4 as illustrated in the middle of Figure 7.4(d), and M_2 does not contain any function nodes, then after replacing its children methods M_5 and M_6, the graph will be transformed to Figure 7.4(d). Finally, the CBG will be generated by removing those leaves that are "null". After the above three steps, each app's CBG could be generated that represents the control flow of its framework API calls.

7.2.2.2 Sensitive Node Extraction

A modality is essentially an ordered sequence of framework API functions and a set of sensitive resources that are commonly observed in malicious apps behavioral graphs. We denote those framework API functions and sensitive resources as sensitive nodes (the former are called *sensitive function nodes*, while the latter are called *sensitive resource nodes*).

We use two strategies to *automatically* extract sensitive nodes. The first strategy is based on the observation that malware samples belonging to the same family tend to share similar malicious logic. Such an observation has been validated by a recent study, which reports that Android malware in the same family tends to hide in multiple categories of fake versions of popular apps. Based on this intuition, we group known malware samples according to their families. (Note that the process of deriving the family label for known malware is only used in the mining phase and depends on the way of collecting malware. DroidMiner automatically acquires the malware's family label by parsing antivirus reports. More details are provided in Section 7.3.2).

Then, for each malware family, we extract function nodes and resource nodes that are commonly shared by at least $\theta\%$ members in this family [5].

Our second strategy is based on the observation that malware samples hosted on third-party market websites tend to be parasitic, i.e., they masquerade as popular benign apps by injecting malicious payloads into original benign apps. Based on this intuition, we automatically extract sensitive nodes by calculating the (δ), i.e., additional bytecode between the known malicious app and official Android apps sharing similar application names. The official apps are acquired by automatically

[5]In our preliminary experiments, we set the threshold as 30%.

searching for known malicious app names in GooglePlay. (We skip this process for known malware whose names are not registered in GooglePlay.)

In practice, our two strategies can be complementary. To detect malicious apps, our approach relies on the control-flow logic of these sensitive nodes. Also, the effect of those false positive sensitive nodes could be further decreased when we add benign apps in training the detection model to decrease the weight of those benign patterns. In terms of the false negatives induced by the second strategy, although not all apps from the *GooglePlay* are benign, this market is still the only official one with the best reputation for Android apps. Also, if the known malware family set contains sufficient malware samples, those missed patterns through the comparison with official apps could still be found by using the first strategy.

7.2.2.3 Modality Generation

As defined in Section 7.2.1, we now detail how we automatically generate function modalities and resource modalities.

Intuitively, our system generates function modalities by mining an ordered sequence (path) of sensitive function nodes from known malware samples' behavior graphs, as illustrated in Figure 7.2. In particular, for each path of each known malware's CBG, we denote a subpath of it as a sensitive path, if it starts from one sensitive function node and ends with another sensitive function node. Then, after *removing those non-sensitive nodes* sitting in the middle of the sensitive path, we generate function modalities from the transformed sensitive path by extracting all of its subsequences. Generating function modalities involves the following two steps: Extract Sensitive Path and Extract All Subsequences.

- **Step 1: Extract Sensitive Path.** For each pair of sensitive nodes S_i and S_j, we extract sensitive paths P_{ij} of framework API functions from all known

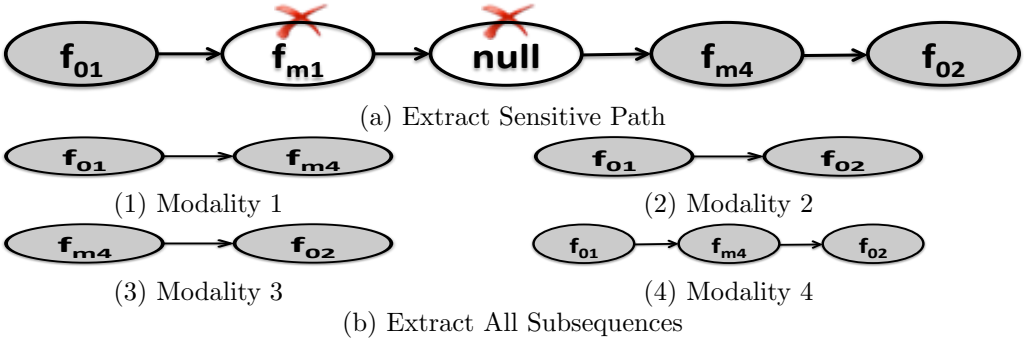
malware samples' CBGs, if P_{ij} starts from S_i and ends with S_j. In particular, for each path in the malware's CBG, we generate modalities from the longest sensitive path, which will cover the results extracted from those shorter sensitive paths. As an illustrative example in Figure 7.4(d), if f_{01}, f_{m4} and f_{02} are sensitive nodes, the longest sensitive path could be illustrated as Figure 7.5(a). Then, we could generate a transformed path of function nodes, through removing non-sensitive nodes in the middle. In the previous example, a transformed sensitive path $f_{01} \rightarrow f_{m4} \rightarrow f_{02}$ can be extracted by removing two non-sensitive nodes f_{m1} and "null" in the middle.

- **Step 2: Extract All Subsequences.** We generate function modalities by extracting all *order-preserving*[6] subsequences of the transformed path of sensitive function nodes. Accordingly, we could mine four function modalities from the previous example (see Figure 7.5(b)). Since DroidMiner utilizes *all subsequences* to generate the modalities instead of using the original single long sequence/path, DroidMiner is resilient to many evasion attempts by malware, e.g., insertion of loop framework API calls in the middle that serve no purpose other than adding noise. Hence, our modalities are a more robust representation of specific malware programming logic than using simple call sequences or frequencies.

7.2.3 Identification of Modalities

After mining modalities, the second phase of DroidMiner involves the identification of modalities in unknown apps (i.e., determine which modalities are contained in unknown apps). As illustrated in Figure 7.2, for each unknown app, DroidMiner

[6]This implies that the order of two function nodes in the subsequece remains the same as in the original path.



(a) Extract Sensitive Path

(1) Modality 1 (2) Modality 2

(3) Modality 3 (4) Modality 4

(b) Extract All Subsequences

Figure 7.5: An illustration of function modality generation.

identifies its modalities by extracting its behavior graph and generating a Modality Vector, specifying the presence of mined modalities.

More specifically, for each unknown app, DroidMiner generates its behavior graph and extracts sensitive paths from the graph. Then, DroidMiner obtains all potential sub-paths by generalizing those sensitive paths. For each sub-path, if it is a modality (belonging to the mined modality set), we consider this app to contain this modality. This process of modality extraction is highly efficient due to the limited number of sensitive nodes present in each app.

In this way, once M different modalities are mined from known malware samples, each app could be transformed into a boolean vector $(X_1, X_2, \ldots, X_M)$, denoted as a "Modality Vector": $X_i = 1$, if the app contains the modality M_i; otherwise, $X_i = 0$. In this way, an app's Modality Vector could represent its spectrum of potentially malicious behaviors.

7.2.4 Modality Use Cases

We introduce how to use an Android app's Modality Vector to address the following three use-case scenarios: Malware Detection, Malware Family Classification, and Malicious Behavior Characterization.

7.2.4.1 Malware Detection

The first use case involves simply determining whether or not an Android app is malicious. In fact, it is challenging to make a confirmative decision. For example, although some sensitive behaviors (e.g., sending network packets or SMS messages to remote identities) are commonly seen in malware, without a deep analysis about such behaviors (e.g., the analysis of the reputation of those remote identities), we cannot blindly declare all apps with such behaviors to be malware. However, as seen in Table 7.6, Android malware typically needs to use multiple sensitive functions (or modalities) to achieve its objectives: e.g., (*i*) sending SMS AND blocking notifications or (*ii*) rooting the phone AND installing new apps.

According to this observation, DroidMiner considers an app to be malicious only if the cumulative malware indication from all of its modalities exceeds a sufficient threshold. That is, the single usage of one modality in a benign app will not cause it to be labeled as malware. We use machine learning techniques (described in Section 7.3) to learn the indication of each modality used in the cumulative scoring process. More specifically, we consider each of mined modalities as one detection feature in the machine-learning model. Thus, the number of detection features is equal to the dimensionality of the *Modality Vector*. By feeding modality vectors extracted from known malware and benign apps into the applied machine-learning classifier, the indication of those modalities that are highly correlated with malicious apps are up-weighted in judging an app to be malicious; those modalities that are also commonly used in benign apps are down-weighted.

DroidMiner could also be designed to detect malware using pre-defined (strict) detection rules, like policy-based detection systems, which may lead to a lower false positive rate. However, such a policy-based design requires considerable domain

knowledge and comprehensive manual investigations of malware samples, which can limit overall scalability and thus is more suitable to be applied to detect specific attacks. Our goal of designing a fully automated approach motivated us to use the learning-based approach instead of policy-based ones.

7.2.4.2 Malware Family Classification

Another use case is automatically determining which malware family an malicious app that is determined to belong to. This problem is also important for understanding and analyzing malware families. In fact, many antivirus vendors still rely on common code extraction techniques, which typically manually extract signatures after gathering a large collection of malware samples belonging to the same malware family.

Different malware samples in the same family tend to share similar malicious behaviors, which could essentially be depicted by *Modality Vectors*. Thus, the degree of similarity between the Modality Vectors of two malware samples provides an indication of whether these two samples belong to the same family. Hence, with the knowledge of Modality Vectors mined from malware samples belonging to existing malware families, we could build a malware family classifier for unknown malicious apps by using machine learning techniques.

7.2.4.3 Malicious Behavior Characterization

The final use case involves characterizing the specific malicious functionality that is embedded within a candidate app. To solve this problem, we essentially need to know which modalities could be used to achieve specific malicious behaviors. Then, if an app contains those modalities, we could claim with high confidence that the app is malicious.

To realize this goal, we use a well-known data mining technique, called "Associ-

ation Rule Mining". The problem of association rule mining is defined as follows: Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of binary attributes. Let $B = \{B_1, B_2, \ldots, B_m\}$ be a set of items, where $B_i = \{a_{i1}, a_{i2}, \ldots, a_{in}\}$. A rule is defined as an implication of the form $X \Rightarrow Y$, where $X, Y \subseteq A$ and $X \cap Y = \phi$. The attribute sets X and Y are called antecedent and consequent of the rule, respectively. It represents the scenario that if the attributes in X are true, then the attributes in Y are also true. The support $supp(X)$ of an attribute set X is defined as the proportion of items in the item set whose attributes in X are all true. The confidence of a rule is defined as $conf(X \Rightarrow Y) = supp(X \bigcup Y)/supp(X)$, which could be interpreted as an estimate of the probability $P(Y|X)$.

We abstract this as an Association Rule Mining problem, i.e., we need to mine relationships (association rules) from modalities to malicious behaviors. More specifically, DroidMiner derives association rules by analyzing the relationship between the modality usage in existing known malware families and their corresponding malicious behaviors. e.g., Zsone has two known malicious behaviors: (i) sending SMS and (ii) blocking SMS. Hence, we attempt to associate modalities generated from this family to these two behaviors.

Our assumption is that in most cases malware samples belonging to particular malware families tend to express similar malicious behaviors. (While our ground truth may not be perfect, we believe that this assumption will be valid for most cases.) More specifically, given a set of modalities $M = \{M_1, M_2, \ldots, M_n\}$ and a set of malware samples $S = \{S_1, S_2, \ldots, S_n\}$ with their malware family names, for each malware sample S_i, we extract its Modality Vector $SM_i = \{M_{i1}, M_{i2}, \ldots, M_{ip}\}$. Given a set of malicious behaviors $B = \{B_1, B_2, \ldots, B_q\}$, we generate a behavior vector for S_i, $B_i = \{B_{i1}, B_{i2}, \ldots, B_{iq}\}$, where $B_{ik} = 1$, if S_i's family contains malicious behavior B_k; otherwise $B_{ik} = 0$. Accordingly, as illustrated in Table 7.1, we build

a behavior matrix $BM_{n\times(p+q)}$ by setting: (1) $BM_{i,j} = (S_i, M_j)$ denotes whether ith malware sample in the malware set contains jth modality in the modality set, where $1 \leq i \leq n$ and $1 \leq j \leq p$; (2) $BM_{i,p+k} = B_{ik}$, where $1 \leq i \leq n$ and $1 \leq k \leq q$.

	M_1	M_2	...	M_p	B_1	...	B_q
S_1	0	1	...	1	0	...	1
S_2	1	0	...	0	1	...	0
S_3	0	1	...	0	0	...	1
...	...	...	...	...	...	...	...
S_n	0	0	...	1	1	...	1

Table 7.1: An example of behavior matrix.

Thus, the problem of *identifying which modalities could be used to achieve the malicious behavior* B_k could be transformed to the following problem:

Finding a set of modalities, $M_k = \{M_i | M_i \in M, 1 \leq i \leq m\}$,

s.t., $C(M_k) = conf(M_k \Rightarrow B_{p+k}) = supp(M_k \bigcup B_{p+k})/supp(M_k) \geq T_{conf}$, *where* T_{conf} *is a pre-defined threshold.*

Then, we consider the set of modalities M_k that could be used to achieve malicious behavior B_k with a confidence score of $C(M_k)$. Accordingly, we mine association rules from modalities to malicious behaviors with high confidence scores and sufficient support scores, and apply them to candidate malicious apps to characterize their malicious behaviors.

7.3 Evaluation

We present our evaluation results by implementing a prototype of DroidMiner and applying it to apps collected from existing third-party Android markets and from the official Android market (*GooglePlay*).

7.3.1 Prototype Implementation

We implement a prototype of DroidMiner on top of a popular static analysis tool (Androguard [45]). In our experience, comparing with other public Android app decompilers (e.g., Dex2Jar [47] or Smali [50]), Androguard produces more accurate decompilation results, especially in terms of handling exceptions. The prototype decompiles an Android app into Dalvik bytecode, further builds its behavior graph and mines its modalities based on the bytecode.

The method call graph in an app is built by analyzing the caller-callee relationships of all methods used in the app. For each method, DroidMiner extracts its callee methods by analyzing the *invoke-kind* instructions (e.g., *invoke-virtual* and *invoke-direct*) used in the method. Since Android is an event-driven system, the entrance of an app could be UI event methods (e.g., `onClick`) instead of lifetime cycle methods. However, such UI event methods could only be executed after the corresponding UI event listeners are registered (e,g., `setOnClickListener`). Thus, to make the program logic more complete, DroidMiner adds an edge from UI events listeners to corresponding UI event methods, although there is no such caller-callee relationship in the bytecode. We use a similar strategy to address registered event handlers by linking the handle method (e.g., `handleMessage`) to its corresponding construction method (e.g., Landroid/os/Handler.init). We also modify Androguard to generate the control-flow graph in each method by analyzing branch jump instructions (e.g., `if-eq`).

As an illustrative example, Figure 7.6 shows part of Dalvik code for the method Myservice.onCreate() used in the malware sample described in Section 7.1.1. From Line 1, DroidMiner will build an edge from `Myservice.onCreat()` to `Myservice.getSystemService()` in its method call graph. Frome Line 9, which contains a

```
1  invoke-virtual v8, v3, Lcom/xxx/yyy/MyService;->getSystemService(Ljava/lang/String;)
2  move-result-object v2
3  check-cast v2, Landroid/telephony/TelephonyManager;
4  invoke-virtual v2, Landroid/telephony/TelephonyManager;->getDeviceId()
5  move-result-object v3
6  iput-object v3, v8, Lcom/xxx/yyy/MyService;->imei Ljava/lang/String;
7  invoke-virtual v2, Landroid/telephony/TelephonyManager;->getSubscriberId()
8  iget-object v3, v8, Lcom/xxx/yyy/MyService;->smsObserver Lcom/xxx/yyy/SMSObserver;
9  if-nez v3, +1e
10 new-instance v3, Lcom/xxx/yyy/SMSObserver;
11 new-instance v4, Landroid/os/Handler;
12 invoke-direct v4, Landroid/os/Handler;-><init>()V
13 invoke-direct v3, v4, v8, Lcom/xxx/yyy/SMSObserver;-><init>
14 iput-object v3, v8, Lcom/xxx/yyy/MyService;->smsObserver Lcom/xxx/yyy/SMSObserver;
15 invoke-virtual v8, Lcom/xxx/yyy/MyService;->getContentResolver()
16 move-result-object v3
17 const-string v4, 'content://sms/'
18 invoke-static v4, Landroid/net/Uri;->parse(Ljava/lang/String;)Landroid/net/Uri;
```

Figure 7.6: The Dalvik bytecode of the method `Myservice.onCreat()` used in a real-world malware with capabilities of reading device ID and accessing SMS.

branch-jump instruction (`if-nez`), DroidMiner will generate a new code block, while generating the control-flow logic. Two sensitive framework API functions will be recorded from Line 4 and Line 7, and one sensitive resource (content provider) will be recorded from Line 17. Thus each application's modalities could be mined through examination of its usage of framework API functions and content providers.

7.3.2 Data Collection

We crawled four representative marketplaces, including GooglePlay, and three alternative Android marketplaces (SlideMe [100], AppDH [5], and Anzhi [70]). The collection from the alternative Android markets occurred during a 13-day period, from June 3 through June 15, 2012. The GooglePlay collection was harvested during a two-months period, from August 23 through October 23. Our resulting app corpus is described in Table 7.2. In total, we collected 67,822 free apps, where 17% of the apps (11,529) were collected from GooglePlay, and the remaining 83% (56,268) were harvested from the alternative markets.

Next, we attempt to isolate the set of malicious apps from our corpus by submitting the set of apps from the alternative markets to "VirusTotal.com", which is a free

	Official Market	SlideMe	AppDH	Anzhi
Location	U.S.A	U.S.A	China	China
Number of Apps	11,529	15,129	2,349	38,790
Total Apps	11,529 (17%)	56,268 (83%)		
	67,797			

Table 7.2: The summary of collecting Android apps.

antivirus (AV) service that scans each uploaded Android app using over 40 different AV products [114]. For each app, if it has been scanned earlier by an AV tool, we can obtain the full VirusTotal report, which includes the first and last time the app was seen, as well as the results from the individual AV scans. For example, BitDefender has a report for a malicious application (*MD5: 7acb7c624d7a19ad4fa92cacfddd9257*) as `Droid.Trojan.KungFu.C`. In this way, we obtained 1,247 malicious apps identified by at least one AV product. For each malicious app, we extract its associated malware family name, and when AV reports disagree, we derive a consensus label using the label that dominates the responses from the AV tools. In addition, we obtain another set of malware samples from Genome Project [140, 141]. This dataset contains the family label for each malware sample. After excluding those already appeared in our crawled malware set, there are 1,219 different malware apps. Thus, in total, our malware dataset consists of *2,466* (1,247+1,219) unique malicious apps that belong to *68* different malware families.

In addition to the malware dataset, we also construct a benign dataset using popular apps collected from GooglePlay. To further clean this dataset, we submit our candidate set of 11,529 free GooglePlay apps to VirusTotal, of which 1,126 apps were labeled as malicious by one AV product. We discarded those apps and constructed our benign dataset using the remaining *10,403* free GooglePlay Android apps. Clearly, the benign app dataset may still contain some malicious apps, but

this set has at least been vetted by the GooglePlay anti-malware analysis and by more than 40 AV products from VirusTotal. The problem of producing a perfect benign app corpus remains a hard challenge, and we note that a similar approach to construct a benign app dataset has been used in prior related work [84].

7.3.3 Evaluation Result

Below, we summarize our system evaluation results for malware detection, malware family classification, behavior characterization, and efficiency.

7.3.3.1 Malware Detection

As introduced in Section 7.2.4, we utilize machine learning techniques to conduct malicious app detection. To better evaluate the effectiveness of DroidMiner, we utilize four widely used machine learning (ML) classifiers: *NaiveBayes*, *Support Vector Machine (SVM)*, *DecisionTree* and *Random Forest*. *NaiveBayes* is a probabilistic-based classifier. It is fast, easy to understand, and has been widely used in spam detection studies. Since this classifier relies on the assumption that each individual feature is distributed independently of other features, its main disadvantage is that it could not learn interactions between features. Accordingly, it is not very powerful when the feature set is complex and the training set is big with high variance. *SVM* is a kernel-function-based classifier, very popular for text classification problems. It could achieve a relatively high accuracy regarding over-fitting, especially when the number of the feature dimensions is very high. However, its performance is sensitive to the choice of the kernel functions and parameters.

Decision Tree and *Random Forest* are two rule-based classifiers. They are non-parametric and could easily handle feature interactions. Thus, they could achieve high performance, even when the data is not linearly separable. *Random Forest* considers the problem of over-fitting, which could perform better than *Decision Tree*.

Classifier	NaiveBayes		SVM	
Method	Permission[84]	DroidMiner	Permission[84]	DroidMiner
DR	75.1%	82.2%	78.8%	86.7%
FP Rate	7.2%	4.4%	3.5%	1.1%
Classifier	Decision Tree		Random Forest	
Method	Permission[84]	DroidMiner	Permission[84]	DroidMiner
DR	85.7%	92.4%	87.0%	95.3%
FP Rate	2.2%	1.0%	2.0%	0.4%

Table 7.3: Effectiveness of malware detection (DR denotes detection rate, FP denotes false positive).

Specifically, *Random Forest* is fast, scalable and often the winner for many problems in classification. Also, *Random Forest* does not require developers to excessively tune parameters as *SVM* does.

For each classifier, we conduct a series of experiments using a *ten-fold cross validation* to compute three performance metrics: *False Positive Rate*, *Detection Rate*, and *Accuracy*. Specifically, we divide both malicious and benign datasets randomly into 10 groups, respectively. In each of the 10 rounds, we choose the combination of one group of benign apps and malicious apps as the testing dataset, and the remaining 9 groups as the training dataset. We further compare the performance of DroidMiner with another classifier (used in [84]), which uses registered permissions as major detection features, based on our collected dataset.[7] Although [84] is mainly designed to rank apps' risks based on apps' registered permissions and categories, it also reports the true positive rate and false positive rate by choosing a particular risk value as indicative of malicious apps.

Table 7.3 shows the results of using permission versus DroidMiner based on dif-

[7]We are unable to provide a direct corpus comparative evaluation with other detection systems discussed in related work [142, 21], because they are not publicly available and it is generally difficult to completely reproduce similar systems and parameter selections.

ferent classifiers. We see that for all four classifiers, the usage of modalities as the input feature set (DroidMiner) produces a higher detection rate and lower false positive rate than the approach of using permission features [84]. In particular, using *Random Forest* DroidMiner achieved a detection rate of 95.3%, roughly 10% higher than the that of using permission. Furthermore, DroidMiner produced a lower false positive rate of (0.4%), or around 1/5th of the compared approach. Also, DroidMiner could maintain the detection rate higher than 86% for all four classifiers. In addition, we can see that *Decision Tree*, *Random Forest* and *SVM* could achieve better performance than *NaiveBayes* by using both permissions and modalities as inputs, mainly because the features (both permissions and modalities) are not totally linear separable. In terms of permission, particular permissions with semantic coordination are often granted together (e.g., `SEND_SMS` and `RECEIVE_SMS`). In terms of modalities, a shorter (more general) modality may be a part of a longer (more specific) modality. Also, since *Random Forest* could solve over-fitting without the need of tuning parameters, its performance could beat *Decision Tree* and *SVM*.

We next compare the average training time used for each classifier with DroidMiner. As seen in Table 7.4, we find that the training time used for all four classifiers could be maintained lower than 150 seconds. Particularly, although *NaiveBayes* could not achieve an accuracy as high as other classifiers, it is the fastest one (taking only 0.15 seconds) to train the model, which validates what have we discussed about this classifier. We see that *Random Forest* is both fast (taking only 8.15 seconds) and accurate.

Classifier	NaiveBayes	SVM	Decision Tree	Random Forest
Time (s)	0.15	141.21	76.08	8.15

Table 7.4: Training time (in seconds).

Further, to understand false positives/negatives, we randomly choose 20 false negatives and 15 false positives generated in the case of *Random Forest* for further investigation that were induced in the first two rounds of our ten-fold cross validation experiment. Through manually analyzing these apps, we find four possible reasons that induce those false negatives: (*i*) Adware: we find DroidMiner missed identifying 11 instances of adware (seven belong to *Leadbolt* and four belong to *Airpush*), due to the diverse implementation of those adware examples. (*ii*) Native code: Since DroidMiner relies on the static analysis on the Dalvik code, it generated four false negatives that utilize native code to achieve malicious goals (e.g., rooting the phone). (*iii*) Dynamic payload: we also find four malware instances that will dynamically launch malicious payloads by either downloading from the remote servers (e.g., *Plankton*) or modifying local files (*AnserverBot*). Since such malware initially does not contain (or activate) malicious payloads, DroidMiner could not detect them through statically analyzing Dalvik code. (*iv*) False label: we also found 1 false negative, which is labeled as malware belonging to the family of *Pjapps* by Sophos in our data collection phase. However, our manual analysis does not find any malicious payload from the app that could be seen in other apps belonging to this family. Then, we re-submited this app to VirusTotal again and found that Sophos has changed its description on this app and identified it as benign.

Similarly, we find that our false positives could be classified into four categories: (*i*) Eight apps from GooglePlay are identified as malicious because they could send out sensitive information. In particular, four apps (three Game apps and one Shopping app) sent out phone information (e.g., IMSI) or account information[8]; three apps sent out Geo-location information; the other app could send out the contact

[8]That could be because some Game or Shopping apps tend to use such information as the unique identifier to distinguish registered accounts.

information. (*ii*) Three apps could achieve sensitive functionalities as malware. Two of them could automatically monitor and send out the phone state, and even unlock the phone without using the password. The other one, named as "Task Manger", could kill the background process, start/restart an app, clean web browsing history, and so on. (*iii*) One app is essentially adware, which belongs to both *Leadbolt* and *Airpush*. (*iv*) Three other benign apps are falsely identified as malware.

7.3.3.2 Family Classification

The purpose of this experiment is to measure the accuracy of using Modality Vectors to correctly assign apps that are classified as malicious to their correct corresponding malware family. To conduct the malware family classification, we use samples from 12 families, each of which has more than 50 samples. The number of samples of each family is shown in Table 7.5.

Ind	**Family**	**Num**	**Ind**	**Family**	**Num**
1	GingerMaster	166	7	KMin	52
2	GoldDream	57	8	BaseBridge	122
3	Airpush	568	9	Geinimi	69
4	AnserverBot	187	10	DroidKungFu3	327
5	DroidKungFu	70	11	DroidKungFu4	104
6	Leadbolt	52	12	Plankton	194

Table 7.5: Malware samples used for classification.

For each family, we use half of the samples as training dataset, and the other half as the testing dataset. In this case, the classification accuracy represents the ratio of the number of correctly classified samples to the total number of samples in the test dataset. Here, we use *Random Forest* for classifying both the training and testing datasets. The classifier produces a relatively high classification accuracy of 92.07%.

Figure 7.7 shows the confusion matrix produced from our classification of the dataset into the malware family label set. The value of the cell (i, j) in the matrix shows the number of samples in family i, which are classified as being family j. Thus, the central diagonal in the matrix shows the number of *correctly* predicted samples per malware family. The darker the cell color is, the higher the classification accuracy is. With the exception of *Leadbolt* (index is 6), most of the other families achieve an accuracy higher than 90%. *Leadbolt* is an adware family, and thus its implementation may be influenced by the campaign it is serving, and thus producing a behavior that has a wide variability, leading its samples to appear to match a wider range of potential families.

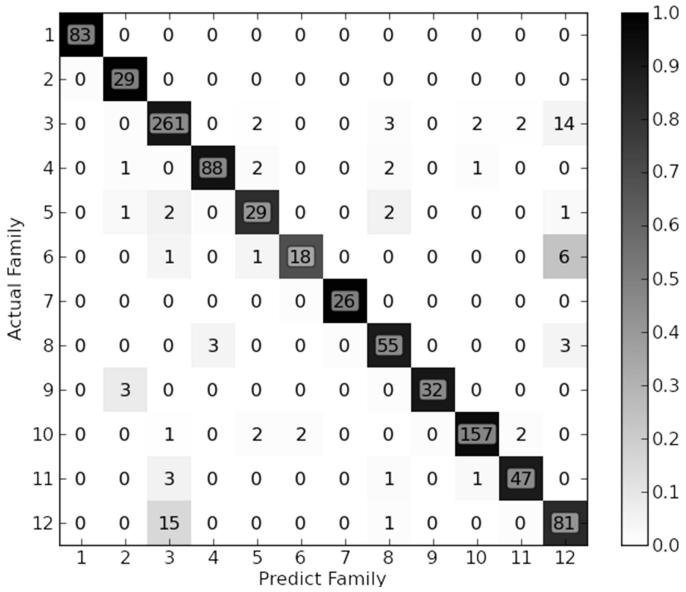


Figure 7.7: The confusion matrix of malware classification for multiple malware families.

This experiment suggests that Modality Vectors also have a potential applicability to assist in the classification of malware family labels.

7.3.3.3 Behavior Characterization

As described in Section 7.2.4, to characterize malicious apps' behaviors, we first construct a behavior matrix based on malicious behaviors observed within an existing training set of known malware applications. To decrease sampling bias, we produce our *training* dataset using malware samples from 29 different malware families, each contributing a minimum of 5 members. Next, for each selected family, we manually extract a malicious behavior description for this family using documentation describing the malware family from sites that contain malware analysis reports, such as threat reports from various AV companies (e.g., Symantec.com). There are many detailed public sources of information regarding malicious behavior description for many existing Android malware families [103]. For this experiment, we focus on the following six malicious behaviors commonly observed within many malware families: stealing phone information (GetPho), Sending SMS (SdSMS), blocking SMS (BkSMS), communicating with a C&C (C&C), escalating root privilege (Root) and accessing geographical information (GetGeo). Table 7.6 summarizes malicious behaviors observed within those 29 malware families.

Using an Association Rule Mining system, DroidMiner automatically learned 439 behavior association rules. In Table 7.7, we summarize the number of association rules mined for each malicious behavior. Applying these learned rules to test new malware samples (not in the training set) with ground truth information, we find that DroidMiner could generate correct behavior characterizations.

7.3.3.4 Efficiency

We now consider the performance overhead of DroidMiner in identifying modalities. As described in Section 7.2.3, modality identification involves three steps: 1) decompilation, 2) behavior graph generation and 3) modality vector generation. Ta-

Family	GetPho	SdSMS	BkSMS	C&C	Root	GetGeo
ADRD		√	√	√		√
AnserverBot		√		√		
Asroot					√	
BaseBridge		√		√	√	
BeanBot	√	√	√	√		
Bgserv	√	√	√	√		
DroidDream				√	√	
DDLight				√		
DroidKungFu1	√			√	√	
DroidKungFu2	√			√	√	
DroidKungFu3	√			√	√	
DroidKungFu4				√		
DroidKungFu5	√			√	√	
FakePlayer		√				
Geinimi	√	√		√		
GingerMaster				√	√	
GoldDream	√					
Gone60	√	√				
GPSSMSSpy		√				
jSMSHider	√	√	√			
KMin	√	√				
Pjapps	√	√				
Plankton				√		
RogueSPPush		√				
SmsSend		√				
SndApps	√					√
YZHC	√	√	√			
zHash					√	
Zsone		√	√			

Table 7.6: Malicious behaviors in different families.

ble 7.8 shows the mean and median value of time spent on each step and the overall time required to identify modalities for all collected apps.

Table 7.8 illustrates that DroidMiner expended an average of 19.8 seconds and a median of 5.4 seconds to identify modalities in an app. We further find that the vast

GetPho	SdSMS	BkSMS	C&C	Root	GetGeo
157	144	11	71	37	19

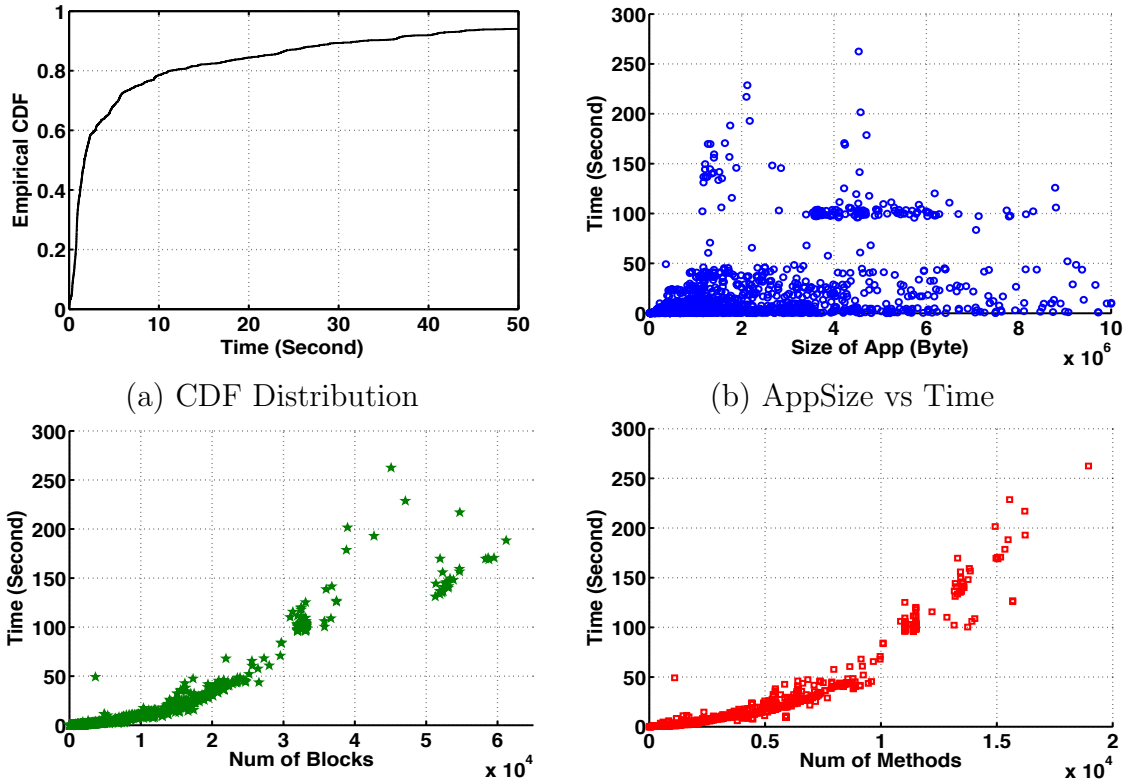
Table 7.7: Number of association rules mined for common malicious behaviors.

majority of this time is spent on behavior graph generation.

Step	Decompile	Behavior Graph	Modality Vector	Overall
Mean	3.87	15.19	1.10	19.83
Median	1.65	3.08	0.56	5.35

Table 7.8: Processing time for identifying modalities.

For a more fine-grained performance analysis of this step, Figure 7.8(a) shows the cumulative distribution of time used to generate behavior graphs for our collected apps. For approximately 80% of the apps, our system generates their behavior graphs within 10 seconds. As seen in Figure 7.8(c) and 7.8(d), the values of time spent generating behavior graphs typically rise with the increased number of control-flow blocks and programmer-defined methods found in the app. This occurs because the behavior graphs of apps are extracted through analyzing the control-flow logic of API functions with the consideration of their located control-flow blocks and programmer-defined methods. Thus, the numbers of control-flow blocks and programmer-defined methods will affect the time used to generate the graphs. However, as shown in Figure 7.8(b), the time spent in generating behavior graphs does not increase due to increase in the app size. That is, a bigger app size does not necessarily contain more complex control-flow logic.



(a) CDF Distribution

(b) AppSize vs Time

(c) Block Number vs Time

(d) Method Number vs Time

Figure 7.8: Processing time for generating behavior graphs.

7.4 Discussion and Limitation

7.4.1 DroidMiner Against Zero-day Attacks

Emerging malware generally falls into two classes: fundamentally new strain with entirely novel code bases, and malware that improves (evolves) from an existing code base. The latter form arguably represents the dominant case. We believe DroidMiner is well designed to adapt to evolutionary change in existing code bases, and thus useful in detecting most emerging variant strains. As long as new malware launches malicious behaviors through utilizing modalities observed in known malware families, DroidMiner should detect it. For entirely novel malware strains, an

additional strength of DroidMinder is that unlike traditional systems that require human expertise, DroidMiner's features (modalities) can be automatically learned and updated by feeding new malware samples.

7.4.2 DroidMiner Against Common Evasion Techniques

As there is an arms race between attackers and defenders, Android malware may evolve to be more evasive. As observed by DroidChameleon [91], common malware transformation techniques (e.g., repackaging, changing field names, and changing control-flow logics) could evade many existing commercial anti-malware tools. However, DroidMiner is resilient to these common malware evasion techniques studied in [91]. Specifically, DroidMiner does not rely on specific signing signatures or class/method/field names to detect malware. The simple program transformation (resigning, repackaging, changing names) will not affect the detection model used in DroidMiner. Another type of evasion technique is to insert noisy code or to change specific control-flow logic. However, DroidMiner is designed to extract all subsequences of suspicious control-flow logic commonly seen in malware. As long as the malware follows a known programming paradigm to achieve malicious goals (e.g., intercepting short text messages after receiving them, and obtaining the phone number before sending it), DroidMiner could still capture such suspicious logic and ignore noise API injections.

7.4.3 Limitations

Like any learning-based approach, DroidMiner requires an accurate training dataset to mine its malicious behaviors into modalities. The effectiveness of our approach depends on the quality of the given training data, e.g., labeled malicious Android apps and their families. Fortunately, it was easy for us to obtain such data (thanks to prior research efforts from academia and industry). In fact, one may also rec-

ognize DroidMiner's automatic learning approach as a feature rather than a strict liability. Whereas most existing approaches require significant manual labor to generate signature, specifications, and models for detection, DroidMiner offers far more automated model generation.

Our current behavior graphs and modalities primarily model the control flow information corresponding to malware behavior, i.e., we may miss some important data flow information that could help build better behavior models. Also, the obfuscation of the control-flow logic and the constant-string for content providers in the malicious apps' bytecode may decrease the detection rate of our approach. Attackers could also split the constant-string for content providers (e.g., "content://sms/inbox/") into segments and recombine them later to avoid the identification of the usage of sensitive content providers.

DroidMiner currently employs static analysis, which is a reasonable choice given that current Android apps are relatively easy to reverse engineer statically, unlike notorious malware programs commonly seen in PC-based malware. We acknowledge that dynamic analysis provides an advantage in accurately studying runtime behaviors, and in the future we plan to extend DroidMiner to utilize a combination of static and dynamic analyses. Like other Java static analysis studies, DroidMiner may fail to identify certain usages of instances/methods, which are encrypted or made by using Java Reflection and native code. This serves as another motivation for us to incorporate dynamic analysis in our future work.

7.5 Summary

Android malware detection is a relatively new and very challenging research area. In this chapter, we introduced a new Android malware detection system, named DroidMiner. Our detection approach is designed based on the intuition that An-

droid malware authors must obey certain specific rules, pre-defined by the Android platform, to realize malware functionality (e.g., using particular Android framework APIs and accessing particular content providers). DroidMiner automatically mines malicious parasitic code segments from a corpus of malicious mobile apps to detect Android malware, while preserving the control flow logic. We reported an experimental evaluation of DroidMiner on many real-world Android apps and showed that it has very promising detection accuracy with a very low false positive rate.

8. LESSONS LEARNED AND A FUTURE MALICIOUS ACTIVITY DETECTION SYSTEM

8.1 Lessons Learned

As we have highlighted earlier regarding the lack of the understanding of the new types of the malicious activities in the OSN and smartphone platforms, an in-depth analysis of the way in which the malicious activities are launched and propagated is indeed needed. From and the success of our findings, and proposed defensive insights, we have learned the following important lessons:

- *An in-depth understanding of the malicious activities can facilitate the generation of the effective defensive insights.* By understanding the differences between malicious activities and benign activities, we can design effective detection features to distinguish malicious activities from benign ones. By understanding the strategies and steps used by cyber-criminals to launch malicious activities, we can both design effective defensive rules to catch those launching actions, and reverse engineer those strategies to find other malicious activities. By understanding how the malicious activities are propagated, wen can design approaches to find more other malicious activities from a small seed set of identified malicious activities.

- *The graph propagation-based algorithms can be effectively used to sample malicious activities in both OSN and smartphone platforms by starting from a small seed set of known malicious activities.* In both the OSN and smartphone platforms, if the cyber-criminals want to achieve significant attacking affect (or sufficient profit), they typically require to launch multiple malicious ac-

tivities. Meanwhile, these malicious activities launched by the same group of cyber-criminals typically inevitably have some intrinsic connections/relationships/similarities. Accordingly, these malicious activities and the connections among them can be modeled into a relationship graph. Then, once we use a few identified malicious activities as seeds, and propagate a score from them to the rest of activities in the graph, those activities accumulate sufficient scores (i.e., have strong connections with existing known malicious activities) are more likely to be malicious. Based on this observation, we can design lightweight and effective inference algorithms to find unknown malicious activities from known ones.

- *A more complete and effective Android malware detection solution should rely on more fine-grained detection features that represent the programming procedure.* Although several existing Android malware detection systems are developed, their detection performance is highly limited due to the features used in those detection systems. These features are selected either too corse-grained that are not effective enough to distinguish malicious apps from benign ones. Or, they can not be used to represent well the programming procedure of the Android apps. Thus, such detection systems tend to generate many false negatives. As we have shown, compared with the corse-grained features, those fine-grained detection features that represent the programming procedure, can be used to effectively distinguish malicious apps from benign ones (i.e., achieve a high detection rate and a low false positive rate).

8.2 A Future Malicious Activity Detection System

Figure 8.1 shows the architecture of an example design of a future malicious activity detection system that incorporates the complementary techniques we have

discussed.

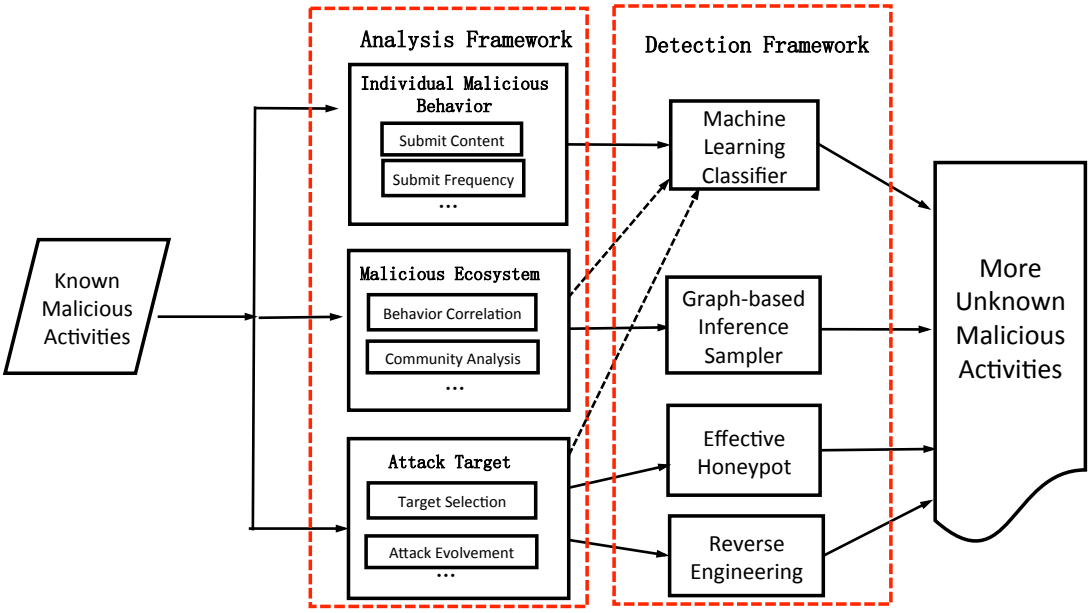


Figure 8.1: Example combination of multiple techniques in a future malicious activity detection system on OSN and smartphone platforms.

The new system is divided into two main parts: analysis components and detection components. The analysis components contains three major categories: analysis of individual malicious behavior, analysis of malicious ecosystem, and analysis of attack targets. Each of the four detection components (Machine learning classifier, inference sampler, honeypot and reverse engineering strategy) are motivated by the defensive insights obtained from the analysis results.

While obtaining a seed set of known malicious activities on OSN or smartphone platform, the analysis of the differences between the malicious behaviors and benign behaviors in each individual identify (OSN account, smartphone market account, and smartphone malware) are very useful (and typically the first step) to design detection approaches. The malicious behaviors in each individual identify can include the

submission of the malicious content (OSN spam messages and malicious smartphone apps), the submission frequency, and etc. Once accumulating sufficient amount and types of known malicious activities, we can design effective detection features by comparing the differences between malicious and benign behaviors, and further build effective machine learning classifiers to detect malicious activities.

Besides focusing on analyzing isolated behaviors, the analysis of the malicious ecosystem can help better understand more deep insights on how attackers launch and spread the malicious activities on the communication platforms. In order to attract more victims, attackers typically need to launch multiple malicious activities (OSN spam and smartphone malware). Also, due to practical restrictions, attackers typically also have to repeatedly use the same malicious accounts to submit their malicious content, or repeatedly use similar strategies (e.g., injecting malicious URLs into OSN messages, and using sensitive Android framework APIs to implement Android malware) to launch and spread malicious activities. Thus, there are obvious behavior correlation relationships among malicious identities (e.g., similar malicious content patterns, similar account behaviors, and similar temporary patterns). These correlations can be further used to find malicious communities, which are very important to understand the influence and the categories of the malicious activities. Also, the analysis of the ecosystem can further motivate the design of community-based detection features, to build more effective machine learning classifier. In addition, by building the relation graph among the malicious activities, we can design effective graph-based inference samplers to find more other unknown malicious activities. Since this approach is essentially a prioritized sampler, it is very suitable for the large-scale of dataset, which typically requires great amount of resource and time. Also, unlike the machine learning classifier, which requires sufficient amount of training data, this type of approach can be effectively applied to find unknown malicious

activities, while starting from a very small seed set of known malicious ones.

By analyzing the attack targets, we can further understand how attackers choose their targets, and even uncover the evolvement of the attacks by keeping monitoring the communication between attackers and their victims. Once knowing the strategies used by attackers to choose their attack targets (specific types of OSN/smartphone market accounts, or versions of smartphones), we can design more effective honeypots to attract attackers, and effective detection approaches by reverse engineering those strategies. In addition, since the malicious activities can become more evasive, this type of analysis is very important to understand the evolvement of the malicious behaviors, and further help design more robust detection features.

To conclude, we can develop a relatively comprehensive and practical solution by combining multiple complementary detection techniques to achieve a multi-perspective view, as shown in this section.

9. CONCLUSION AND FUTURE WORK

9.1 Conclusion

Malicious OSN accounts and malicious Android apps are considered as the most dangerous malicious activities to the security on OSN and smartphone platforms. Millions of benign users have suffered a lot from those new types of malicious activities, and they can further be utilized by cyber-criminals to spread attacks and fraudulent actives on these two types of communication platforms. Thus, we urgently need to better understand the ecosystem of malicious OSN accounts and malicious Android apps, and further to design effective solutions to mitigate and defend against them.

In this dissertation, we have proposed an in-depth analysis of the ecosystem of malicious activities in the two emerging communication platforms, and further presented effective defensive insights against those malicious activities. Our analysis mainly focuses on three facets of the ecosystem (attack infrastructure, attack target, and relationships among attack identities). We also presented three inference algorithms for sampling more likely malicious activities (two are made for the OSN platform, and one is made for the smartphone platform), and detection system (DroidMiner). We have discussed our analysis and defensive insights in details, and summarized the lessons we have learned.

Our analysis of the malicious activities and defensive insights have the following three major characteristics:

First, our analysis covers multiple perspectives of the ecosystem of the malicious activities. Our analysis of the spammers' social networks uses the knowledge of graph theory to understand the social relationships among social spammers, and further

to reveal the reason why spammers have mixed well in the current real-world OSN platforms. By deploying social honeypots with multiple fine-grained social behaviors, our analysis of the spammers' spamming targets reveals that many spammers tend to build unsolicited social relationships with those accounts that expose specific interests. Based on these findings, we provide guidelines for deploying more effective social honeypots. Our analysis of the ecosystem of the Android malware first reveals the fact that a few indications that are commonly used to select Android apps with high quality are not that trustable. Then, the analysis further shows the characteristics of the networking infrastructure that are tend to be utilized by Android malware authors. It further reveals the characteristics of the community relationships among multiple Android apps that share the same developer or similar remote communication servers. All the analysis of these perspectives facilitate us to better understand these new types of malicious activities. Such analysis further spurs new defensive insights against those malicious activities.

Second, our solutions are practical for the large-scale datasets. One common challenge of detecting malicious activities in these two types of platforms is the huge volume of the objects that are uploaded in the platforms. Also, these objects are constantly updated by their providers. Thus, given the limited resource/time, it is very challenging or even impossible to make in-depth checks on every object whether it is malicious in a short time period. Thus, lightweight defensive algorithms, to guide to more suspicious objects instead of scanning or analyzing all objects at the same time, are indeed needed. By exploring social relationships and semantic coordinations among spam OSN accounts, and reverse engineering spammers' strategies of selecting spam targets, we provide light-weight inference algorithms to sample more likely spam OSN accounts. Similarly, by exploring the community relationships among Android malware, we present light-weight inference algorithms to sample more likely

malicious Android apps.

Finally, our proposed defensive sights are practical and capable to work in the real world. Our defensive insights are evaluated on real-world dataset (real-world Twitter accounts and Android apps). Experimental results are promising, showing that our defensive insights can be used to effectively capture those malicious activities on real-world communication platforms.

9.2 Future Work

In the future, we plan to study the following directions:

- Larger dataset and more types of data. We plan to design and test more crawling strategies and crawl more data, to decrease possible sampling bias. We also plan to obtain more data from other OSN and smartphone platforms (e.g., Facebook accounts and iOS malware), to reveal more insights of the malicious activities.

- Dynamic view of the malicious activities. Besides analyzing the static view of the malicious activities by crawling the dataset at a particular timestamp, we plan to further analyze a more dynamic (evolvement) of the malicious activities by monitoring the change of identified malicious activities for a longer time. By doing this, we can further understand how the malicious activities evolved, and further design more effective detection approaches. We also plan to adaptively change the strategies of our honeypots to more effectively collect spammers.

- Analysis of more prospectives. We plan to provide more analysis of the similarities and differences of the malicious activities among more communication platforms. We also plan to make a more in-depth analysis of the strategies utilized by cyber-criminals to gain their profits.

- More comprehensive detection approaches. We plan to cooperate our inference algorithms with other detection features to build more comprehensive detection models. Then, we can further evaluate the advantages of our proposed defensive approaches, compared with other existing ones.

- Improvements of the DroidMiner. We plan to explore how to model malicious code segments written in native code, which DroidMiner currently does not handle well. We also plan to combine the dynamic analysis approaches with DroidMiner to further improve the detection performance.

REFERENCES

[1] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-level Features for Robust Malware Detection in Android. In *Proceedings of the 9th International Conference on Security and Privacy in Communication Networks (SecureComm)*, Sydney, Australia, 2013.

[2] Alexa. Alexa Top Websites, April 2014. `http://www.alexa.com/topsites/category/Top/Computers/Internet/Domain_Names`.

[3] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011.

[4] S. Antonatos, I. Polakis, T. Petsas, and E. Markatos. A Systematic Characterization of IM Threats Using Honeypots. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2010.

[5] AppDH. App DH Android Market, May 2014. `http://www.appdh.com/`.

[6] C. Arthur. Twitter Phishing Hack Hits BBC, Guardian and Cabinet Minister, Feburary 2010. `http://www.guardian.co.uk/technology/2010/feb/26/twitter-hack-spread-phishing`.

[7] K. Au, Y. Zhou, Z. Huang, D. Lie, X. Gong, X. Han, and W. Zhou. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, USA, 2012.

[8] BarracudaLabs. Barracuda Labs 2010 Midyear Security Report, October 2010. `http://www.barracudalabs.com/downloads/`

BarracudaLabs2010MidyearSecurityReport.pdf.

[9] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. Automatically Securing Permission-Based Software by Reducing the Attack Surface: An Application to Android. In *Proceedings of the 27th International Conference On Automated Software Engineering (ASE)*, Essen, Germany, 2012.

[10] F. Benevenuto, G. Magno, T. Rodrigues, and V. Almeida. Detecting Spammers on Twitter. In *Proceedings of Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS)*, Redmond, Washington, 2010.

[11] F. Benevenuto, T. Rodrigues, V. Almeida, J. Almeida, C. Zhang, and K. Ross. Identifying Video Spammers in Online Social Networks. In *Proceedings of International Workshop on Adversarial Information Retrieval on the Web (AirWeb)*, Beijing, China, 2008.

[12] F. Benevenuto, T. Rodrigues, V. Almeida, J. Almeida, C. Zhang, and K. Ross. Detecting Spammers and Content Promoters in Online Video Social Networks. In *Proceedings of ACM SIGIR Conference*, Boston, Massachusetts, USA 2009.

[13] A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral Detection of Malware on Mobile Handsets. In *Proceeding of the 6th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, Breckenridge, CO, USA, 2008.

[14] Y. Boshmaf, I. Muslukhov, K. Beznosov, and M. Ripeanu. The Socialbot Network: When Bots Socialize for Fame and Money. In *Proceedings of 2011 Annual Computer Security Applications Conference (ACSAC)*, Orlando, FL, USA, 2011.

[15] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards Taming Privilege-escalation Attacks on Android. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2012.

[16] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-Based Malware Detection System for Android. In *Proceedings of the 1st Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, Chicago, IL, 2011.

[17] Carol. New Koobface Campaign Spreading on Facebook, January 2011. `http://forums.cnet.com/7726-6132_102-5064273.html`.

[18] C. Castillo, M. Mendoza, and B. Poblete. Information Credibility on Twitter. In *Proceedings of International World Wide Web Conference (WWW)*, Hyderabad, India, 2011.

[19] M. Cha, H. Haddadi, F. Benevenuto, and K. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *Proceedings of International AAAI Conference on Weblogs and Social Media (ICWSM)*, Washington DC, USA, 2010.

[20] P. P. Chan, L. C. Hui, and S. M. Yiu. Droidchecker: Analyzing Android Applications for Capability Leak. In *Proceedings of the 5th ACM conference on Security and Privacy in Wireless and Mobile Networks*, Tucson, Arizona, USA, 2012.

[21] K. Chen, N. Johnson, V. Silva, S. Dai, K. MacNamara, T. Magrino, E. Wu, M. Rinard, and Dawn Song. Contextual Policy Enforcement in Android Applications with Permission Event Graphs. In *Proceedings of the 20thNetwork and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2013.

[22] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, Washington DC, USA, 2011.

[23] A. Chowdhury. State of Twitter Spam, March 2010. `http://blog.twitter.`

com/2010/03/state-of-twitter-spam.html.

[24] N. Christin, S. Yanagihara, and K. Kamataki. Dissecting One Click Frauds. In *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS)*, Chicago, IL, USA, 2010.

[25] M. Christodorescu, S. Jha, and C. Kruegel. Mining Specifications of Malicious Behavior. In *Proceedings of the 6th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Dubrovnik, Croatia, 2007.

[26] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-Aware Malware Detection. In *Proceedings of the 26th IEEE Security and Privacy*, Oakland, CA, USA, 2005.

[27] Z. Chu, S. Gianvecchio, H. Wang, and S. Jajodia. Who is Tweeting on Twitter: Human, Bot, or Cyborg? In *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, Austin, Texas, USA, 2010.

[28] G. Cluley. Twitter onMouseOver Security Flaw Widely Exploited, September 2010. http://nakedsecurity.sophos.com/2010/09/21/twitter-onmouseover-security-flaw-widely-exploited/.

[29] M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-related Policy Enforcement for Android. In *Proceedings of the 13th Information Security Conference (ISC)*, Boca Raton, FL, USA, 2010.

[30] Cyveillance. Malware Detection Rates for Leading AV Solutions, 2013. https://www.cyveillance.com/web/docs/WP_MalwareDetectionRates.pdf.

[31] G. Danezis and P. Mittal. SybilInfer: Detecting Sybil Nodes using Social Networks. In *Proceedings of the16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2009.

[32] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire:

Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the 20th USENIX Security Symposium (USENIX)*, San Francisco, CA, USA, 2011.

[33] DNS-BH. Malware Domain Blocklist, May 2014. `http://www.malwaredomains.com/`.

[34] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. Revirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, USA, 2002.

[35] W. Enck, P. Gilbert, B.G. Chun, L. P. Cox, J. Jung, P. Mc-Daniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, BC, Canada, 2010.

[36] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium (USENIX)*, Portland, OR, USA, 2011.

[37] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, USA, 2009.

[38] J. Erman, A. Gerber, K. K. Ramadrishnan, S. Sen, and O. Spatscheck. Over the Top Video: The Gorilla in Cellular Networks. In *Proceedings of the 11th ACM SIGCOMM conference on Internet measurement (IMC)*, Berlin, Germany, 2011.

[39] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A First Look at Traffic on Smartphones. In *Proceedings of the 10th ACM SIGCOMM*

conference on Internet Measurement (IMC), Melbourne, Australia, 2010.

[40] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystied. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, USA, 2011.

[41] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing Near-optimal Malware Specifications from Suspicious Behaviors. In *Proceedings 31th of IEEE Security and Privacy*, Oakland, CA, USA, 2010.

[42] W. Galuba, K. Aberer, D. Chakraborty, Z. Despotovic, and W. Kellerer. Outtweeting the Twitterers - Predicting Information Cascades in Microblogs. In *Proceedings of USENIX Workshop on Online Social Networks (WOSN)*, Boston, MA, USA, 2010.

[43] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, and B. Zhao. Detecting and Characterizing Social Spam Campaigns. In *Proceedings of ACM SIGCOMM conference on Internet measurement (IMC)*, Melbourne, Australia, 2010.

[44] A. Gember, A. Anand, and A. Akella. A Comparative Study of Handheld and Non-handheld Traffic in Campus Wi-Fi Net-works. In *Proceedings of the 12th International conference on Passive and Active Measurement*, Atlanta, GA, USA, 2011.

[45] Google. Androguard, May 2014. `http://code.google.com/p/androguard/`.

[46] Google. Android TCPDump, April 2014. `http://www.kandroid.org/online-pdk/guide/tcpdump.html`.

[47] Google. Dex2Jar, May 2014. `https://code.google.com/p/dex2jar/`.

[48] Google. Google Play, May 2014. `https://play.google.com/store?hl=en`.

[49] Google. Google Safe Browsing API, April 2014. `http://code.google.com/apis/safebrowsing/`.

[50] Google. Smali, May 2014. `https://code.google.com/p/smali/`.

[51] C. Grier, K. Thomas, V. Paxsony, and M. Zhang. @spam: The Underground on 140 Characters or Less. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, USA, 2010.

[52] Honeynet. Capture HPC, April 2014. `https://projects.honeynet.org/capture-hpc`.

[53] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Arent the Droids Youre Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, USA, 2011.

[54] Instagr. Instagr – Photo Sharing for Your iPhone, April 2014. `http://instagr.am/`.

[55] L. Invernizzi, P. Comparetti, S. Benvenuti, C. Kruegel, M. Cova, and G. Vigna. EVILSEED: A Guided Approach to Finding Malicious Web Pages. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, USA, 2012.

[56] D. Ionescu. Twitter Warns of New Phishing Scam, October 2009. `http://www.pcworld.com/article/174660/twitter_phishing_scam.html`.

[57] D. Irani, M. Balduzzi, D. Balzarotti, E. Kirda, and C. Pu. Reverse Social Engineering Attacks in Online Social Networks. In *Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, Amsterdam, Netherlands, 2011.

[58] J. John, F. Yu, Y. Xie, A. Krishnamurthy, and M. Abadi. Heat-seeking Honeypots: Design and Experience. In *Proceedings of the 20th International World Wide Web Conference (WWW)*, Hyderabad, India, 2011.

[59] Jose. Twitter-based Botnet Command Channel, August 2009. `http://ddos.arbornetworks.com/2009/08/twitter-based-botnet-command-channel/`.

[60] C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and Efficient Malware Detection at the End Host. In *Proceedings of USENIX Security Symposium (USENIX)*, San Diego, CA, USA, 2009.

[61] G. Koutrika, F. Effendi, Z. Gyongyi, P. Heymann, and H. Garcia-Molina. Combating Spam in Tagging Systems. In *Proceedings of International Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*, Alberta, Canada, 2007.

[62] C. Kreibich and J. Crowcroft. Honeycomb: Creating Intrusion Detection Signatures using Honeypots. In *Proceedings of the 2nd Workshop on Hot Topics in Networks (HotNets)*, Cambridge, MA, USA, 2003.

[63] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a Social Network or a News Media? In *Proceedings of the International World Wide Web Conference (WWW)*, Raleigh, NC, USA, 2010.

[64] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4Android: A Generic Operating System Framework for Secure Smartphones. In *Proceedings of the 1st Workshop on Security and Privacy in Smartphones and Mobile Devices*, Chicago, IL, USA, 2011.

[65] K. Lee, J. Caverlee, and S. Webb. Uncovering Social Spammers: Social Honeypots + Machine Learning. In *Proceedings of ACM SIGIR Conference*, Geneva, Switzerland, 2010.

[66] K. Lee, B. Eoff, and J. Caverlee. Seven Months with the Devils: A Long-Term Study of Content Polluters on Twitter. In *Proceedings of 5th International AAAI Conference on Weblogs and Social Media (ICWSM)*, Dublin, Ireland, 2012.

[67] N. Leontiadis, T. Moore, and N. Christin. Measuring and Analyzing Search-

Redirection Attacks in the Illicit Online Prescription Drug Trade. In *Proceedings of the 20th USENIX Security Symposium (USENIX)*, San Francisco, CA, USA, 2011.

[68] C. Lever, M. Antonakakis, and B. Reaves. The Core of the Matter: Analyzing Malicious Traffic in Cellular Carriers. In *Proceedings of the 20th Network & Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2013.

[69] S. Levine. So Much Twitter Spam, April 2011. `http://blog.sysomos.com/2011/04/07/so-much-twitter-spam/`.

[70] LiTianWuXian. Anzhi Android Market, May 2014. `http://www.anzhi.com/`.

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

[72] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerablilities. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, USA, 2012.

[73] MalwareDomainList. Malware Domain List, May 2014. `http://www.malwaredomainlist.com/`.

[74] A. Matwyshyn, A. Keromytis, A. Cui, and S. Stolfo. Ethics in security vulnerability research. *Journal of IEEE Security and Privacy*, 8(2):67–72, 2010.

[75] P. Metaxas and E. Mustafaraj. Prominence in Minutes: Political Speech and Real-time Search. In *Proceedings of the Web Science (WebSci)*, Raleigh, NC, USA, 2010.

[76] A. Mislove, M. Marcon, K. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet measurement (IMC)*, 2007.

[77] A. Moshchuk, T. Bragin, S. Gribble, and H. Levy. A crawler-based study of

spyware on the web. In *Proceedings of the 13th Annual Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, USA, 2006.

[78] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *Proceedings of the 5th International Conference on Cyber Security (ICCS)*, Amsterdam, The Netherlands, 2010.

[79] M. Ongtang, K. Butler, and P. McDaniel. Porscha: Policy Oriented Secure Content Handling in Android. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, Austin, TX, USA, 2010.

[80] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, Honolulu, Hawaii, USA, 2009.

[81] A. Ostrow. Twitter Spam Invades Trending Topics, May 2009. `http://mashable.com/2009/05/11/twitter-spam-trending-topics/`.

[82] B. Pan. 17 Bad Mobile Apps Still Up, 700,000+ Downloads So Far, May 2012. `http://blog.trendmicro.com/trendlabs-security-intelligence/17-bad-mobile-apps-still-up-700000-downloads-so-far/`.

[83] D. Pelleg and A. Moore. X-Means: Extending K-means with Efficient Estimation. In *Procceedings of International Conference on Machine Learning (ICML)*, Stanford, CA, USA, 2000.

[84] H. Peng, C. Gates, B. Sarm, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using Probabilistic Generative Models for Ranking Risks of Android Apps. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, USA, 2012.

[85] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android:

Versatile Protection for Smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, Austin, Texas, USA, 2010.

[86] R. Pozo, K. Remington, and A. Lumsdaine. Sparselib++, October 2008. `http://math.nist.gov/sparselib++/`.

[87] N. Provos. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium (USENIX)*, Boston, MA, USA, 2004.

[88] Purchasetwitterfriends. Purchase Twitter Friends, May 2014. `http://www.purchasetwitterfriends.com/`.

[89] Quora. Why Do I Get Spam Followers on Twitter?, September 2010. `http://www.quora.com/Why-do-I-get-so-many-spam-followers-on-Twitter`.

[90] A. Ramachandran and N. Feamster. Understanding the Network-Level Behavior of Spammers. In *Procceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, Pisa, Italy, 2006.

[91] V. Rastogi, Y. Chen, and X. Jiang. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In *Proceedings of the 8th International Conference on Cyber Security (ICCS)*, Hangzhou, China, 2013.

[92] J. Ratkiewicz, M. Conover, M. Meiss, B. Goncalves, S. Patil, A. Flammini, and F. Menczer. Detecting and Tracking the Spread of Astroturf Memes in Microblog Streams. In *Proceedings of the 5th International Conference on Weblogs and Social Media (ICWSM)*, Barcelona, Spain, 2011.

[93] A. Ray. Seven Things I learned to Bait Twitter Spammers, July 2011. `http://www.experiencetheblog.com/2011_07_01_archive.html`.

[94] A. Schmidt, R. Bye, H. Schmidt, J. Clausen, O. Kiraz, K. Yxksel, S. Camtepe, and A. Sahin. Static Analysis of Executables for Collaborative Malware Detection on Android. In *Procceddings of Communication and Information Systems Security Symposium*, Dresden, Germany, 2009.

[95] A. Schmidt, H. Schmidt, J. Clausen, K. Yuksel, O. Kiraz, A. Sahin, and S. Camtepe. Enhancing Security of Linux-based Android Devices. In *Proceedings of 15th International Linux Kongress*, Hamburg, Germany, 2008.

[96] Seantm. How to Get Rid of DM Spam on Twitter, August 2010. `http://seanmalarkey.com/rid-dm-inbox-spam-auto-dms-mafia-invites-twitter`.

[97] A. Shabtai, Y. Fledel, and Y. Elovici. Securing Android-Powered Mobile Devices Using SELinux. In *Proceedings of the 31th IEEE Security and Privacy*, Oakland, CA, USA, 2010.

[98] Sina. Money-stealing Apps are Hosting in the Mobile Devices, April 2012. `http://finance.sina.com.cn/money/lczx/20120410/070311783396.shtml`.

[99] Sina. Android Markets are the Source for Android Malware, October 2013. `http://tech.sina.com.cn/t/2013-10-07/02398791182.shtml`.

[100] SlideMe. SlideMe Android Market, May 2014. `http://slideme.org/`.

[101] V. Sridharan, V. Shankar, and M. Gupta. Twitter Games: How Successful Spammers Pick Targets. In *Proceedings of 28th Annual Computer Security Applications Conference (ACSAC)*, Orlando, Florida, USA, 2012.

[102] G. Stringhini, S. Barbara, C. Kruegel, and G. Vigna. Detecting Spammers On Social Networks. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Austin, Texas, 2010.

[103] Symantec. Symantec Security Response, May 2014. `http://www.symantec.com/security_response/landing/azlisting.jsp`.

[104] Tapp. Tapp Android market, May 2014. `http://tapp.ru/`.

[105] K. Thomas, C. Grier, V. Paxson, and D. Song. Suspended Accounts in Retrospect:An Analysis of Twitter Spam. In *Proceedings of Internet Measurement*

Conference (IMC), Berlin, Germany, 2011.

[106] Twitpic. Twitpic, April 2014. http://twitpic.com/.

[107] Twitter. Twitter Streaming API, September 2012. https://dev.twitter.com/docs/streaming-apis.

[108] Twitter. The Twitter Rules, May 2014. http://help.twitter.com/entries/18311-the-twitter-rules.

[109] Twitter. Trending Topics, May 2014. http://support.twitter.com/entries/101125-about-trending-topics.

[110] Twitter. Twiends, May 2014. http://twiends.com/.

[111] Twitter. Twitter Public Timeline, April 2014. http://twitter.com/public_timeline.

[112] Twitter. Twitter Search, May 2014. https://twitter.com/#!/search-home.

[113] Twitter. Twitter's Following Limits, May 2014. http://support.twitter.com/groups/32-something-s-not-working/topics/117-following-problems/articles/66885-i-can-t-follow-people-follow-limits.

[114] VirusTotal. VirusTotal Blacklist Service, May 2014. https://www.virustotal.com/.

[115] L. Walker. Automatically Follow Back, May 2014. http://personalweb.about.com/od/howtotwitter/a/What-Is-Auto-Follow-And-How-Does-It-Work.htm.

[116] A. Wang. Don't follow me: spam detecting in Twitter. In *Proceedings of Conferene on Security and Cryptography (SECRYPT)*, Athens, Greece, 2010.

[117] T. Wang, Y. Chen, Z. Zhang, P. Sun, B. Deng, and X. Li. Unbiased Sampling in Directed Social Graph. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, New Delhi, India, 2010.

[118] Yi. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated Web Patrol with Strider HoneyMonkeys. In *Proceedings of the 13th Annual Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, USA, 2006.

[119] M. Warman. Fake Android Apps Scam Costs, May 2012. http://www.telegraph.co.uk/technology/news/9286538/Fake-Android-apps-scam-costs-28000.html.

[120] C. Warren. Warning: Facebook Clickjacking Attack Spreading Through Likes, May 2010. http://mashable.com/2010/05/31/facebook-like-worm-clickjack/.

[121] C. Warren. Lady Gaga Falls Prey to Rogue Twitter Attack, April 2011. http://mashable.com/2011/04/28/lady-gaga-twitter-attack/.

[122] WhatIsMyIPAddress. WhatISMyIPAddress Blacklist, May 2014. http://whatismyipaddress.com/blacklist-check.

[123] C. Whittaker, B. Ryner, and M. Nazif. Large-Scale Automatic Classification of Phishing Pages. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2010.

[124] C. Whittaker, B. Ryner, and M. Nazif. Large-Scale Automatic Classification of Phishing Pages. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2010.

[125] R. Whitwam. Circumventing Google Bouncer, June 2012. http://www.extremetech.com/computing/130424-circumventing-googles-bouncer-androids-anti-malware-system.

[126] Wikipedia. Gaussian Error Function, May 2014. http://en.wikipedia.org/wiki/Error_function.

[127] Wikipedia. Latent Dirichlet Allocation, May 2014. http://en.wikipedia.

org/wiki/Latent_Dirichlet_allocation.

[128] Wikipedia. PageRank Algorithm, May 2014. http://en.wikipedia.org/wiki/PageRank.

[129] D. Wu, C. Mao, T. Wei, H. Lee, and K. Wu. DroidMat: Android Malware Detection through Manifest and API Calls Tracing. In *Proceedings of the 7th Asia JCIS*, Tokyo, Japan, 2012.

[130] R. Xu, H. Saidi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21st USENIX Security Symposium (USENIX)*, Bellevue, WA, USA, 2012.

[131] L. Yan and H. Yin. Droidscope: Seamlessly Reconstructing the Os and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Security Symposium (USENIX)*, Bellevue, WA, USA, 2012.

[132] C. Yang, R. Harkreader, and G. Gu. Die Free or Live Hard? Empirical Evaluation and New Design for Fighting Evolving Twitter Spammers. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Menlo Park, CA, USA, 2011.

[133] C. Yang, R. Harkreader, J. Zhang, S. Shin, and G. Gu. Analyzing Spammers' Social Networks For Fun and Profit – A Case Study of Cyber Criminal Ecosystem on Twitter. In *Proceedings of the 21st International World Wide Web Conference (WWW)*, Lyon, France, 2012.

[134] S. Yardi, D. Romero, G. Schoenebeck, and D. Boyd. Detecting Spam in a Twitter Network. *Journal of First Monday*, 15(1), 2010.

[135] V. Yegneswaran, J. Giffin, P. Barford, and S. Jha. An Architecture for Generating Semantics-Aware Signatures. In *Proceedings of the 14th USENIX Security Symposium (USENIX)*, Anaheim, CA, USA, 2005.

[136] H. Yu, M. Kaminsky, P. Gibbons, and A. Flaxman. SybilGuard: Defending

Against Sybil Attacks via Social Networks. In *Procceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, Pisa, Italy, 2006.

[137] J. Zhang and G. Gu. NeighborWatcher: A Content-Agnostic Comment Spam Inference System. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2013.

[138] J. Zhang, P. Porras, and J. Ullrich. Highly Predictive Blacklisting. In *Proceedings of 17th USENIX Security Symposium (USENIX)*, San Jose, CA, USA, 2008.

[139] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zhou. SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *Proceedings of the 2ed Workshop on Security and Privacy in Smartphones and Mobile Devices*, Raleight, NC, USA, 2012.

[140] Y. Zhou and X. Jiang. Android Malware Genome Project, August 2012. `http://www.malgenomeproject.org/`.

[141] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33th IEEE Security and Privacy*, San Francisco, CA, USA, 2012.

[142] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2012.

[143] Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, Low Wood Bay, Lake District, UK, 2012.