Permlyzer: Analyzing Permission Usage in Android Applications

Wei Xu, Fangfang Zhang, and Sencun Zhu
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA
Email: {wxx104,fuz104,szhu}@cse.psu.edu

***Abstract*—As one of the most popular mobile platforms, the Android system implements an install-time permission mechanism to provide users with an opportunity to deny potential risky permissions requested by an application. In order for both users and application vendors to make informed decisions, we designed and built Permlyzer, a general-purpose framework to automatically analyze the uses of requested permissions in Android applications. Permlyzer leverages the combination of runtime analysis and static examination to perform an accurate and in-depth analysis. The call stack-based analysis in Permlyzer can provide fine-grained information of the permission uses from various aspects include location, cause and purpose. More importantly, Permlyzer can automatically explore the functionality of an application and analyze the permission uses. Our evaluation using 51 malware/spyware families and over 110,000 Android applications demonstrates that Permlyzer can provide detailed permission use analysis and discover the characteristics of the permission uses in both benign and malicious applications.**

I. Introduction

Android has become one of the most popular mobile platforms. There are over 675,000 active applications [1] in Google Play App Store and the growth rate of new applications is over 12,000 per month [2]. Besides, there are more than 100,000 active publishers who are publishing new applications [2]. Android's popularity owes much to its comprehensive framework API. This API provides applications with the capability of accessing hardware information (e.g., GPS location), reading phone state, reading/writing user's data, modifying phone settings, etc.

Some API methods involve security/privacy sensitive operations. Therefore, Android enforces a permission-based mechanism to restrict and to control the use of these sensitive API methods. Android requires applications to declare all the required permissions, and Android will prompt declared permissions to users at the beginning of an installation process. Users can choose to accept all the permissions or to cancel the installation. Once a required permission is granted by a user, it cannot be withdrawn unless the application is uninstalled. Besides, permissions cannot be granted after installation. Android's install-time permission mechanism provides users with an opportunity to protect themselves from vulnerable and/or malicious applications. However, there are several issues with this mechanism. First, the information provided to users about the risks of the permissions is high-level and coarse-grained. For example, the information on "`INTERNET`" permission is "full Internet access". Based on the information, users can neither know the detailed use of requested permissions (e.g., where and how the permission is used), nor understand the potential risks of granting the permissions. In fact, Au *et al.* [3] show that users often ignore the information and grant all the requested permissions. As a result, providing fine-grained information about the use of permissions is necessary for users to make informed decisions. It can also improve the effectiveness of the install-time permission mechanism.

Another issue is that install-time permission mechanism does not examine the actual use of the requested permissions before asking user to grant the requests. Stowaway [4] partially solved this problem by statically examining requested permissions in applications using a permission-to-API-calls map. However, it is difficult for static examination to determine whether a permission is actually used or the real use of the permission. For example, malicious applications may obfuscate the malicious code that actually requires a permission and insert unnecessary API methods that also require the permission. In this way, the request of the permission will be justified by static examination because of the unnecessary API methods. Other than obfuscation, Java reflection can also make static analysis-based approaches prone to inaccuracies [5], [6].

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

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

Contributions. We make the following contributions in this work:

1) We designed and implemented Permlyzer, a framework for automatically analyzing the use of permis-

sions in Android applications. We evaluate Permlyzer by analyzing the characteristics (abstracted as findings) of the use of permissions among 51 malware/spyware application families and over 110,000 free applications.

2) We proposed a call stack-based analysis approach to extract fine-grained information about various aspects of the use of permissions including locations, causes and purposes.
3) We proposed a functionality exploration scheme that can automatically explore the functionality of Android applications. We also developed an algorithm to efficiently construct call stacks. Using both techniques, Permlyzer can automatically and efficiently analyze the permission uses on a large scale.
4) To the best of our knowledge, Permlyzer is the first tool that can automatically discover the characteristics of the permission uses from a large set of applications. In fact, some of the findings in our evaluation have been confirmed by other works but as a result of manual efforts.

Organization. The rest of the paper is organized as follows: Section 2 reviews the Android permission model. Section 3 provides an overview of Permlyzer. Section 4 elaborates the design of Permlyzer. Section 5 evaluates Permlyzer by analyzing the use of permissions in malicious applications and free applications respectively. Section 7 discusses the limitation of Permlyzer and Section 8 reviews the related work. Finally, Section 9 concludes this work.

II. Android Permission Model and Android Applications

A. Android Permission Declaration and Enforcement

Both the Android system and an application can define permissions, but most of the permissions requested by Android applications are defined by the Android system. This is because Android-defined permissions control the access to sensitive resources and functionalities. There are 130 Android-defined permissions [3], among which 122 permissions are available to third party applications [7]. Permissions are defined with one of the four different protection levels, which characterize the potential risks implied in the permission and enforce different install-time approval processes. These four levels include: 1) *Normal* 2) *Dangerous* 3) *Signature* and 4) *SignatureOrSystem*. Only dangerous permissions are prompted to users for their explicit approval. Signature permissions are automatically granted when requesting application is signed with the same certificate as the application that declared the permissions. SignatureOrSystem permissions are essentially limited to applications that are pre-installed in Android's "/system" partition OR signed with the firmware key. Normal permissions are always granted by the system automatically. Android-defined permissions are checked when an application tries to interact with the Android API or to access a system content provider or to send and receive specific system Intents.

In this work, we focus on *Android-defined permissions* that have a dangerous protection level, since these permissions can cause security and privacy risks and are more frequently used than other permissions.

B. Android Applications

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

Android applications are built upon application components, which include four types: activities, services, content providers and broadcast receivers. Each of these components has its own life cycle and UI. Most of the components can be invoked individually. We focus on activities and service components since most functionality of an application is implemented in these two types of components.

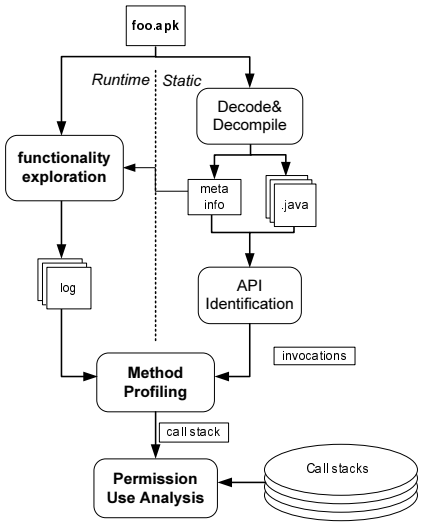

Fig. 1. An Overview of Permlyzer

III. Overview

The goal of this work is to design a tool that is capable of in-depth analysis of the use of permissions in Android applications. To achieve this goal, we outline the following design requirements: (R1) Capability to analyze permissions from various aspects (e.g., locations, causes and purposes). Information collected from different aspects can characterize the use of permissions and provide detailed information to users and developers; (R2) Resiliency to static evasions. As we pointed out, existing static analysis-based approaches can be spoofed by inserting unnecessary or unreachable API methods that require sensitive permissions. Therefore, it is important to identify the real "user" of a requested permission and the necessity of the permission; (R3) Capability to analyze different permissions. Different permissions have different purposes. Some permissions protect the access of sensitive information, and some permissions restrict the invocation of sensitive operations. As a consequence, an analysis approach that can only track sensitive information may not be able analyze permissions that do not involve any sensitive information. Given this, the analysis approach should be generic so that it

can be applied to various permissions with different purposes; (R4) Scalability to analyze a large number of applications. Since both the number of existing Android applications and the increasing rate of new Android applications are high, the analysis approach should be efficient so that it can analyze millions of applications.

To meet these requirements (R1-R4), we design Permlyzer to leverage information collected from both an application's runtime and its source code. Figure 1 illustrates an overview of Permlyzer. At a high level, Permlyzer first decodes and decompiles an application's installation package into Java source files. It extracts meta-information (e.g., list of requested Android permissions) about the application. For each requested Android permission, Permlyzer lists the Android API methods, the invocation of which can trigger the permission check, based on the permission-to-API-calls map [4]. After that, it automatically explores the functionality of the application and logs the execution. These log files are then processed using method profiling to identify the permission triggering API calls and to analyze the context of these calls. Method profiling also leverages static analysis to locate the identified API calls in the Java source files. Based on the call stacks of the permission triggering API calls, Permlyzer can analyze the use of each checked Android permission in terms of location, cause and purpose of the permission use. In the analysis, Permlyzer can also evaluate the potential security/privacy risks in the use of Android permissions.

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

IV. Automated Analysis of the Use of Permissions

In this section, we discuss our techniques to address the listed challenges C1-C3. To address C1, we propose a scheme to automatically explore the functionality of an application (Section IV-A). To address C2, we propose a search tree-based algorithm that can build call-stacks of API methods from logs (Section IV-B). To address C3, we develop an analysis framework to examine the use of permissions from different perspectives (Section IV-C).

A. Functionality Exploration

Permlyzer explores the functionality of an application by invoking its activity and service components since most application components (i.e., activity and service) can be executed individually. Permlyzer first identifies all the activity and service components, and it then starts each of the identified components individually to reveal the functionality implemented in that component. The identification of activity/service components is achieved by parsing the meta-information stored in the AndroidManifest.xml file. Permlyzer parses the ids of "`<activity>`" and "`<service>` " tags in the AndroidManifest.xml file. To start an identified component, Permlyzer parses the intent-filter and sends out Intent messages to the target component using the Android debug bridge (adb) console.

Each activity component defines multiple functions that can only be triggered by proper user events. To trigger these functions, Permlyzer first leverages the layout information of the UI. Each activity has its layout information, which specifies the types of the UI elements (e.g., textview, button, etc.) as well as their positions. With the layout information, which is stored in the "/res/layout/", Permlyzer can send specific user events to the positions of the target UI elements so that it can automatically trigger the functions. Permlyzer uses the adb tool and a testing tool MonkeyRunner [8] to generate and to send user events. However, the positions of some UI elements are not explicitly defined. This is because either the positions are inherited from the parent objects in the view tree, or the positions are relative in order to fit in various screen sizes.

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

By applying both activity-based and layout-based UI elements interaction, Permlyzer can automatically explore most of the functionality of an activity component. We note that the mechanism has some limitations and we will discuss them in Section 7. Service components do not provide any user interface. Therefore Permlyzer does not need to inject user events in order to trigger the functions in service components.

B. Call Stacks Construction

When each activity/service is started, it is started with a profiler, which will log all the function invocations in an activity. Profiling is a functionality of the activity manager (am) in adb. As a part of the official adb toolkit, profiling using am is more reliable and efficient than instrumenting the Android system. However, the profiling will generate a log file contains several millions of method invocations. The construction of call-stacks for hundreds of candidate API methods from the log files proves to be a very time-consuming task given there are usually tens of log files generated for one application. The efficiency of call-stack construction is important for Permlyzer to be able to scale to a large number of methods and Android applications. To this end, we propose a search tree-based algorithm to improve the efficiency of constructing call stacks.

We first use a search tree to speed up the identification of an API method in logs. Each leaf node in this tree represents an API name, and the depth of the tree equals to the length of the

longest common substring between any two API names. For each invoked method in the log, Permlyzer compares the name of the method to the names of the candidate API methods. The comparison is performed by following the path composed of the characters in the method name. If there exists such a path in the tree and the path ends at a leaf node, a match to one of the candidate API methods has been found.

To build a call stack, an intuitive approach would be to trace back from the identified API method to the root. The complexity of this approach is $O(nm)$, where n is the number of method invocations in a log and m is the number of candidate API methods. To improve the performance, we propose an algorithm that can avoid tracing back by maintaining a call-stack during the search for candidate API methods. Each line in a log file contains the following information: the thread ID, the name of the method and the call-depth of the invocation. At each step, the call-stack is updated by comparing the call-depth of the current invocation with that of the previous invocation. The algorithm is listed in 1. The complexity of our proposed algorithm is $O(n)$.

Algorithm 1 Call Stacks Construction

Input: a sensitive API set $A = a_1, ..., a_n$, a log file tf
Output: call stacks CS_{a_i} for each found API a_i

an array of thread number $TN \leftarrow empty$
$i \leftarrow 0$
while $i < length(tf)$ **do**
  apiname $an \leftarrow getAPIName(tf[i])$
  apidepth $d \leftarrow getAPIDepth(tf[i])$
  threadid $t \leftarrow getThreadID(tf[i])$
  if $t \notin TN$ **then**
    a stack $CS_t \leftarrow empty$
    a array $CD_t \leftarrow empty$
    $TN.append$(t)
  end if
  if $d = 0$ **then**
    $CS_t.push(an)$
    $CD_t \leftarrow [0]$
  else
    if $d \geq length(CD_t) - 1$ **then**
      $CS_t.push(an)$
      if $d > length(CD_t) - 1$ **then**
        $CD_t[lengh(CD_t) : d] \leftarrow 0$
      end if
      $CD_t[d] \leftarrow length(CS_t) - 1$
    else
      $CS_t.pop()$ (repeat $length(CS_t) - 1 - CD_t[d]$ times)
      $CS_t.push(an)$
      $CD_t[d] \leftarrow length(CS_t) - 1$
      $CD_t \leftarrow CD_t[0 : d + 1]$
    end if
  end if
  if $an \in A$ **then**
    $CS_{an} \leftarrow CS_t$
  end if
end while

C. Analysis of Permission Use

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

1) The Location of Permission Use: To identify where a permission is used, Permlyzer searches for the API methods that can trigger the check of the permission in the logs recorded in functionality exploration. For each found API call, Permlyzer also analyzes its call stack to determine the component where the API call resides in. That is, Permlyzer traces back from the API call to the user-defined method that invokes the API method and the container of the user-defined method (i.e., a user-defined class). The names of all user-defined classes are obtained by parsing the source files so that they can be distinguished from Android libraries and Java native methods.

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

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

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

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

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

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

V. Evaluation

We evaluate the analyzing capability and scalability of Permlyzer in this section. The main purpose of the evaluation is not to present the findings, but to use these findings to demonstrate Permlyzer's capability of analyzing permission use automatically, which is the main contribution of this work.

A. Collection of Android Applications

We collected two datasets. In the first dataset, we collected 325 known malicious Android applications from VirusTotal [9] [2]. All of the collected malicious applications have been detected by more than four anti-virus vendors listed on VirusTotal. The second dataset contains 113,237 free Android applications collected from 25 categories in Google Play App Store.

B. Permission Analysis Coverage

In the evaluation, Permlyzer covers 90% of the permissions requested by applications. Note that Permlyzer leverages the API-to-permission map provided in Stowaway [4]. Therefore, if a permission does not map to any API in the API-to-permission map, Permlyzer cannot analyze the use of that permission. We acknowledge that the API-to-permission map obtained in [4] is based on Android 2.2 (Android API level 8), which are currently used in 12.9% Android devices [10]. [10] also shows that Android 2.3 (Android API level 9,10) is the most widely installed Android version (e.g., on 55.8% of the devices). Besides, the difference between Android 2.2 and Android 2.3 in terms of code happens only in 58 classes [11], [12], most of which are adding new fields instead of changing API-to-permission map.

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

C. Performance Analysis

The evaluation is performed on a commodity machine with 1.6 GHz dual core processor and 2 GB RAM. On average, Permlyzer took less than 3 minutes to analyze one application. Most of the time (i.e., over 70%) is consumed by method profiling, which includes uploading the application to a phone, installing the application, starting activities, generating logs and delete the application. The decoding and decompiling process takes less than 10 seconds on average. The time used by method profiling varies from a few seconds to tens of seconds depending on the size and number of log files.

D. Permission Use in Malicious Applications

Malicious applications often demonstrate distinct features in permission use. To evaluate Permlyzer's capability in exposing and characterizing these features, we first manually determined the malicious behavior of these applications. After that, we used Permlyzer to analyze the permission use in each application and to discover the connection between aggregated permission uses of a malware category and its malicious behavior.

1) Malicious Behavior and Permission Requests: We identified 51 Android malware/spyware families and categorized their malicious behavior into four categories [3]:

- Collect and Send (C&S). Malicious applications collect sensitive information (i.e., device ID, IMEI/IMSI, device model, geographic location, personal SMS etc.) from the phone and then send the (encrypted) information to a remote server. The information is normally sent through SMS or Internet. 47% of the malicious applications exhibit this behavior.
- Obtain a Financial Gain (OFG). One of the goals of malicious application developers is to obtain a financial gain. This is achieved in three ways based on observed behavior. Some applications send SMS messages to predetermined or fetched premium rate numbers to charge unwitting users. Some applications silently issue multiple HTTP requests to promote a specific website. There are also some applications that register the phone to a charged service by sending the phone number in a SMS message to the service. 57% of the malicious applications exhibit this behavior.
- Annoy Users (AU). Some applications affect smartphone users by performing annoying operations without user's awareness. These actions include displaying advertisements, consuming battery, modifying the configuration of the phone (e.g. changing the wallpaper, etc). 6% of the malicious applications exhibit this behavior.

[2]The collected malicious applications are submitted to VirusTotal between Dec, 2011 to March, 2012

[3]some malicious applications exhibit behavior that are in more than one category

TABLE I. ANALYSIS OF PERMISSION USE IN MALICIOUS APPLICATIONS

Behavior	Top checked Permissions	Top Locations	Top Causes
C&S	INTERNET	java.net.URL.openConnection()	Activity.OnCreate
	READ_PHONE_STATE	android.telephony.TelephonyManager.getDeviceId()	Activity.OnResume
	ACCESS_WIFI_STATE	android.net.wifi.WifiManager.getConnectionInfo()	Service.OnStart
	ACCESS_COARSE_LOCATION	android.location.LocationManager.getLastKnownLocation()	User events
OFG	SEND_SMS	android.telephony.SmsManager. sendTextMessage()	Activity.Run
	INTERNET	android.webkit.WebView.<init>()	Activity.OnCreate
	READ_PHONE_STATE	android.telephony.TelephonyManager.getLine1Number()	User events
	CAMERA	android.telephony.TelephonyManager.getDeviceId()	User events
AU	WAKE_LOCK	android.os.PowerManager$WakeLock.acquire()	Activity.OnStart
	ACCESS_WIFI_STATE	org.apache.http.impl.client.DefaultHttpClient.<init>()	Activity.OnResume
	INTERNET	android.location.LocationManager.getLastKnownLocation()	Service.OnStart
	ACCESS_COARSE_LOCATION	android.net.wifi.WifiManager.getConnectionInfo()	User events
MSA	READ_PHONE_STATE	java.lang.Runtime.exec()	Activity.OnCreate
	INTERNET	android.telephony.TelephonyManager.getDeviceId()	Activity.OnStart
	READ_LOGS	java.net.URL.openConnection()	Service.OnStart
	ACCESS_FINE_LOCATION	android.telephony.TelephonyManager.getCellLocation()	User events

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

This categorization is by no means a comprehensive one, but it does cover a large part of the malicious behavior observed in Android applications [13]. Our analysis shows that these malicious applications request 6.5 Android permissions on average and 64 unique permissions in total.

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

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

1) Checked Permissions:

Finding 1: Permlyzer's results show that permissions that are checked in runtime indicate the behavior observed in the malicious applications. For instance, for applications that exhibiting "Collect and Send" behavior, permissions such as "INTERNET" and "ACCESS_WIFI_STATE" are frequently checked because these permissions are required to send out the sensitive information via local network proxy. "READ_PHONE_STATE" permission is also checked by most of the applications in this behavior category because this permission is necessary to obtain identification information of the phone (e.g., device ID). For malicious applications aiming at obtaining a financial gain, permission such as "SEND_SMS" is the topmost checked permission because it is essential for sending SMS to premium rate numbers, which is the most commonly observed behavior in this category. Another permission that is frequently checked by applications that try to obtain a financial gain is "READ_PHONE_STATE". This permission is used to obtain the phone number so that the applications can register the phone to a charged service by sending the phone number to the service. For malicious applications that monitor the activities of the phone, two unique permissions are frequently checked are "READ_LOGS" and "ACCESS_FINE_LOCATION". We noticed that "READ_LOGS" is not checked by applications in other categories because those applications do not access low-level system log files (e.g., activity manager state)

[4]the causes listed in Table I are summarized from multiple instances for the sake of clarity. Therefore, we use the description (e.g., "element.OnClick") to represent a click event on all UI elements.

2) Locations of Permission Use:

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

TABLE III. CAUSE OF PERMISSIONS IN MALICIOUS APPLICATIONS

Cause	Behavior Category			
	C&S	OFG	AU	MSA
User Event	3.8%	28.2%	4.6%	0%
Activity	86.7%	71.7%	83.9%	27.1%
Service	9.3%	0%	11.3%	72.9%

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

Finding 3: Permlyzer's results show that over 90% of the permission uses in malicious applications are caused by the start of application components (i.e., activities and services), whereas less than 10% of the permission uses are caused by user events (e.g., touch, click, etc). Combining with previous finding that the default entrance activity (i.e., main activity) is the location where most malicious behavior is coded, it is clear that permission checks related to malicious behavior are mostly triggered without user interaction. Besides, Andriod applications can self start. This characteristic in the permission use of malicious applications suggests that malicious appli-

TABLE II. ANALYSIS OF THE MOST REPRESENTATIVE PURPOSES OF PERMISSION USE IN MALICIOUS APPLICATIONS

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

cation requires minimal, if any, user-interactions in order to perform the behavior. This finding stresses the importance of a user's understanding about an application's permission use at the install-time.

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

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

Finding 5: The results from Permlyzer indicate that correlated permission use can better expose its purpose. For instance, in "Collect and Send" category, the most representative correlation is actually among "`READ_PHONE_STATE`" permission (i.e., triggered by the first 4 API calls), "`ACCESS_FIN-E_LOCATION`" permission (i.e., triggered by the 5th API call), and followed by "`INTERNET`" permission (i.e., triggered by the last two API calls in the sequence). This correlation indicates multiple phone information is first collected and then send to a remote server, which could cause a privacy leak. In "Obtain a financial gain" category, one correlated permission is "`READ_PHONE_STATE`" followed by "`INTERNET`", the purpose of which is to send phone numbers to a remote server and the combined security risks are being registered to a premium service.

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

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

E. Permission Use in Free Applications

We evaluated the scalability of Permlyzer leveraging the large number of free applications. Meanwhile, we also discuss several aggregated results obtained by Permlyzer to demonstrate the capability of Permlyzer in helping us better understand the permission uses in applications.

1) Permission Use in Free Android Applications: Table IV summarizes the analysis of location, cause and purpose of top checked permissions in free applications. Given the large number of applications in our dataset, we only discuss the most representative statistic results in this section.

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

TABLE IV. ANALYSIS OF TOP TEN MOST CHECKED PERMISSION USE IN FREE APPS

Permission	Top Locations	Top Causes	Top Purpose
INTERNET	java.net.Socket.<init> android.webkit.WebView.<init>	Activity.OnCreate Activity.OnStart	Client/Server communication
READ_PHONE_STATE	android.telephony.TelephonyManager.getDeviceId android.telephony.TelephonyManager.getLine1Number	Activity.OnStart Activity.OnResume	Collect and send device ID
ACCESS_COARSE_LOCATION	android.location.LocationManager.getBestProvider android.location.LocationManager.getLastKnownLocation	Activity.OnStart Activity.OnResume	Obtain location info.
ACCESS_WIFI_STATE	android.net.wifi.WifiManager.getWifiState android.net.wifi.WifiManager.isWifiEnabled	Component.Run Activity.OnCreate	Connect to remote server
ACCESS_FINE_LOCATION	android.location.LocationManager.isProviderEnabled android.location.LocationManager.getLastKnownLocation	Activity.OnResume Activity.OnStart	Obtain location info.
WAKE_LOCK	android.os.PowerManager$WakeLock.release android.os.PowerManager$WakeLock.acquire	Activity.OnPause Activity.OnResume	Keep device on
READ_CONTACTS	android.content.ContentResolver.query android.content.ContentResolver.openFileDescriptor	Component.OnCreate User events	Query user data
GET_ACCOUNTS	android.accounts.AccountManager.getAccountsByType android.accounts.AccountManager.getAccounts	User events Activity.OnCreate	Purchase and billing
READ_LOGS	java.lang.Runtime.exec java.lang.Runtime.exec	Activity.OnCreate Activity.OnResume	Access low-level system logs
CAMERA	android.hardware.Camera.open android.hardware.Camera.native_setup	User events User events	start camera

TABLE V. PERMISSIONS USE CAUSED BY THIRD-PARTY LIBRARIES

Permission	Ad Network	Analytic	Data	Other
INTERNET	81.3%	10.0%	8.6%	<1%
READ_PHONE_STATE	93.8%	3.5%	0%	2.6%
ACCESS_FINE_LOCATION	28.2%	68.1%	0%	3.7%

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

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

Finding 3: Collection of Location Information Perm-lyzer's results show that 4% of all the permissions checks are related to location access permissions (i.e. "ACCESS_FINE_LO-CATION" and "ACCESS_COARSE_LOCATION"). Moreover, most of the API calls that trigger the checks of these permissions are to obtain the location information of the smartphone. The analysis of Permlyzer shows that the primary cause of these permission checks is the start and/or resume of activity components, which causes 59% of the permission checks. We also notice that 33% of the causes are related to third-party libraries. As shown in Table V, 68.1% of the third-party libraries belong to analytic services. We believe this is because location information can help application developers to monitor the geographical distribution of the downloads and use of their applications. In fact, location information is required by all the analytic services that we have identified. However, the collection of smartphone's location information may also raise privacy concern to the users, especially when the users are unaware of most of such collections, as the primary cause of permission checks suggests.

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

VI. Discussion

A. Limitations of Functionality Exploration

Most of the components can be invoked individually because Android requires each component to maintain its own life cycle. However, a few components can not be started because these components handle the Intent to "startActivity()" or "startService()" improperly. Unfortunately, this can only be fixed by revising the source code. Another issue arises when there exist functionality that can only be triggered by some rare conditions. This limits the functionality coverage of Permlyzer. Since it is also a known issue in software testing, we plan to deal with this issue in the future with more advanced software testing tools.

B. Permission Use in Other Components

Permlyzer focuses on activity and service components since both of these components implement most functionality of an application, especially the functionality for UI. However, some operations on content providers and Android Intents also require permissions. Since most of the content provider components and Intents that are protected by Android permissions are part of the Android system instead of third-party applications, Permlyzer can not be applied on these components. Besides, identification of the checked permissions in operations with content providers and in sending/receiving Intents has been done in Stowaway [4] using ComDroid [14].

C. Applications of Permlyzer

The goal of this work is to provide a general-purpose permission analysis platform that can provide fine-grained information about the use of requested permissions in an application's run-time. Permlyzer can be used in various cases. The information obtained by Permlyzer can be used to assist application users to make more informed decisions regarding granting requested permissions and evaluating potential security/privacy risks based on their own preference. Permlyzer can also help application developers to examine their applications for unnecessary permission requests. Moreover, given Permlyzer's scalability, it can also be used by application vendors to automatically check the use of permissions in large numbers of submitted applications. Permlyzer can detect misuse of Android permissions based on a set of rules defined by the vendors (e.g., a rule can specify any discovered correlation of permission use that is similar to a vendor-defined correlation as a misuse).

Since the information obtained by Permlyzer characterize the permission use in an application, we believe Permlyzer can also be combined with a classification technique to classify the applications based on their permission use.

VII. Related Work

Android Permission Analysis This category includes advancements in analyzing Android permissions. Kirin [15] maps dangerous functionalities with the permissions required to perform them after specifying permission based security rules. Barrera *et al.* [16] studied the permission usage among a variety of categories of applications in Android market by mapping an application to a category based on its requested permissions. Felt *et al.* [17] manually compare the functionalities of 36 Android applications to the permissions requested by these applications. Their results show that 4 out of 36 applications are over-privileged. Felt *et al.* [4] also propose Stowaway, which identifies over-privileged applications by detecting unnecessary permissions for API calls in applications. The mapping provided in [4] is very helpful, but the mapping alone can not explain the cause and the purpose of the use of permissions.

In comparison, Permlyzer can not only analyze the permission use from a static point of view (e.g., code analysis), it can also examine the use of permission in runtime to determine the cause and purpose of the permissions that are actually used. More importantly, the capability of automatically analyzing the use of permissions also makes Permlyzer different from existing approaches. To the best of our knowledge, similar findings in permission uses in other works are all the results of manual efforts. This capability is especially important because both the number and the increase of applications in current Android markets (e.g. Google Play Apps, Amazon Appstore, etc.) are very high.

Smartphone Platform Security This category includes a wide range of approaches [18]–[27] that aims at improving the security and privacy of smartphone platform. For example, TaintDroid [23], based on the Android platform, provides a scheme to monitor a third-party application's usage of sensitive information such as what information leaves a device and where it is sent. PiOS [20] uses control flow analysis followed by data flow analysis to confirm whether private information reaches outbound sink. Privacy Oracle [24] and TightLip [25] are both black-box-based differential testing schemes for PCs to detect sensitive information leakage by third-party applications via network traffic. These approaches leverage various analysis techniques to enhance the security of the platform. In comparison, Permlyzer focuses on extracting information and interpretation of the permission use in applications.

Application Security This category includes approaches to study the security of Android applications. For example, Felt *et al.* [28] propose inter-process communication (IPC) inspection to monitor messages used for IPC and reduces privilege of the recipient to the intersection of recipient's and the requester's permissions. Dietz *et al.* [29] propose QUIRE to track the call-chain of IPC in order to defend the confused deputy attack and to provide a mutual verification scheme for applications. Chin *et al.* [14] ComDroid to detect the vulnerabilities in the inter-application communication. Bugiel *et al.* [30] proposed XmanDroid to prevent privilege escalation. XmanDroid monitors the communications between applications and apply policy to restrict the interaction. Among the most related, Gilbert *et al.* [31] proposed AppInspector, which leverages information-flow tracking on sensitive information to automatically identify security and privacy violation in an application. Both AppInspector and Permlyzer facing similar challenges e.g., analyzing the logs collected from runtime; traverse code paths. The difference between AppInspector and Permlyzer is that AppInspector follows the flow of sensitive information, while Permlyzer logs the invocations of API call related to any Android permissions.

Malicious Application Characterization and Detection The fourth category of related works aim at detecting malicious Android applications. Zhou *et al.* [32] systematically characterize the existing Android malwares. In [33], Zhou *et al.* also leverage the combination of static and dynamic analysis to

detect malicious applications in various Android marketplaces. Grace *et al.* [34] propose a two-order detection tool to identify potentially malicious behaviors. In comparison, the goal of Permlyzer is not to detect malicious applications. However, some of the findings obtained by Permlyzer are similar to the manual analysis results in [32], for example, [32] also shows that many of the malicious application families sending out SMS to premium numbers or harvesting user information. Besides, [32] also discovers that many malicious applications use the entry activity (e.g., "ACTION_MAIN") to trigger the execution.

VIII. Conclusions

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

Acknowledgement

We want to thank the reviewers for their valuable comments and suggestions. This work was supported by NSF CAREER 0643906 and NSF CCF-1320605.

References

[1] Distimo, "Google android market tops 675,000 applications," http://officialandroid.blogspot.com/2012/09/google-play-hits-25-billion-downloads.html 2012.

[2] AppBrain, "Number of available android applications," http://www.appbrain.com/stats/number-of-android-apps 2012.

[3] K. W. Y. Au, Y. F. Zhou, Z. Huang, P. Gill, and D. Lie, "Short paper: a look at smartphone permission models," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ser. SPSM '11, 2011, pp. 63–68.

[4] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11, 2011, pp. 627–638.

[5] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: effective taint analysis of web applications," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '09, 2009, pp. 87–97.

[6] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for java," in *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '06, 2006, pp. 387–400.

[7] "Android manifest.permission." http://developer.android.com/reference/android/Manifest.permission.html 2012.

[8] "Android developer: monkeyrunner," http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html, 2012.

[9] "Virustotal," http://www.virustotal.com 2012.

[10] "Platform versions," http://developer.android.com/about/dashboards/index.html, 2012.

[11] "Android api differences report (8-9)," http://developer.android.com/sdk/api_diff/9/changes.html, 2012.

[12] "Android api differences report (9-10)," http://developer.android.com/sdk/api_diff/9/changes.html, 2012.

[13] "Mobile threat report (q4 2011)," F-Secure Labs, Tech. Rep., 2011.

[14] 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*, ser. MobiSys '11, 2011, pp. 239–252.

[15] 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*, ser. CCS '09, 2009, pp. 235–245.

[16] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," in *Proceedings of the 17th ACM conference on Computer and communications security*, ser. CCS '10, 2010, pp. 73–84.

[17] A. P. Felt, K. Greenwood, and D. Wagner, "The effectiveness of application permissions," in *Proceedings of the 2nd USENIX conference on Web application development*, ser. WebApps'11, 2011, pp. 7–7.

[18] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh, "Cells: a virtual mobile smartphone architecture," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11, 2011, pp. 173–187.

[19] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: trading privacy for application functionality on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '11, 2011, pp. 49–54.

[20] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications," in *18th Annual Symposium on Network and Distributed System Security*. San Diego, California: Internet Society, February 2011.

[21] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX conference on Security*, ser. SEC'11, 2011, pp. 21–21.

[22] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android applications," Tech. Rep.

[23] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Symposium on Operating System Design and Implementation*, ser. OSDI '10. USENIX, October 2010.

[24] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno, "Privacy oracle: a system for finding application leaks with black box differential testing," in *Proceedings of the 15th ACM conference on Computer and communications security*, ser. CCS '08, 2008, pp. 279–288.

[25] A. R. Yumerefendi, B. Mickle, and O. P. Cox, "Tightlip: Keeping applications from spilling the beans," in *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '07. USENIX, April 2007.

[26] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *19th Annual Symposium on Network and Distributed System Security*. San Diego, California: Internet Society, February 2012.

[27] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications (on android)," in *Proceedings of the 4th international conference on Trust and trustworthy computing*, ser. TRUST'11, 2011, pp. 93–107.

[28] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proceedings of the 20th USENIX Security Symposium*, ser. USENIX'11, 2011.

[29] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *20th USENIX Security Symposium*, San Francisco, CA, Aug. 2011.

[30] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischera, and A.-R. Sadeghi, "Xmandroid: A new android evolution to mitigate privilege escalation attacks," Technische Universitat Darmstadt, Center for Advanced Security Research, Tech. Rep., 2011.

[31] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung, "Vision: automated security validation of mobile apps at app markets," in *Proceedings*

of the second international workshop on Mobile cloud computing and services, ser. MCS '11, 2011, pp. 21–26.

[32] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," *Security and Privacy, IEEE Symposium on*, pp. 95–109, 2012.

[33] 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 *19th Annual Symposium on Network and Distributed System Security*. San Diego, California: Internet Society, February 2012.

[34] M. Grace, 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*, ser. MobiSys '12, 2012, pp. 281–294.