Using Provenance Patterns to Vet Sensitive Behaviors in Android Apps

Chao Yang[1], Guangliang Yang[1], Ashish Gehani[2],
Vinod Yegneswaran[2], Dawood Tariq[2], and Guofei Gu[1]

[1] Texas A&M University, TX, USA
[2] SRI International, CA, USA

Summary. We propose Dagger, a lightweight system to dynamically vet sensitive behaviors in Android apps. Dagger avoids costly instrumentation of virtual machines or modifications to the Android kernel. Instead, Dagger reconstructs the program semantics by tracking provenance relationships and observing apps' runtime interactions with the phone platform. More specifically, Dagger uses three types of low-level execution information at runtime: system calls, Android Binder transactions, and app process details. System call collection is performed via Strace [7], a low-latency utility for Linux and other Unix-like systems. Binder transactions are recorded by accessing Binder module logs via *sysfs* [8]. App process details are extracted from the Android */proc* file system [6]. A data provenance graph is then built to record the interactions between the app and the phone system based on these three types of information. Dagger identifies behaviors by matching the provenance graph with the behavior graph patterns that are previously extracted from the internal working logic of the Android framework. We evaluate Dagger on both a set of over 1200 known malicious Android apps, and a second set of 1000 apps randomly selected from a corpus of over 18,000 Google Play apps. Our evaluation shows that Dagger can effectively vet sensitive behaviors in apps, especially for those using complex obfuscation techniques. We measured the overhead based on a representative benchmark app, and found that both the memory and CPU overhead are less than 10%. The runtime overhead is less than 63%, which is significantly lower than that of existing approaches.

1 Introduction

With the proliferation of Android smartphones and applications, there is a growing interest in scalable tools and techniques for blackbox testing of applications. Of specific interest are tools that enable screening for suspicious behavior patterns commonly exhibited by malware. While a rich body of prior work exists, contemporary static and dynamic analysis techniques fall short in many respects.

Static analysis techniques [55, 56, 48] analyze Android apps by disassembling them into Dalvik (or Java) source code, and further evaluating the permissions list, analyzing programming interfaces (i.e. Android APIs) and program logic used in the source. However, such approaches are unable to cope with complex code obfuscation techniques (e.g., source encryption, noise insertion, and use of Java reflection) or analyze code logic that uses the Android Native Development Kit (NDK) [1].

[1] The volume of apps involving native code has dramatically increased in recent years [53, 33].

In contrast, dynamic analysis approaches monitor apps' behaviors by running them in real or emulated Android environments. Certain systems (e.g., [51]) rely on application source instrumentation to record API invocation details (e.g., API names and parameter values). However, such approaches are blind to malicious logic implemented using NDK. A few dynamic approaches [50, 41] employ virtual machine introspection (VMI) techniques to gather the lower-level system information and thereby reconstruct high-level application semantics. Such approaches typically incur high performance overhead, especially when taint tracking is enabled. Thus, direct application of these approaches is impractical for analysis of a large corpus of apps.

We present Dagger as a lightweight system to dynamically vet sensitive behaviors in Android apps. Dagger avoids costly overheads and complexities associated with virtual machine instrumentation and modifications to the Android kernel. Instead, Dagger reconstructs the apps' semantics by tracking its runtime interactions with the phone platform and building provenance relationships. More specifically, at an app's runtime, Dagger uses the open source SPADE [26] provenance middleware to collect three types of low-level execution information, including Linux system calls, Android Binder transactions, and app process details. System call collection is done via Strace [7], a low-latency utility for Linux and other Unix-like systems. Binder activity is recorded by accessing transaction logs via *sysfs* [8]. App process details are extracted from the Android */proc* file system [6]. A data provenance graph is then built to record the interactions between the app and the phone system based on these three types of information. Dagger identifies behaviors by matching the provenance graph with a library of *sensitive provenance patterns* that have been previously extracted by carefully studying the inner workings of the Android framework.

We have built a prototype of Dagger, and evaluated both its effectiveness and efficiency. We first used Dagger to vet three representative Android malware families. These case studies demonstrate the effectiveness of Dagger in vetting sensitive behaviors that are implemented in more evasive ways (e.g., code obfuscation or encryption). Then, we evaluated Dagger on a large corpus of apps, which consists of over 1200 known malicious apps, and 1000 official apps randomly selected from a set of over 18,000 samples downloaded from Google Play. Our evaluation demonstrates that Dagger can effectively vet sensitive behaviors in a large scale of apps. To evaluate system efficiency, we used a popular benchmark app called AnTuTu (v 3.0.3) [1] that measures Android system overhead. We found both the memory and CPU overhead to be less than 10% and the runtime overhead to be less than 63%, which is significantly lower than that of existing approaches that utilize VMI techniques (e.g., [50]). To summarize, the salient contributions of this paper include the following:

1. Design of a lightweight approach for runtime tracking of sensitive behavior that does not rely on the high overhead techniques of virtual machine introspection or Dalvik monitoring.
2. Development of the Dagger prototype that automates the abstraction of Android apps' runtime low-level execution information into high-level behavior semantics using the data-provenance approach.
3. Development of a library of sensitive provenance patterns for vetting Android apps.

4. Comprehensive system evaluation on a corpus of over 2200 benign and malicious applications that demonstrates how Dagger can be used to efficiently vet sensitive behaviors with minimal memory and runtime overhead.

2 Background And System Goals

The Android operating system is built on the top of the Linux kernel and organized in a layered architecture consisting of four layers: (i) the Linux kernel, (ii) Android's native system libraries and Dalvik virtual machine runtime, (iii) Android's application frameworks, and (iv) a collection of installed applications.

Linux Kernel: The bottom layer of the Android system is a customized Linux kernel. It provides services such as memory and process management, access control, and a driver framework. As the abstraction between the hardware and software, this layer provides generic services to the user space layer above while hiding the details of the hardware. Android also enhances the standard Linux kernel in several respects, including inter-application communication and power management. Android implements a custom inter-process communication (IPC) mechanism called Binder. Binder is used to mediate interactions between apps, as well as between apps and the operating system.

Android Libraries and Runtime: This layer contains two major parts: Android libraries and the Dalvik virtual machine runtime. The libraries consist of C and C++ code that compiles to the native binary format. The functionality in these libraries is exposed to applications from third party developers through the Android framework.

Android Framework: Many of the application-level functionalities for interacting with system resources are provided by the Android framework. It provides the interfaces (*Android Framework APIs*) to access the system apps; that is, components that provide indirect access to the underlying system resources (such as reading contacts, recording the current geographic location, or sending SMS messages) by invoking *system calls*, low-level interactions between app processes and GNU/Linux. For instance, the framework API of TelephonyManager.getDeviceId() provides the functionality of reading device ID; SmsManager.sendTextMessage() supports sending text messages. These framework APIs essentially achieve the functionalities by invoking low-level system calls, e.g., open(), which opens file operators, and execve(), which executes shell commands. Thus, the usage of the low-level system calls and the access of Android resources in the runtime can indicate rich high-level behavior semantics.

Applications: Android distributions include a collection of system apps, including: one that provides the functionality of a phone, another that allows short message service (SMS) and multimedia message service (MMS) messages to be sent and received, an email client, a calendar, and a contact manager. The core set of applications also export services to third party applications through APIs in the Android application framework.

2.1 System Goals

Our objective is to design an effective and efficient system for vetting sensitive behaviors in Android apps that does not rely on VMI techniques or modifications to the operating system. In Table 1, we list a set of sensitive behavioral patterns in Android apps (Phone Call, Send SMS, Block SMS, Steal SMS, Steal Contact, Track Location,

Table 1. Malicious Android app behaviors targeted by prior work

Work	**Type**	**Financial Charge**			**Privacy Leak**			**Remote Control**	**Rooting**
System	Technique	Phone Call	Send SMS	Block SMS	Steal Contact	Track Location	Steal Phone Number	Net	Execute Shell
[54]	Static	√	√	√			√	√	√
[51]	Dynamic		√	√	√	√	√		
[50]	Dynamic						√	√	√

Table 2. Fined-grained sensitive behaviors associated with malicious behaviors

Index	**Malicious Behaviors**	**Sensitive Behaviors**	**Index**	**Malicious Behaviors**	**Sensitive Behaviors**
1	Phone Call	Phone Call	5	Steal Contact	Read Contact and Net
2	Send SMS	Send SMS	6	Track Location	Read Location and Net
3	Block SMS	Receive SMS, not Write SMSDB	7	Execute Shell	Execute Shell
4	Steal SMS	Read SMSDB and Net	8	Net	Net

Steal Phone Number, Network Connection, and Execute Shell) that have been targeted by prior studies as indicators of malicious behavior.

Instead of focusing on such coarse-grained malicious behaviors, we designed Dagger to vet fine-grained sensitive behaviors that may be launched by both malicious and benign apps. As seen in Table 2, the aforementioned malicious functionalities can essentially be achieved by multiple fine-grained sensitive behaviors. In Table 2, we list 9 fine-grained sensitive behavioral patterns associated with the 8 malicious behaviors listed in Table 1. These are: Phone Call, Send SMS, Receive SMS, not Write SMSDB, Read SMSDB, Net, Read Contact, Read Location, and Execute Shell. (*Read SMSDB* and *Write SMSDB* refer to reads from and writes to the Android provider *content://sms/inbox/*.)

3 System Design

A rich body of prior work have attempted to vet the behavior of desktop applications by analyzing system call invocations. However, such approaches cannot be directly extended to vet the behavior on Android apps due to the unique aspects of the Android system. (1) Android apps access kernel resources through the Android application framework. Consequently, there is a semantic gap between low-level system call invocations and high-level Android-specific behavior. (2) Android apps interact with system services and the Android framework through the Binder IPC mechanism, which is unique to Android. Thus, vetting Android app behavior requires analysis of the Binder transactions that occur between apps and the system. (3) Android is an event-driven system; its multiple behavior patterns interweave together. Therefore, traditional temporal monitoring approaches are not effective during analysis of Android malware.

Dagger's design is motivated by the observations that an Android app's behaviors are achieved through (i) low-level interactions (system calls and Binder IPC) between app process and the Android kernel and (ii) accesses to underlying system resources (e.g., contacts, geo location, SMS messaging). Dagger uses data provenance analysis to first translate an app's runtime behaviors into a provenance graph that captures three types of low-level information: system call invocations, Binder IPC transaction logs, and process details. Essentially, the graph captures all interactions of the app with the Android application framework and the OS kernel. Dagger further identifies sensitive

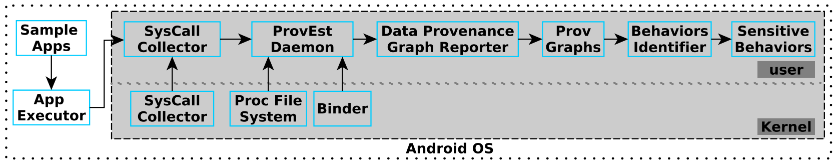

Fig. 1. Dagger takes a corpus of apps, runs each one, collects provenance records, and performs pattern matching to identify potentially sensitive behaviors.

behaviors by matching the provenance graph with sensitive provenance patterns that have been extracted and developed through careful analysis of the inner workings of the Android framework.

To understand the internal logic of the Android framework, we ran Android apps with selected input that is known *a priori* to trigger sensitive behavior. We utilized two broad approaches for this investigation. In the first approach, we manually selected representative malware samples that belong to particular families with known sensitive behavior. We then used Androguard, a static analysis tool, to extract the relevant logic that would trigger sensitive behavior in each piece of malware. In a complementary approach, we triggered flows in synthetic apps that were developed to contain representative sensitive behavior.

3.1 Design Overview

Dagger is built on the open source SPADE provenance middleware [26]. Dagger is composed of five major components, as illustrated in Figure 1: AppExecutor, SysCall Collector, ProvEst Daemon, Graph Reporter, and Behavior Identifier. Sample apps are first loaded into the App Executor, which automatically executes the app in a sandbox Android runtime environment. Once the app is executed, SysCall Collector starts to collect the system call invocations, and ProvEst Daemon analyzes the binder transactions and collects more detailed information of the process in order to build the provenance relationships of the identities (e.g., processes and files) in the system call invocations. The Graph Reporter outputs the data provenance graph according to the provenance relationships established by the ProvEst Daemon. Finally, the Behavior Identifier detects sensitive behaviors from the provenance graph according to the working mechanism of the Android system. Below, we discuss each component in greater detail.

1. App Executor is a Python script for controlling app execution. It first extracts the package and activity names (including the main activity) from the Android package (APK file), installs the package, and then automatically launches selected activities by using Android debugger `adb` commands.

App Executor uses MonkeyRunner [9] to drive the app with randomly generated events (such as pressing buttons or touching the screen). It first extracts the main activity of the app, and then sends an *intent* to initiate the activity. App Executor continues till it has generated at least 500 events or the app has run for at least three minutes.

2. SysCall Collector records low-level system call invocations (e.g., fork, read, write, setuid32) using the `strace` utility. Each system call invocation is internally recorded in the following format:

$[pid][timestamp][syscall(paramenters)] = [return]$

for example, “183 16:54:15.805684 open(”/dev/binder”, O_RDWR) = 9”. The output of SysCall Collector is persisted in non-volatile storage. To avoid app-specific storage limits, the log is stored in the mobile device’s Secure Digital (SD) card. The SysCall Collector functionality was developed by extending SPADE’s *Strace Reporter* so it can run on Android (in addition to Linux).

3. ProvEst Daemon generates data provenance relationships by collecting system calls, Binder transactions, and process details. A data provenance record describes how a piece of information was derived, a historical approach which has been widely used in a variety of fields such as performance optimization, scientific computation, security verification, and policy validation. The data provenance graphs in Dagger conform to the Open Provenance Model [34] which has three types of elements, as illustrated in Figure 2. ProvEst leverages significant functionality from SPADE (that is summarized below), and augments the *Strace Reporter* with Android-specific details (from Binder transactions, for example).

Process Vertices. These are created to record dynamic entities; typically, these entities are operating system processes created by app execution. Each vertex contains a range of annotations, including the name of the process, the process identifier (pid), and the owner (uid) and group (gid). It also records the parent process, the command line with which it was invoked, and the values of environment variables.

Artifact Vertices. These are used to represent static elements that are consumed or produced by processes. There are four subtypes of such vertices: (i) *File Vertex*, which represents a file read or written by a process at a particular point in time; (ii) *Binder Vertex*, which denotes a Binder transaction that occurred between a pair of processes; (iii) *Socket Vertex*, which indicates a communication from or to a process through a socket; and (iv) *Command Vertex*, which records the details of high-level commands (e.g., AT commands, described in Section 3.2) issued by a process.

Edges. These are directed and used to represent the dependency between a pair of vertices. For example, an edge to a file vertex indicates that the file was read, and an edge from a file vertex indicates that the file had been modified. There are four types of edges: (i) *WasTriggeredBy*, from a process to another process; (ii) *WasGeneratedBy*, from an artifact to a process; (iii) *Used*, from a process to an artifact; and (iv) *WasDerivedFrom*, from an artifact to another artifact.

Given the design of the provenance graph, once a new entry is collected by the SysCall Collector, the ProvEst Daemon parses it to extract the *pid* of its process. Based on the *pid*, it further extracts its process details (e.g., process name, GID, UID, command line, etc.) from the “/proc” file system [6]. All these details are used to depict the process as a vertex in the graph. Every file, socket, and pipe that is accessed by the process is depicted as a single artifact vertex. Once the system call *ioctl()*, which leads to a Binder transaction, is invoked by one process, the Daemon inspects the Binder transaction log from *sysfs* [8], and extracts the communicated process in the transaction. Then, a directional edge is built from the request process to the response process. Edges are also generated to record accesses of sensitive system resources (e.g., read and write operations of content providers) from the app’s process vertex to the resource artifact vertex.

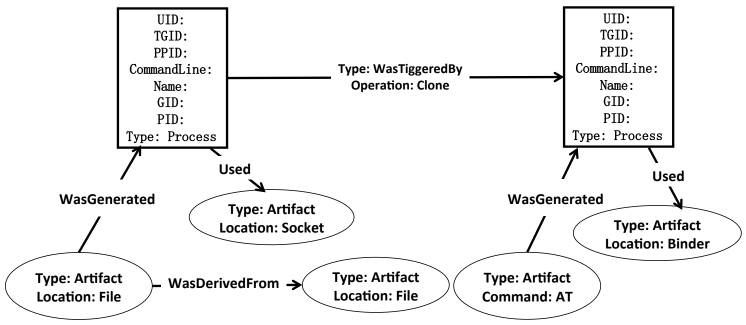

Fig. 2. Apps are represented with rectangular vertices, annotated with the properties of the executing process. Data artifacts, such as files and Binder transactions, are denoted with elliptical vertices. Edges have types define the operations being performed – for example, an artifact is related to a process with a *WasDerivedFrom* edge when it has been written to. In general, the types conform to the Open Provenance Model.

4. Graph Reporter generates a provenance graph using Graphviz [2] and based on the low-level provenance relationships established by the ProvEst Daemon. Specific patterns can be further extracted from the graph by using our graph-based query service, which is implemented by Neo4j [4], an open-source graph-based database tool. This component uses SPADE's *Graphviz Reporter* to replay provenance records, sending them through SPADE's *Kernel*, and to its *Neo4j Storage*.

5. Behavior Identifier detects sensitive behaviors by using the provenance graphs output by the Graph Reporter. Intuitively, we abstract each sensitive behavior into a provenance graph pattern, according to the internal working logic in the Android platform to perform that behavior. We then identify an app's behaviors by mapping its provenance graph with these provenance graph patterns. Next, we elaborate on a few exemplar sensitive provenance patterns.

3.2 Exemplar Sensitive Provenance Patterns

We describe motivating examples, illustrated with figures that use a previously described [26] provenance data model.

Pattern 1: ***Send SMS, Receive SMS and Phone Call.*** Figure 3 illustrates the working logic of an app on the Android platform when sending an SMS, receiving an SMS, and making a phone call. When an app attempts to perform one of these three behaviors, it will first communicate with a process from the Telephony Manager Application Framework. The Telephony Manager will call the Radio Interface Layer (RIL) daemon in the Android's using sockets for communication. RIL is radio-agnostic and provides an abstraction layer between the Android Telephony Manager and the hardware. Once it receives communications from Android's Telephony Manager, the RIL daemon dynamically loads the Vendor RIL Library to dispatch the communications to the Vendor RIL. The radio-specific Vendor RIL processes communicate with radio hardware by using AT commands. The AT commands are used to control mobile modems in order to perform the specified functions. For example, the AT commands for sending an SMS, receiving an SMS and making phone calls are "AT+CMGS", "AT+CNMI", and "ATD+CLCC", respectively.

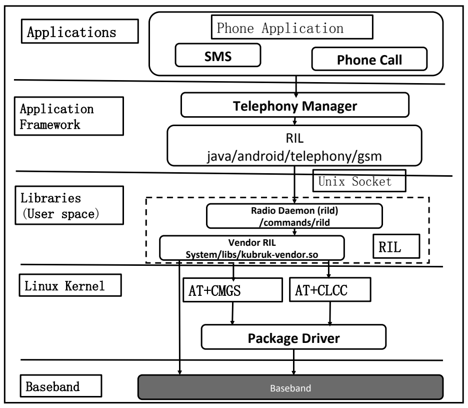

Fig. 3. Working logic of sending an SMS, receiving an SMS, and making a phone call.

By exploiting an understanding of this functionality, the provenance patterns of these behaviors can be abstracted as Figure 4. From this figure, we can see that for each sensitive behavior, there is a provenance path from the app process to the final AT command with different command parameters.

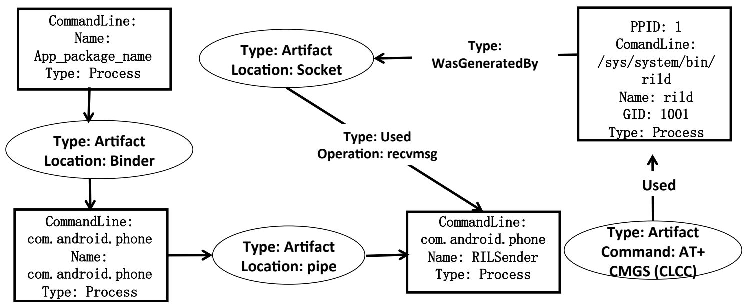
CommandLine: Name: App_package_name Type: Process
Type: Artifact Location: Binder
CommandLine: com.android.phone Name: com.android.phone Type: Process
Type: Artifact Location: pipe
CommandLine: com.android.phone Name: RILSender Type: Process
Type: Artifact Location: Socket
Type: WasGeneratedBy
Type: Used Operation: recvmsg
PPID: 1 ComandLine: /sys/system/bin/rild Name: rild GID: 1001 Type: Process
Used
Type: Artifact Command: AT+ CMGS (CLCC)

Fig. 4. Provenance pattern for sending an SMS, receiving an SMS, and making a phone call.

Pattern 2: ***Read Geolocation.*** Figure 5 illustrates the system logic in the Android system that runs when an app gets the current location. Once an app attempts to read the geographic location, it will interact with the Location Manager Service, which will further request the location from the GpsLocationProvider. From this logic we can abstract the app's provenance pattern as Figure 6, which has a path from the process vertex to the GpsLocationProvider.

Pattern 3: ***Read SMSDB and Write SMSDB.*** The Android system workflow dictates that once an app reads or writes the SMS database (i.e., the content provider of SMS inbox), it will first interact with the TelephonyManager, and then read and write in the "/data/data/com.android.providers.telephony/database/" directory, to the "mmssms.db" file, in particular. The provenance pattern that results is illustrated in Figure 7.

Pattern 4: ***Read Contact, Net, and Rooting.*** On Android, the local Contacts resource is uniquely managed by the Acore process[2]. An app must interact with this process to

[2] The process is identified as "com.android.acore".

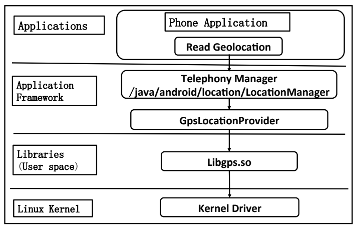

Fig. 5. Working logic of reading geolocation.

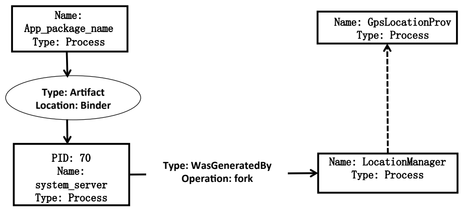
Name: App_package_name Type: Process
Type: Artifact Location: Binder
PID: 70 Name: system_server Type: Process
Type: WasGeneratedBy Operation: fork
Name: LocationManager Type: Process
Name: GpsLocationProv Type: Process

Fig. 6. Provenance pattern for reading geolocation.

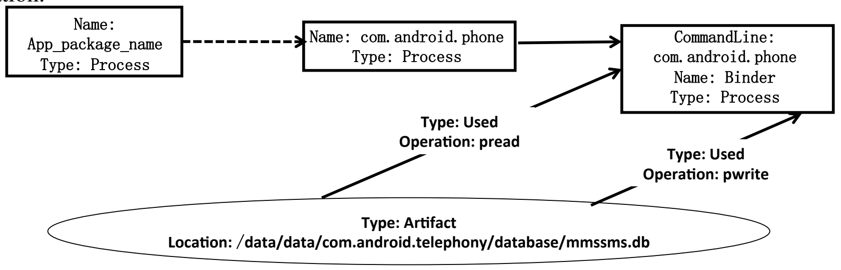
Name: App_package_name Type: Process
Name: com.android.phone Type: Process
CommandLine: com.android.phone Name: Binder Type: Process
Type: Used Operation: pread
Type: Used Operation: pwrite
Type: Artifact Location: /data/data/com.android.telephony/database/mmssms.db

Fig. 7. Provenance pattern for reading from and writing to the SMS database.

read the contact list. If an app reads the contact, there is a path from the process of the app to the Acore process. Network usage can be identified by analyzing whether the process (or its descendants) makes system calls related to network sockets. Rooting behavior can be identified by analyzing whether the process (or a descendant) invokes the exeve("/system/bin/su") system call to attain root privilege. Since our data provenance graph will also record the UID of the process, if the app successfully roots the phone, this behavior can be further identified by checking the change in UID from a non-zero value to zero.

After generating these provenance patterns, we can vet app behavior by matching these patterns in apps' provenance graphs as they are generated at runtime. Note that these provenance patterns are uniquely defined according to the working mechanism of the Android system, from the top layer to the bottom layer, and are more likely to remain unchanged than the source code is. Thus, our approach is more general than other approaches which rely on hooking specific APIs whose functions may be changed later. Also, since the patterns cover all the layers, our approach can identify those behaviors that are implemented by using both the Android SDK and NDK, as long as they follow the same workflow.

4 System Evaluation

Our prototype implementation of Dagger is capable of running on both Android phones and emulators. We evaluated the prototype implementation by running the app in a customized Android emulator and using it to extract provenance graphs with pre-settings of SMS inbox, contact list and geolocation information. Before each run, we restored the image to a clean snapshot to mitigate interference from other apps.

We evaluated the effectiveness of Dagger from the following three perspectives: (i) vetting real-world malware case studies, (ii) vetting Android Genome Project malware, and (iii) vetting official market (Google Play) apps. Then, we evaluated the efficiency of Dagger by using a popular benchmark app to measure the performance overhead of Dagger including CPU overhead, memory overhead, I/O overhead and processing time.

4.1 Effectiveness Case Study on Representative Malware Families

To evaluate the effectiveness and demonstrate its unique advantages, we applied Dagger to vet sensitive behaviors on three representative real-world Android malware families: Gamex, Gone60 and Zsone.

Gamex: Code Encryption. Gamex, one of the most evasive Android malware, uses complex code obfuscation techniques. In an attempt to slow down discovery and detection, Gamex [5] uses encryption (byte XOR with 0x12) to hide a package in a fake image file named "assets/logos.png". When the malware is activated, it uses a decryption function to decrypt the file, and launch sensitive functions. Thus, due to the encryption, the static analysis will only find the paths that lead to the shell code execution function, instead of knowing specific malicious behaviors. Upon using Dagger to vet Gamex samples (MD5: 50836808a5fe7febb6ce8b2109d6c93a), we find shell code execution as well as hidden sensitive behaviors, including attempts to read contact list information and sensitive network communications, such as exfiltration of IMSI/IMEI numbers and malicious software downloads.

Gone60: Privacy Leakage. Gone60 steals private user information such as SMS messages, contact lists, recent call histories and browser-cached URLs by using the standard query API on the content providers of SMS inbox and browser. The app can access these content providers, which work as databases, by setting specific local URLs as the parameters. However, such parameters (i.e., strings) are easier for malware authors to obfuscate than Android framework APIs (e.g., by using complex string operations). Thus, simple approaches based on static analysis may fail to detect such malware. Upon using Dagger to vet a sample of Gone60 (MD5: 859cc9082b8475fe6102cd03d1df10e5), we successfully identified many sensitive behaviors exhibited by this malware, including reading of SMSDB and contact lists, as well as sensitive network communications. Moreover, since Dagger recognizes the access of content providers by checking the read operation of the file system instead of statically analyzing the parameters in the query function, it is more robust against string-obfuscating malware.

Zsone: SMS Service Usage. Dagger can also be used to vet the malicious behavior of blocking SMS by checking for the absence of a certain pattern in a specific event (i.e., receiving an SMS message but not writing to SMSDB). We applied Dagger to an exemplar Zsone malware sample (MD5: c0e6ba0e1b757e3c506a02282ffc5b4), which can both send and block SMS messages. In this experiment, we used Dagger to send the same pre-customized SMS to the phone in two situations: running without and alongside the malware sample. We found that while both receive the SMS message (observing the AT command "AT+CNMA=1"), the provenance graph in the first scenario includes the behavior pattern of writing SMSDB, while the second scenario does not. This validates that Dagger can be used to identify the blocking SMS behavioral pattern.

4.2 Measuring Effectiveness Using a Large App Corpus

We further evaluated the effectiveness of Dagger on a corpus of 1,260 real-world malware samples collected from the Genome Project [55], and another corpus of 1,000 apps

Table 3. Sensitive behaviors in different malware families identified by Dagger.

Family	Send SMS	Receive SMS	Read SMSDB	Write SMSDB	Read Contact	Read GeoLocation	Execute Shell	Net
ADRD	0	0	0	0	0	0	0	9
AnserverBot	0	7	3	3	3	2	61	78
Asroot	0	0	0	0	0	0	0	1
BaseBridge	0	1	1	1	1	0	16	37
BeanBot	0	0	0	0	0	0	0	1
Bgserv	0	1	0	0	0	1	0	3
CoinPirate	0	0	0	0	0	0	0	2
CruseWin	0	0	0	0	0	0	0	0
DogWars	0	0	0	0	0	0	0	0
DroidCoupon	0	0	0	0	0	0	0	0
DroidDeluxe	0	0	0	0	0	0	0	0
DroidDream	0	0	0	0	0	0	1	5
DroidDreamLight	0	0	0	0	3	1	1	9
DroidKungFu1	0	2	2	1	1	0	2	13
DroidKungFu2	0	1	1	0	1	0	5	8
DroidKungFu3	0	0	8	7	8	0	26	111
DroidKungFu4	0	4	5	3	4	2	7	47
DroidKungFuSapp	0	0	0	0	0	0	0	0
DroidKungFuUpdate	0	0	0	0	0	0	0	1
Endofday	0	0	0	0	0	0	0	0
FakeNetflix	0	0	0	0	0	0	0	0
FakePlayer	0	0	0	0	0	0	0	0
GamblerSMS	0	0	0	0	0	0	0	0
Geinimi	0	4	3	2	3	3	1	20
GGTracker	0	0	0	0	0	0	0	1
GingerMaster	0	0	0	0	0	0	0	2
GoldDream	0	1	1	1	1	2	0	20
Gone60	0	0	1	0	3	0	0	5
GPSSMSSpy	0	0	0	0	0	0	0	0
HippoSMS	0	0	0	0	0	0	0	3
Jifake	0	0	0	0	0	0	0	0
jSMSHider	0	0	0	0	0	0	7	4
KMin	1	1	0	0	13	0	0	17
LoveTrap	0	0	0	0	0	0	0	1
NickyBot	0	0	0	0	0	0	0	0
NickySpy	0	0	0	0	0	0	0	1
Pjapps	0	0	0	0	1	0	0	28
Plankton	0	0	0	0	0	0	0	5
RogueLemon	0	0	0	0	0	0	0	0
RogueSPPush	0	0	0	0	0	0	0	9
SMSReplicator	0	0	0	0	0	0	0	0
SndApps	0	1	0	0	0	0	0	0
Spitmo	0	0	0	0	0	0	0	0
Tapsnake	0	0	0	0	0	0	0	0
Walkinwat	0	0	0	0	0	0	0	1
YZHC	0	1	1	1	1	0	0	0
zHash	0	0	0	0	0	0	0	0
Zitmo	0	0	0	0	0	0	0	0
Zsone	2	1	0	0	1	0	0	12

that were randomly selected from 18,527 official market (Google Play) apps. For each app, to increase the code path execution coverage, we added 500 random UI events by using MonkeyRunner.

Table 3 shows the number of apps and corresponding malware families that perform each behaviors. Since there is no easy way to obtain complete ground truth about the sensitive behaviors found in these specific malware samples, we simply show the absolute number instead of the false positive/negative rate.

As summarized in the table, we find that Dagger can find sensitive behaviors from all malware families. Moreover, Dagger can successfully find all three types of sensitive behaviors (Send SMS, Net and Execute Shell) in the malware families that were reported by a prior measurement study on the same malware corpus [54].

Table 4. Dagger analysis summary for 112 randomly selected malware samples.

Send SMS	Block SMS	Read SMSDB	Write SMSDB	Read Contact	Read Location	Execute Shell	Net
9	19	1	0	2	3	17	60

After using Dagger to vet 1000 official apps from Google Play, we found the following. (i) One app reads SMSDB, which is a TV Channel client embedded with multiple advertisements, and reads users' SMS messages. (ii) Four apps have executed external/shell commands. After submitting them to VirusTotal, one app was recognized as malware belonging to Plankton. This malware dynamically downloads additional code from external server and executes it. The malware then executes shell commands (e.g., "/system/bin/cat /proc/cpuinfo") to get the system information. Two apps were recognized by VirusTotal as abusive adware. Both of them executed the shell commands to use the Logcat to obtain the system runtime log information. The fourth app was not recognized as malware by VirusTotal. However, it attempted to obtain root privilege by executing "su", which is recognized as a sensitive behavior by Dagger. (iii) Seven apps read users' geolocation information. More specifically, three apps use such geolocation information for the usage of maps; two apps are used for car rental guides; one app is for local shopping and another one is a photo editor app that can be used by users to share photos with geolocation information to their friends. Our findings confirm that our system has a low false positive rate, i.e., only a small number ($< 2\%$) of official apps are identified as performing sensitive behaviors, and the majority of these are related to known malware/adware families.

Analysis of False Positives and Negatives. Since it is very challenging to obtain a perfect ground truth for the Android malware dataset (i.e., knowing the exact sensitive behaviors of each malware sample we collect), we further evaluated the accuracy of Dagger by comparing it with other existing systems, instead of claiming accurate value of the false positive and false negative rate. More specifically, we ran Dagger on 112 malware samples, which were randomly selected from the Genome malware dataset. The specific number of malware samples that perform each type of sensitive behaviors can be seen in Table 4.

To measure possible false positives, we examined those behaviors identified by Dagger, which are not reported by [54]. [54] reports possible sensitive behaviors of the malware samples in each family by statically extracting programming paths that may execute sensitive behaviors. We found that only two apps, Asroot and DroidKungFuUpdate, access the Internet but are not reported by [54]. We manually examined these two apps, and found that they do indeed access the Internet to load advertisements when they are activated.

To measure possible false negatives, we compared our system with CopperDroid. (Since CopperDroid is not open-source, we obtained its results by submitting apps to its public website.) Since CopperDroid instruments QEMU to intercept all instructions that are executed in the Android emulator, it can report most sensitive behaviors. Compared with CopperDroid, we find that Dagger misses one network behavior and 2 reading contact behaviors due to the failure of triggering the execution paths. We also tested

Table 5. Dagger's performance overhead as measured using the AnTuTu benchmark app.

Metric	OffScore	OnScore	Overhead
CPU score	7,199	6,522	9.40%
RAM score	1,213	1,092	9.98%

these malware samples on TaintDroid, which only detects that 1 app reads location information, and 23 apps access the Internet.

4.3 Measuring System Performance Overhead

To evaluate the efficiency of Dagger, we tested the performance overhead of Dagger by using AnTuTu (v 3.0.3) [1]. AnTuTu is a popular Android benchmark app developed to test the performance of Android devices. We are mostly interested in the major performance benchmark metrics such as CPU score and RAM score. CPU score represents the computation ability of the current CPU status; a higher score implies the CPU has more free computation ability. RAM score reflects the real processing ability of RAM; a higher score implies more free space in RAM.

Table 5 shows the scores of each benchmark metric while turning Dagger off/on (denoted as $OffScore$ and $OnScore$, respectively). In this table, the overhead of each metric is calculated as: $Overhead = (OffScore - OnScore)/OffScore$.

From this table, we can find that the overheads of CPU and RAM after turning Dagger on are acceptable, which are less than 10%. This clearly indicates that Dagger is a lightweight vetting approach that consumes a very small number of resources, an advantage makes it attractive for practical use.

Besides the above metrics, we also measure the time overhead generated by Dagger. The time spent running the Antutu benchmark app on an unmodified system was 1.89 seconds. When Dagger was used, Antutu took 3.07 seconds to run. From this we can see that the time overhead is reasonably low: 62.43%. It is worth noting that in another representative approach based on system call tracing, DroidScope [50], the slowdown was around 11 to 34 times (with taint-tracking enabled). This experiment clearly demonstrates that Dagger is a very lightweight tool.

In comparison with existing work, we can find that though it requires neither instrumentation of the system nor modification the OS, Dagger can achieve significantly high accuracy with appreciably lower performance overhead. Note that queries are performed offline using an indexed graph database. This ensures that complex graph queries can scale to large data sets, limited only by the underlying database Neo4j (that is used in production environments).

5 Related Work

We broadly classify related work into four major categories: detection of Android malware, security analysis and defense of the Android platform, and analysis of behaviors in Android apps.

Detection of Android Malware: An extensive body of systems has been developed to detect Android malware by monitoring system calls [15, 39, 42, 43, 50, 46, 30, 27], analyzing the usage of Android permissions [24, 11, 23, 38], analyzing the usage of Framework APIs [13, 55, 47, 56, 17, 52], and extracting information from the *sysfs* pseudofilesystem [12]. The design of these detection systems requires deep domain knowledge about Android system and the development of Android malware. Most of them also require effective and robust disassemblers to disassemble the target apps into Dalvik bytecode. These static approaches achieve limited effectiveness when detecting more evasive malware that is implemented with complex obfuscation techniques (e.g, encrypting the source, inserting noisy code, using Java reflection) and NDK. In contrast, Dagger does not require robust (or any) disassembly or deobfuscation technology.

Android Security Analyses: A few existing studies focus on analyzing the security mechanism of the Android platform and its applications. Stowaway [24] is designed to find those over-privileged apps. SmartDroid [52] finds UI triggers that result in privacy leakage. DroidChameleon [40] demonstrates the vulnerability of existing android anti-malware tools. Other related studies include attempts to detect component-hijacking vulnerabilities [32], inter-app communication vulnerabilities [19], and capability leaks [25, 16]. In contrast to these analyses which focus on the leakage of security privileges, we focus on the leakage of sensitive data.

Android Platform Defenses: A variety of techniques have been developed to extend the security policies that can be supported by Android. Quire [21] is designed to prevent confused deputy attacks. Bugiel [14] *et al.* proposed a framework to prevent collusion attacks with pre-defined security policies. Saint [37], Porscha [36], and CRepE [20] were developed to isolate apps by designing more fine-grained access control policies. AppFence [28] prevents privacy leaks by either feeding fake data or blocking the leakage path. Checking at install time, Apex [35] allows for the selection of granted permissions, and Kirin [23] performs lightweight certification of applications. Paranoid Android [39], L4Android [31] and Cells [10] use virtualization as an isolation mechanism to manage the risk of running malicious applications on Android. A prototype implementation of SELinux on an Android [3] device[44] provides mandatory access control. Aurasium [49] protects the system by enforcing practical policies. Previous work that relies on extensive modifications to the operating system that is brittle in the face of evolving codebases. In contrast, we are able to support sensitive behavior monitoring without modifying apps, the Dalvik virtual machine, or the Linux kernel.

Behavioral Analysis of Android Apps: Besides detecting malware and enhancing security mechanism of the Android platform, a few studies focus on analyzing sensitive/malicious behaviors in Android apps.

Dalvik Monitoring. As summarized in Table 6, TaintDroid [22] is one of the first few systems that are designed to track possible sensitive leak from Android apps. VetDroid [51] vets sensitive behaviors by checking the permission usage at runtime. It requires modification to both the Android Dalvik virtual machine to intercept API invocations, and the Android framework to monitor invocations of app callbacks. Since these two approaches achieve the goal by mainly monitoring the execution of the Java instructions in the Dalvik, they are not effective when applied to finding sensitive behaviors that are implemented by Native Code. In addition, depending on whether hooks in the source of Android OS are used, these approaches are limited to the periodical change of the Android OS.

Virtual Machine Instrumentation (VMI). DroidScope [50] is designed to vet behaviors in Android apps by reconstructing both OS-level and Java-level semantics.

Table 6. Comparison of Dagger with alternative sensitive behavioral vetting approaches.

	TaintDroid	**VetDroid**	**DroidScope**	**NDroid**	**CopperDroid**	**Dagger**
Technique	Taint Analysis	Taint Analysis & Permission Analysis	Taint Analysis & VMI	Taint Analysis & VMI	VMI & Monitoring System Calls and Binder	Data Provenance Analyiss & Monitoring System Calls, Binder and Process
Runtime Modifications	Modifying Android OS	Modifying Android OS	Instrumenting QEMU	Instrumenting QEMU	Instrumenting QEMU	None
Native Code Support	No	No	Yes	Yes	Yes	Yes
Overhead	Medium	Medium	High	High	High	Medium

NDroid [18] is a supplementary of TaintDroid, which is aware of the JNI semantic to track the data flow in the native code. CopperDroid [41] reconstructs malware behaviors by monitoring the system calls and the binder. Since these approaches rely on instrumenting the Android emulator, which typically incurs high overheads, especially when taint tracking is enabled, their direct application to analysis of a large scale of Android apps is inefficient. In addition, similar to the emulation-resistant desktop malware, Android malware can evade such approaches by staying dormant or simply crashing themselves, once the malware identifies that it is running within an emulated environment [45, 29].

Motivated by the limitations of these approaches, Dagger is designed as a complementary and lightweight system to effectively and efficiently vet sensitive behaviors in Android apps. Dagger fills the semantic gap by representing Android apps' interactions with the system in a data provenance graph, and further matching the provenance graph with a library of sensitive provenance patterns.

6 Limitations and Future Work

Since Dagger's approach relies on the analysis of the inner working flow of the Android system to vet sensitive behaviors, it has to be updated if the workflow of the Android system is significantly altered. However, due to the practical implications of such design, e.g., changes on a huge system that is being used by millions of devices, we believe that such significant changes are likely to be infrequent.

In the current design of Dagger, failed system call invocations are not captured in its data provenance graph. Such failure information might be useful in capturing certain sensitive behaviors that are missed by the current system. In the future work, we plan to improve Dagger by incorporating these into our analyses.

A common limitation of dynamic analysis techniques is that an exhaustive search of the space of all possible behavior of a target piece of code requires an untenable amount of testing. Consequently, techniques such as "fuzz testing" use random inputs or other methods for selecting a sparse subset of the test space. While Dagger is able to trigger security-sensitive behavior that matches particular provenance patterns, it is not exhaustive.

Finally, Dagger requires some manual effort to fine-tune the extracted provenance patterns that are currently used in vetting sensitive behaviors. We plan to extend our system with a learning-based approach to automatically mine graph patterns from apps that share similar sensitive behaviors. Furthermore, Dagger's provenance pattern library may be easily extended with additional verified patterns.

7 Conclusion

This paper presents Dagger, a novel and lightweight approach to dynamically vet sensitive behaviors in Android apps without system instrumentation or OS modification. Dagger achieves its goals by collecting three types of lower-level information and summarizing the app's system interactions through a lightweight provenance graph. In addition, Dagger contains a library of sensitive provenance patterns that can be used to automatically identify sensitive behaviors embedded in Android apps. Our evaluation demonstrates that Dagger is able to quickly and effectively isolate sensitive behaviors across a large corpus of (benign and malicious) real-world apps, with significantly lower performance overhead than prior studies.

8 Acknowledgements

This work was partially funded by the U.S. Department of Homeland Security (DHS) Science and Technology Directorate under contract HSHQDC-10-C-00144, National Science Foundation (NSF) under Grants IIS-0905518 and IIS-1116414, and Air Force Office of Scientific Research (AFOSR) under Grant FA9550-13-1-0077. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official views of DHS, NSF, AFOSR, or the U.S. government.

References

[1] Antutu benchmark. `https://play.google.com/store/apps/details?id=com.antutu.ABenchMark&hl=en`.

[2] Graphviz - graph visualization software. `http://www.graphviz.org/`.

[3] National security agency. security-enhanced linux. `http://www.nsa.gov/research/selinux`.

[4] Neo4j. `http://www.neo4j.org/?gclid=CIXUs_D-xb0CFQaBfgodIAMARw`.

[5] Obfuscating embedded malware on android. `http://www.symantec.com/connect/blogs/obfuscating-embedded-malware-android`.

[6] The proc filesystem. `http://en.wikipedia.org/wiki/Procfs`.

[7] Strace - trace system calls and signals. `http://linux.die.net/man/1/strace`.

[8] Sysfs. `http://en.wikipedia.org/wiki/Sysfs`.

[9] Ui/application exerciser monkey. `http://developer.android.com/tools/help/monkey.html`.

[10] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: A virtual mobile smartphone architecture. In *Proceedings of 23rd SOSP*, 2011.

[11] 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 CCS*, 2012.

[12] Michael Backes, Sven Bugiel, and Sebastian Gerling. Scippa: System-centric IPC provenance on Android. In *30th Annual Computer Security Applications Conference*, 2014.

[13] A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral detection of malware on mobile handsets. In *Proceedings of the 6th MobiSys*, 2008.

[14] 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 NDSS*, 2012.

[15] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for android. In *Proceedings of the 1st Workshop on CCSSPSM*, 2011.

[16] 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*, 2012.

[17] 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 NDSS*, 2013.

[18] Q. Chenxiong, L. Xiapu, S. Yuru, and C. Alvin. Ndroid: On tracking information flows through jni in android applications. In *Proceedings of the 44th DSN*, 2014.

[19] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th MobiSys*, 2011.

[20] M. Conti, V. T. N. Nguyen, and B. Crispo. Crepe: Context- related policy enforcement for android. In *Proceedings of the 13th ISC*, 2010.

[21] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proceedings of the USENIX Security*, 2011.

[22] 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 OSDI*, 2010.

[23] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th CCS*, 2009.

[24] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystied. In *Proceedings of the 18th CCS*, 2011.

[25] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the USENIX Security*, 2011.

[26] Ashish Gehani and Dawood Tariq. SPADE: Support for provenance auditing in distributed environments. In *ACM/IFIP/USENIX 13th International Middleware Conference*, 2012.

[27] You Joung Ham, Daeyeol Moon, Hyung-Woo Lee, Jae Deok Lim, and Jeong Nyeo Kim. Android mobile application system call event pattern analysis for determination of malicious attack. *International Journal of Security and Its Applications*, 8(1), 2014.

[28] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Are not the Droids You are Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th CCS*, 2011.

[29] Y. Jing, Z. Zhao, G. Ahn, and H. Hu. Morpheus: Automatically Generating Heuristics to Detect Android Emulators. In *Proceedings of ACSAC*, 2014.

[30] Mohammad Karami, Mohamed Elsabagh, Parnian Najafiborazjani, and Angelos Stavrou. Behavioral analysis of Android applications using automated instrumentation. In *7th International Conference on Software Security and Reliability Companion*, 2013.

[31] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4android: A generic operating system frame- work for secure smartphones. In *Proceedings of the 1st Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2011.

[32] 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 CCS*, 2012.

[33] S. Mengtao and T. Gang. Nativeguard : Protecting android applications from third-party native libraries. In *Proceedings of ACM conference on Security and privacy in wireless & mobile networks*, 2014.

[34] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. Van. The open provenance model core specification (v1.1). In *Future Generation Computer Systems*, 2010.

[35] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on ICCS*, 2010.

[36] M. Ongtang, K. Butler, and P. McDaniel. Porscha: Policy oriented secure content handling in android. In *Proceedings of the 26th ACSAC*, 2010.

[37] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. In *Proceedings of the 25th ACSAC*, 2009.

[38] 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 CCS*, 2012.

[39] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: Versatile protection for smartphones. In *Proceedings of the 26th ACSAC*, 2010.

[40] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ICCS*, 2013.

[41] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of the EUROSEC*, 2013.

[42] 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 *ICC Communication and Information Systems Security Symposium*, 2009.

[43] 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*, 2008.

[44] A. Shabtai, Y. Fledel, and Y. Elovici. Securing android- powered mobile devices using selinux. In *Proceedings of 31th IEEE Security and Privacy.*, 2010.

[45] T. and N. Christin. Evading Android Runtime Analysis via Sandbox Detection. In *Proceedings of ASIACCS*, 2014.

[46] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Profiledroid: Multi-layer profiling of Android applications. In *18th Annual International Conference on Mobile Computing and Networking*, 2012.

[47] 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*, 2012.

[48] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In *Proc. of the CCS*, 2013.

[49] R. Xu, H. Saidi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the USENIX Security Symposium*, 2012.

[50] 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*, 2012.

[51] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proc. of CCS*, 2013.

[52] 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 ACM workshop on security and privacy in smartphones and mobile devices*, 2012.

[53] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *CODASPY*, 2012.

[54] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[55] 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 NDSS*, 2012.

[56] Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th MobiSys*, 2012.