Detect repackaged Android application based on HTTP traffic similarity

Article *in* Security and Communication Networks · September 2015

DOI: 10.1002/sec.1170

4 authors, including:

Dafang Zhang
Hunan University
111 PUBLICATIONS **304** CITATIONS
SEE PROFILE

Xin Su
Hunan University
9 PUBLICATIONS **14** CITATIONS
SEE PROFILE

SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks 2015; **8**:4273–4288
Published online 15 September 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/sec.1341

RESEARCH ARTICLE

AndroGenerator: An automated and configurable android app network traffic generation system

Xin Su[1], Dafang Zhang[1]*, Wenjia Li[2] and Xiaofei Wang[1]

[1] College of Computer Science and Electronics Engineering, Hunan University, Changsha, China
[2] Department of Computer Sciences, New York Institute of Technology New York, NY 10023, U.S.A.

ABSTRACT

With the rapid growth in the popularity of Android smartphones, a large number of Android applications (or apps) have emerged in both official and alternative Android markets. It is important for network operators and security analysts to understand the network traffic generated by new Android apps for the purposes of network management, app traffic analysis, and malware detection. However, it is time-consuming and tedious to manually install and run Android apps to generate network traffic. Moreover, existing synthetic network traffic generators are unable to generate network traffic that can accurately reflect the network behaviors of Android apps. In this paper, we propose and implement AndroGenerator, an automated Android network traffic generation system, to generate various types of network traffic that can be produced by Android apps. Our system reproduces the network traffic based on the traffic characteristics extracted from traffic traces captured by running a large number of Android applications from several popular Android markets, such as Google Play. The system first generates network traffic through automated execution of Android applications. Then, the system is also able to extract network characteristics from the captured traffic traces and store the extracted results into a database for benchmarking purposes. Finally, AndroGenerator reproduces Android app traffic based via simulating network characteristics of captured traffic traces. In the experiments, we evaluate our system with real-world mobile traffic, and the experiment results show that AndroGenerator can reproduce Android app network traffic accurately. Copyright © 2015 John Wiley & Sons, Ltd.

KEYWORDS
Android application; traffic generator; automatic execution

***Correspondence**
Dafang Zhang, College of Information Science and Engineering, Hunan University, Changsha, China.
E-mail: dfzhang@hnu.edu.cn

1. INTRODUCTION

1.1. Motivation

In recent years, we have witnessed a rapid growth in the popularity of mobile devices, such as smartphones and tablet computers. In addition, there have been dramatic changes to the way that users behave, interact, and utilize the network: more and more users are surfing the Internet via mobile devices such as smartphones and tablets by using various kinds of mobile apps. With the explosive growth in the popularity of mobile apps, the mobile traffic has experienced a 5000% increase in the past 3 years [1]. However, most of these applications are created by unknown developers, and hence, they may not be executed in the user's best interests, leading to malware [2–5] and frequent leakage of privacy and other sensitive information such as phone identifiers and location information via network communication [6,7].

These scenarios have made the network level analysis of mobile apps become more important in network management, security, and traffic analysis.For example, there is an interesting trend that the proportion of personal devices being used in the enterprise networks is growing rapidly. Hence, it is crucial for network operators to have a clear view regarding the mobile apps that are running in their network. However, some researchers have found it hard to obtain the real-world Android apps traffic trace, and manual collection of Android app traffic is not scalable because the number of Android app is increasing rapidly.

1.2. Limitation of prior art

There have been an increasing number of research efforts in recent years that analyze mobile app network traffic [8,9], and mobile traffic can be obtained by the following two methods: either collected from cellular carriers or

manually generated. Lever *et al.* [8] used cellular traffic traces to analyze malware at Domain Name Service (DNS) level. This type of traffic traces can be used to analyze traffic that is normally generated by a large number of mobile devices. However, this kind of traffic trace is hard to obtain because they generally contain users' privacy information, such as phone number or credit card number. Therefore, the cellular traffic traces usually need to be anonymized first before they can be used and analyzed. In the paper authored by X.Wei *et al.* [9], the Android apps are manually running on the mobile devices to generate the traffic, and then the traffic will be analyzed. This method can produce very detailed traffic patterns, but it requires a great amount of human labor to run the apps. As the number of apps grows explosively, this is not realistic. To address the limitations of manual execution, Dai *et al.* [1] proposed an automatic approach that executes Android apps to generate traffic and extract network signatures to identify Android apps with two kinds of tests, namely, random test and direct test. This approach uses random test to execute apps that would access the original sever a and direct test to run apps that would not access origin server. However, random test only triggers part of network behaviors from the apps, which makes the generated traffic not able to represent the real-world Android app network traffic. Moreover, Dai *et al.* did not evaluate the efficiency of their approach. Therefore, how to automatically generate and obtain representative traffic traces from mobile apps play a key role in the network-level analysis.

Moreover, there are many other research efforts on generating synthetic traffic to simulate realistic network traffic [10,11]. Some of them focus on generating synthetic network traffic that generally includes Transmission Control Protocol (TCP), User Datagram Protocol (UDP), and HyperText Transfer Protocol (HTTP). Some of them generate traffic based on self configuration by automatically extracting parameters from standard Netflow logs or packet traces. All of these works focus on capacity planning [12], high-speed router design, queue management studies [13], and bandwidth measurement tools [14,15]. The synthesis of Android app network traffic may not focus on these studies. It is well known that mobile apps have a number of different network behaviors [1] when they visit different servers. Thus, it is infeasible for the existing synthetic traffic generators to simulate such a diverse set of network behaviors as well.

1.3. Proposed approach

In this paper, we address these limitations by designing AndroGenerator, a system that can generate realistic and comprehensive network traffic for Android apps in an automatic and customized manner. The key to reproduce the typical network traffic is to ensure that the original traffic traces that we extract must be collected from a large number and different kinds of Android apps, and the apps are driven along most of their execution paths when the network traffic traces are being generated. To achieve this goal, we first collect a 10K+ real-world Android app dataset that consists of the most popular apps that are frequently downloaded from Google Play, several popular third-party markets, and Android Malware Genome Project [16].

In particular, we present AndroGenerator, which contains three components. The first is Android app execution, which executes apps by utilizing an automated techniques based on two Android testing tools, monkeyrunner [17] and hierarchy viewer [18]. The second component is responsible for extracting traffic features and advertisement (or ad) traffic from the captured traffic traces. The third component is traffic generator, which provides configurable profiles to configure AndroGenerator on how it should generate traffic. As S.Dai *et al.* [1] illustrated, the majority of Android app traffic is carried over HTTP/HTTPS protocol, and the HTTPS protocol is often used only in the authentication phase of the session, and HTTP is used for the rest of the user interaction. In Y. Zhou *et al.* [16], they found that most of Android malware used the HTTP-based web traffic to receive bot commands from their command-and-control servers. Based on these findings, our approach mainly focuses on HTTP traffic generation.

1.4. Key contributions

In summary, the main contributions of this work are listed as the following:

- We design a Android app traffic generation system, AndroGenerator, which is application-based and able to generate live network traffic for Android apps. Compared with the existing manual or synthetic traffic generation approaches, this system can automatically execute Android apps and generate more representative network traffic.
- We extract network characteristics and ad traffic from the original traffic traces and store them in an updated database to establish an Android app network traffic benchmarking for high-level studies.
- We evaluate the automatic execution algorithm of AndroGenerator, and compare the results of activity coverage with three mainstream execution approaches. We also compare the time of path identification between our work and *NetworkProfiler*. Then, we validate AndroGenerator with two real-world mobile traffic traces, which clearly show that our approach can generate mobile traffic that is similar to the traffic that is generated by actual Android apps.

The rest of this paper is organized as follows. Section 2 introduces related works. Section 3 describes the basic design of AndroGenerator. Section 4 discusses the details of AndroGenerator. In Section 5, we evaluate our approach with real world mobile traffic traces. We conclude this paper in Section 6.

DOI: 10.1002/sec

2. RELATED WORK

Android app automatic execution and network traffic analysis are hot research topics that have been investigated for decades. In this section, we briefly present the research efforts in the literature from the following three aspects: Android app automatic execution, synthetic network traffic generation, and Android app traffic analysis.

Android app traffic analysis. There have also been a large number of research efforts in analyzing Android app and advertisement (or ad) library based on network traffic traces that were either obtained from cellular service providers or generated manually. In [19], the authors aimed to understand smartphone usage behavior based on Android app traffic analysis. In [8], the authors analyzed the app network traffic at DNS level to identify malicious mobile traffic. In AdCache [20], the authors characterized ad traffic, and the goal of this work was to reduced the energy and network signalling overhead caused by the ad system. In addition, Profiledroid [9] aimed to build app profiles at multiple levels, including the network level. However, this technique completely relies on manual execution of Android apps by users to generate mobile traffic, which does not scale well for a large number of apps.

Synthetic traffic generation. There are several research efforts that primarily focus on generating Internet traffic. LiTGen is an easy-to-use and tune open-loop traffic generator that statistically models wireless traffic on a per user and application basis [21]. OpenAirInterface traffic generator is a realistic packet-level traffic generator, which also takes into account the intrinsic traffic characteristics of the emerging applications [22]. In [23], the authors proposed an online gaming traffic generator that analyzed the packet size and inter departure time distributions of popular online games for regenerating gaming traffic to reflect user behavior patterns. All of these works focus on generating traffic for various purposes in traditional wired network, such as router design and link capacity measurement, On the contrary, AndroGenerator focuses on generating network traffic for mobile networks, which can then be used for Android app traffic analysis, Android app traffic characterization, and so on.

App automatic execution. Recent research efforts provide app automation, but they have been focused on different applications: automated testing [24–26] and automated privacy and security detection, such as examine information flow for potential privacy leak in apps [27], app security validation [28], and app security testing [29]. Automated testing efforts evaluate their system only on a handful of apps, and many of their User Interface (UI) automation techniques are tuned to those apps. Systems that look for privacy and security violations execute on a large collection of apps, but they only use basic UI automation techniques. PUMA [30] is a programmable UI automation framework for conducting dynamic analyses of mobile apps. Dai *et al.* [1] use an automated tool to generate traffic fingerprints for Android apps identification. However, this work did not mention how to generate traffic and compare the generated network traffic with real-world traffic traces. In contrast to all of these prior research works, AndroGenerator is designed for automatic execution of Android apps, and it can also generate Android app traffic, provide traffic traces for Android app traffic analysis, characterization, and malware detection.

3. OVERVIEW OF ANDROGENERATOR

3.1. Requirements

This section discusses our design goals, requirements, and approach for AndroGenerator. We extract bidirectional characteristics of traffic generated by our target, the Android apps. Before we describe our approach to generate traffic traces, we present our metrics for successful traffic generation: realism, comprehensiveness, and maximum randomness.

Traffic generation for Android apps is generally viewed as a powerful tool that facilitates other related research work, such as malware detection. Thus, the definition of realism for a traffic trace generation mechanism must be considered before we use the generated traffic traces. For instance, generating traces for traffic characteristics study only requires matching trace characteristics such as packet size over relatively coarse time intervals. On the other hand, traffic trace generation for other applications such as Android app identification method or design detection method for malware has much more stringent requirements.

We aim to generate realistic traces for a range of application scenarios, and hence, our goal is to generate traces that accurately reflect the following network characteristics: (i) packet size distribution; (ii) flow characteristics including packet number and length distribution; and (iii) HTTP request and response length distribution.

To make the generated traffic traces comprehensive, Android app traffic trace generation system should extract network characteristics from traffic traces captured by a large number of Android apps. The set of Android apps that we use to generate traffic traces should include several kinds of Android apps, such as apps from Google Play, apps from third-party Android markets, and Android malware. The traffic traces also should cover the network behavior of Android apps as much as possible. The manual execution of apps is very time-consuming and lacks scalability to handle a large number of apps. To solve these problems, AndroGenerator not only executes apps automatically but it also can identify all action paths of apps for execution purpose.

By saying maximum randomness, we mean that the Android app traffic generation system should be able to generate traffic traces that are similar to the traffic generated by Android users executing Android apps. Thus, such traffic traces should vary across individual apps or users while still following the appropriate underlying

distributions. This requirement eliminates the trivial solution of simply replaying the exact same apps in the exact same order. However, quantifying the extent to which we are successful in generating comprehensive traffic is beyond the scope of this paper.

3.2. Overview

The goal of this work is to design a system that is capable of generating live Android app network traffic that should accurately reflect a wide range of both current and future scenarios. Such a system would be valuable in a variety of settings, including the following: traffic characterization study [20,31] and malware detection methods [8]. We define traffic generation to result in a distribution of packet and flow length with realistic values. The traffic should appropriately map to flow and packet-size distributions, for example, length distributions. Our hypothesis is that realistic and comprehensive traffic generation system must be informed by accurate models of the following: (i) the Android app execution initiating communications with servers; (ii) the duration of executing each app; and (iii) composition of different kinds of apps (e.g., benign app and malware app).

We first describe our methodology for trace generation, assuming that we possess perfect knowledge of these three models. In the following sections, we describe how to extract values for all of these characteristics based on packet and flow traces generated from AndroGenerator. In the evaluation section (Section 4), we evaluate the similarity between the traffic traces generated by our system and the real-world mobile traffic.

To generate a traffic trace, we first execute a bulk of Android apps in several controlled emulation environments running commodity operating systems or real Android smart phones. Then we simply record the packets and flows that can be used to form the traffic trace. The characteristics of individual flows generated by apps are drawn from our models of individual app and user behavior.

Assuming that we are able to accurately generate traffic traces, which would realistically match the characteristics of the real world trace, this same emulation environment allows us to extrapolate to scenarios that are different from the real-world trace. For instance, we could simply modify application characteristics or the formation of the app set to change the generated trace.

A traffic trace generated by different configurable rules generally will match the essential characteristics of a real-world trace or empirical distributions for packet and flows. These traces can serve as input to higher level studies, for example, flow categorization algorithms or anomaly app detection. It will also be interesting to explore the possible application for traffic trace generation to other Android application studies. For instance, Android app traffic measurement tools may be studied when a variety of smart phone users generate traffic traces at a given cellular network.

4. SYSTEM DESIGN

In this section, we present the details of how to design AndroGenerator. The design objectives of this system are (i) to generate an Android app network traffic in an automatic and configurable manner and (ii) the generated traffic trace should be realistic and representative of Android app traffic or even more generalized mobile app traffic.

4.1. Overall architecture

We aim to design a system for the automatic and configurable execution of Android apps for traffic generation purpose. We now describe the architecture of our approach in Figure 1.

In Figure 1, we find that our approach consists of three parts: first is app execution, which automatically executes Android apps on several real smartphones or Android emulators and outputs the captured network traffic traces that are generated during execution. This part is also

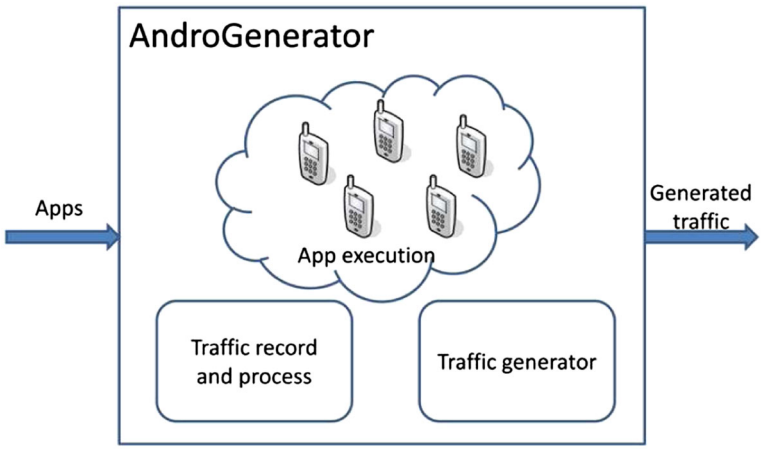


Figure 1. Architecture of AndroGenerator.

configurable by choosing different rules, for example, composition of Android apps, execution duration, and execution behaviors. The second part is responsible for receiving the captured traffic traces as input, and then extracting packets and flows characteristics. It is known that ad library plays an important role in Android ecosystem and that all ad libraries are embedded in the Android apps. Therefore, the ad traffic should be blended with the generated traffic, and we can also precisely identify the ad traffic when we look up the configuration file that contains information about the 100 most popular ad libraries [32]. The third part is traffic generator, which reads configurable files and generates a traffic for high-level studies.

4.2. Implementation

In this section, we first describe the Android app data set that is used for original traffic traces generation for AndroGenerator. Then, we follow with a description of the implementation of the traffic generation component of AndroGenerator.

4.2.1. App execution.

The goal of the app execution component is to execute apps automatically, which is the base of AndroGenerator. We design this component based on two Android testing tools, monkeyrunner [17] and hierarchy viewer [18]. This component is responsible for executing the app automatically and triggering apps to generate traffic, which consists of four modules as follows: (i) event generation; (ii) path identification; (iii) auto execution; and (iv) configurable rules, which is shown in Figure 2.

Event generation: The event generation Module implements automatic install, run, and uninstall of Android apps events based on the Application Program Interface (API) of *monkeyrunner*. It also can simulate user actions such as clicking buttons, touching screen, and pressing keyboard, and these actions would trigger apps to access the Internet and generate network traffic. The simulated user action events include two categories: random execution and specified execution. To execute the app randomly, we use the API of *monkeyrunner*, which simply sends in a stream of random events; it may be described as a random walk on the execution path (discuss it later). Given the ability to restart from the start state any number of times, it can eventually explore any finite execution paths. Apps that do not need any meaningful text to be filled in have a small state space consisting of screen taps and drags. Random execution can deal with such apps quite well without any knowledge of their interaction models. On the other hand, if some meaningful texts such as login credentials are required, random execution cannot enter in the right input and fails to trigger app code to generate traffic. For such cases, we need specific execution, which executes the app based on some certain paths based on receiving execute path from Path Identification Module.

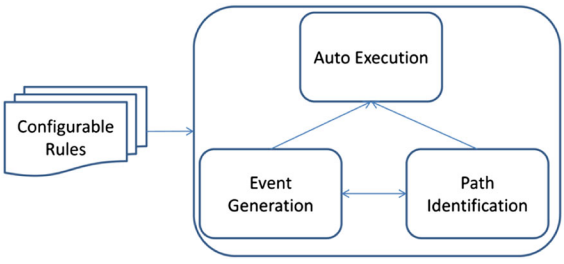


Figure 2. Overview of App execution model.

Path identification: The path identification module is responsible for identifying paths to be executed by the app. A typical Android app consists of separate screens named *Activities*. An activity acts as a container for typical *Widget* elements such as buttons and list items. Users navigate (i.e., transit among activities) different activities using the aforementioned *Widget* elements when interacting with an app. Activities can serve different purposes. For example, in a typical news app, the home screen shows the list of top popular news; selecting a news headline will trigger the transition to another activity that displays the full news item. We illustrate how activity transitions as a result of a user interaction in a popular Android app, CNN News, which is shown in Figure 3.

We have the textual description of users'actions on the top in Figure 3. In the middle, we have an actual screen shot, and on the bottom, we have the activities and their transitions. At the beginning, the screen shot of this app is in the Main Activity. When the user clicks on the US button, the app jumps to the US news list Activity (note the different screen). The screen shows a list of news and pictures. When the user presses one textual of the news list, the screen layout changes as the app jumps to the news Activity. There is no *Widget* that can be clicked in the third activity, we consider that the process of execution is end. Therefore, we considered the whole process of activity transitions as an app execution path in our approach.

It is well known that an app may contain several different activities; the transition of activity would trigger different network behaviors, and the generated traffic would have different network characteristics. We still use the app of CNN News as an example: if the user clicks the textual news, the app would generate an HTTP request to ask for text files. If the user clicks the video news, the app generates an HTTP request to ask for video files. Both of user'actions would cause two different network behaviors, and the generated traffic traces also reflect different network characteristics, such as packet size. In our approach, we aim to generate traffic traces that are similar to live Android app network traffic. Hence, we have to design an efficient algorithm to identify execution path of apps, and send the identified path to the auto execution module.

Based on observations of each execution path of the CNN News, we find that each path begins at the *Main Activity* and ends at several other activities. Then, we use a tree to represent the execution path structure, which is shown in Figure 4. The solid line stands for the execu-

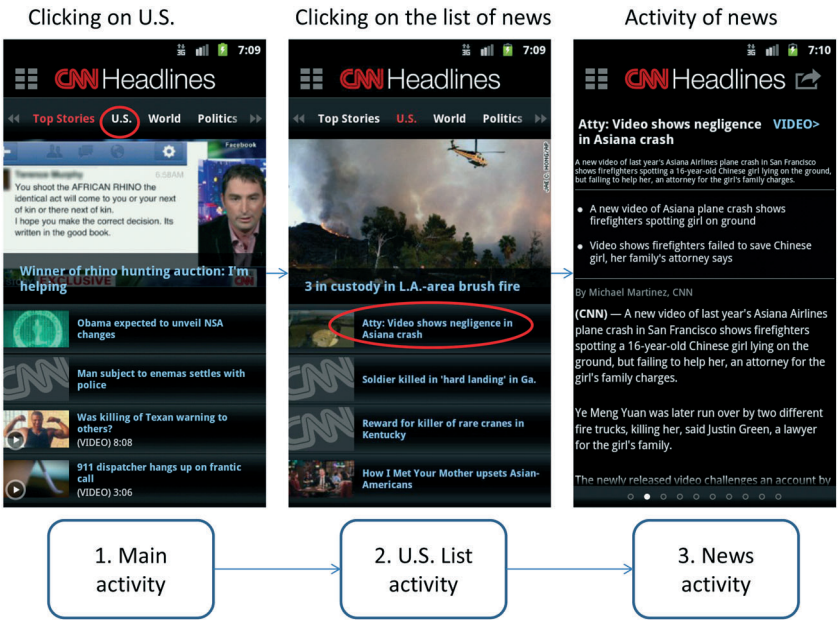


Figure 3. An example of a execution path from the popular Android app, CNN News.

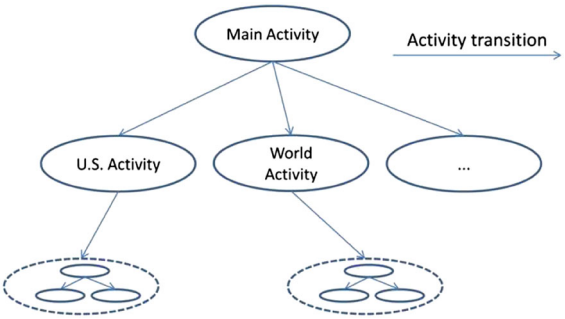


Figure 4. Process of activity transition of CNN News.

tion path, namely Activity transition. For instance, the user clicks buttons in the Activities that belong to US, World, or others categories, and the Main Activity will jump to the Activity of US, World or others, respectively. The circle with dashed line represents the rest of transition activities. A path from root node to leaf node represents an execution path of the app. Therefore, we can consider the path identification problem as a tree search problem. In this paper, we propose an algorithm named depth-first identification to identify execution path for apps. This technique employs depth-first search to mimic how an actual user would interact with the app. It is well understood that the search strategy for breadth-first search (e.g., NetworkProfiler [1]) and that for depth-first search is different. According to the characteristics of Android app activities, depth-first search scheme is able to identify a larger amount of execution paths than breath-first search scheme in the same period of time, which has been validated in Table V in Section 5.2.2, as depth-first identification is a dynamic approach, which can be performed even when the tester does not have activity transition information beforehand. Figure 4 shows the process of path identification based on depth-first identification, which we will describe in details in Algorithm 1. Algorithm 1 provides the precise description of the depth-first identification approach. We first extract the entry point activities (e.g., Main Activity) from the Manifest file of apps, and these activities will act as starting points for the execution. We then choose a starting point A and start depth-first identification from that point (lines 1–5). For each activity A_i, we extract all its *Widget* elements (line 9). We then systematically check the feature of *isClickable* of each *Widget* elements (line 11). Whenever we detect a transition to a new activity A_j, we record this change, and apply the same algorithm recursively on A_j (line 14). This process continues in a depth-first manner until we do not find any transition to a newer activity or any *Widget* element that can be clicked in that screen. We then go back to the upper-level activity and continue checking its *Widget* elements (line 15).

Next, we will explain the definition of an execution path based on depth-first identification. As we mentioned before, an execution path represents a path from root node (e.g., Main Activity) to leaf node (e.g., news Activity). Root node is the activity shown on screen when the app launches. Therefore, the definition of leaf node is the key to define execution path. In this paper, a leaf node should satisfy one of the follow three conditions according to our definition. First, there is no clickable widget in leaf node. Second, there are clickable widgets in leaf node, but the activity would not transit when a user clicks them. Third, there are clickable widgets in leaf node, but the activity would transit to the activities that have already been visited. Based on these explanations, we can define execution paths of the Android apps.

Algorithm 1 Depth-first identification

```
Require: Entry Point Activities |A|
  function DFI(|A|)
      for All Activities A_i in |A| do
          Change to Activity A_i
          DEPTH_FIRST_IDENTIFICATION(A_i)
      end for
  end function

  function DEPTH_FIRST_IDENTIFICATION(A_i)
      Widgetset ← GET_WIDGET(A_i)
      for each widget in Widgetset do
          if (Widget.isclickable == TRUE) then
              if (A_i → A_j) then
                  record this activity transition
                  DEPTH_FIRST_IDENTIFICATION(A_j)
                  Back to Activity A_i
              end if
          end if
      end for
  end function
```

Moreover, during the process of automatically executing Android apps, we observed that some apps have a similar activity hierarchy, which mean the execution path of these apps may be similar. However, it does not mean that the traffic generated by these apps will contain similar traffic characteristics. For example, let us assume that there are two book apps, *Book1* and *Book2*, both of them execute by scroll screen to read page by page. In practice, *Book1* pre-downloads books when the app starts, and *Book2* downloads books at real-time when the user reads it. Therefore, the traffic generated by these two apps will not have similar characteristics.

Auto execution: The auto execute module is responsible for executing the app to generate traffic traces. This module executes apps based on receiving different kinds of actions and configurable rules from the event generation module and the configurable rule module (the latter module will be discussed later), respectively. As we have mentioned earlier, the execution action can divide into random action and specific action. In the random action, this module receives random execution events from the event generation module and configurable rules from the configurable rule module. In the specific action, this module receives action events, execution paths, and configurable rules from the event generation module, path identification module, and the configurable rule module, respectively. Both execution actions ensure that the generated traffic traces can cover the network behaviors of the app as much as possible and closely emulate live network traffic. This module is also responsible for launching traffic capture tool, *tcpdump* [33], automatically to capture traffic generated by Android apps. The captured traffic traces could also be used for high-level analysis.

Configurable rule: To be flexible and comprehensive, AndroGenerator can generate various kinds of traffic based on choosing several kinds of configurable rules. We list three main configuration rules and their description as follows.

- **App composition:** This rule allows AndroGenerator to generate different kinds of traffic traces based on choosing different sets of Android apps. For example, if we want to study network characteristics of benign Android app traffic, then we should choose apps from Google Play as the execution data set for AndroGenerator. On the other hand, if we want to generate a mixed traffic trace, then we would choose several kinds of app as the execution data set for AndroGenerator (e.g., 40% of apps from Google Play, 40% of apps from third-party markets, and 20% of apps from malware).
- **Execution duration:** This rule divides into two categories: the first is fixed execution duration and the other one is dynamic execution duration. The fixed execution duration is generally used to combine with specific action for automatic execution of mobile apps, and generated traffic traces can be used for the study of network traffic characteristics. The range of fixed execution duration is from 1 to 10 minutes, and the time interval is 1 minute (the duration of typical average app session is 71.56 seconds [34]). To determine maximum execution time for dynamic execution duration, we set the execution time distribution in the range of 1 to 300 seconds. The distribution of execution time of each app follows the Poisson distribution. The dynamic execution duration is usually combined with random action for analysis of Android app traffic identification.
- **Execution action:** This rule decides what kind of execution action is generated by the event generation module, which has already been discussed.

4.2.2. Traffic record and process.

Extracting network characteristics: To capture network characteristics of generated traffic traces, we extract packet and flow characteristics from the generated traffic traces. As we have mentioned, most Android apps run over HTTP protocol. Therefore, network characteristics extraction focuses on HTTP traffic. The first class of network characteristics extracted from the generated traffic traces is based on packet characteristics, which indicate the total number of packets or bytes transferred for the apps, such as packet count and byte count. The second class is flow characteristics, which stands for the size of flow and the number of packets per flow. The third class is the HTTP characteristics, which indicates that the network behavior can be characterized in terms of the different HTTP requests, such as host name, request method, and length of HTTP request. These extracted features are stored in a database and updated by newly extracted results. There is one thing that needs to be explained: the component does not consider unsuccessful connections, such as incomplete Transmission Control Protocol (TCP) three-way handshaking traffic.

Extracting advertisement traffic: Besides network characteristics extraction, this component also extracts traffic generated by advertisement libraries. The advertisement library (also known as ad library) is an in-app library, which is defined as an activity in Mainfest file. At runtime, the embedded ad libraries are launched together with the host app, establishing connections with ad network's servers to request ads for display. Hence, ad traffic plays an important role in Android app traffic, and we also need to reproduce it in our approach.

Compared with normal Android app traffic, the protocols used by ad libraries for fetching advertisements are generally based on plain HTTP requests, with most using HTTP GET methods. In contrast, normal Android app traffic generally contains several more types of protocols because of the rich and diverse functionalities, such as using HTTP to interact with servers, using HTTPS for the purposes of authentication and confidential data transmission, and using simple mail transfer protocol to send email. Second, the majority of ad libraries would refresh their ad content in background after a certain amount of time by accessing the same ad servers regardless of the change of the ad content [20]. Android apps also have background services, but they only receive messages when there are something new. Moreover, the ad libraries that studied differ slightly in the way they interact. As an example, AdMob acts as an internal mediation service to aggregate all Google's advertisement services (e.g., Doubleclick and AdSense), whereas InMobi only requires a single HTTP POST request per action. In addition, Millennial Media needs two HTTP connections with two different servers: one to obtain the ad, and the other one to obtain the associated static content. Android apps would access more servers and generate more HTTP request flows during the running time [9].

In this paper, we use the combination of host name and redirect domain name to extract ad traffic. The extraction method based on host name has been previously used successfully by Vallina-Rodriguez *et al.* [20] to characterize ad traffic. Finally, we total extract ad traffic generated by 53 different ad libraries. Table I list s host name and redirect domain of the10 most popular ad libraries.

In Table I, we find that there are two ad libraries that need to redirect HTTP requests to another URL and this scenario was not discussed in [20]. These traffics generated by connecting to the redirect domain also should be treated as ad traffic.

Table I. Host name and redirect domain of popular ad libraries.

Ad library	Host name	Redirect domain
Admob	admob.com	
Doubleclick	g.doubleclick.net	lawoethailand.net
Umeng	umeng.com	
Google.ads	googleadservices.com	airkx.com
Google-analytics	google-analytics.com	
Mobclix	mobclix.com	
Adwhirl	adwhirl.com	
Youmi	youmi.net	
Flurry	flurry.com	

4.2.3. Traffic generator.

This component is composed of the following modules:

- **Network signature generator (NSigGen):** It automatically identifies Android apps's signatures from captured traffic traces, such as *URI* in HTTP request packets.
- **Pattern generator (PatternGen):** It extracts network characteristics of Android apps, such as packet size and flow size distribution.
- **Traffic generator (TrafficGen):** It follows a configurable traffic profile. It also inserts real application network signatures and network characteristics (from NSigGen and PatternGen) into synthetic traffic.

In the following paragraphs, we will describe each component in detail.

NSigGen: In previous work [35–37], AutoSig [35] is used for finding application signatures, LASER [36] and Autograph [37] focuses on how to find the signatures of worms. Based on that, NSigGen uses the algorithm implemented by AutoSig, with some enhancements to reduce the number of false positives and memory consumption. Its main focus is to identify the most relevant substrings in a given set of sampled flows, to rank in order of relevance, and to combine them if necessary. Each substring uniquely represents an Android app. NSigGen also uses a sibling tree data structure to create signatures as opposed to AutoSig, which uses a regular tree. A sibling tree keeps all nodes that are brothers on the same level. With this approach, this component is able to create signatures faster and with a higher precision.

PatternGen: The PatternGen component computes packet size distribution based on receiving extracted packet size from NSigGen component. This component also computes other network characteristics (e.g., flow size and number of packets) and sends these distributions patterns to TrafficGen.

TrafficGen: Given the network characteristics of packets and flows, we are now in a position to generate the actual traffic trace based on these characteristics. TrafficGen generates traffic by following configurable profiles parameterized by the NSigGen and PatternGen. The information of parameterized characteristics is listed in Table II.

For each app in the original trace, we generate a list of packets and flows according to the distribution of packet and flow size extracted in the original trace. In the beginning, TrafficGen reads the configuration profiles. For our target scenarios, we typically require thousands of TrafficGens, and each of them is configured to generate traffic matching the characteristics of a single app. We create the necessary configuration profiles for all TrafficGens according to our extracted app network characteristics. Then, we

DOI: 10.1002/sec

Table II. Detail information of configurable profiles.

Category	Parameter: *Description*
Packet	PacketLen: *Size of packet*; Signature: *URI in HTTP request, payload in packet*
Flow	FlowLen: *Size of flow*; HTTPLen: *Size of request/response*
Application	App composition: *Combination of certain apps or app categories*

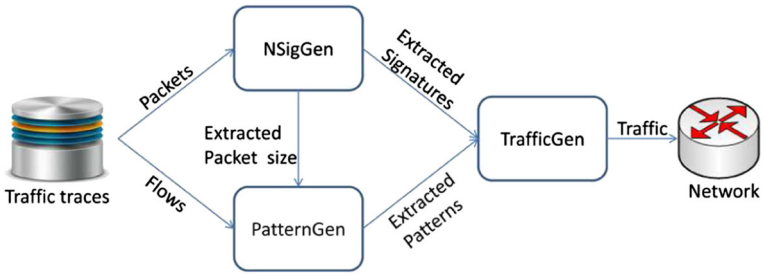


Figure 5. Process of traffic generation.

run the TraffiGens on a cluster of commodity workstations, multiplexing multiple instances on each workstation depending on the requirements of the generated trace.

Process of traffic generation: Figure 5 shows an overview of the traffic generation process. Basically, it has two phases, namely, the creation of the desired traffic profile and the composition of app traffic that will be used for high-level analysis. Initially, we must collect traffic from different Android apps separately to extract its network characteristics more precisely. Then it is necessary to profile traffic and to extract properties related to network characteristics of app. NSigGen and PatternGen tools play an important role in these phases.

From the traffic generation profile configuration file, TrafficGen generates comprehensive traffic traces that follow the specification and create several flows for each app. For example, traffic profile may define that 80% of the sending traffic must come from benign apps and 20% from Android malware or app category of game, video, social, and news, each of them accounts for 25% of the generated traffic.

5. EVALUATION

In this section, we evaluate the ability of AndroGenerator to generate traffic that exhibits the same statistical network characteristics as the real-world traffic traces.

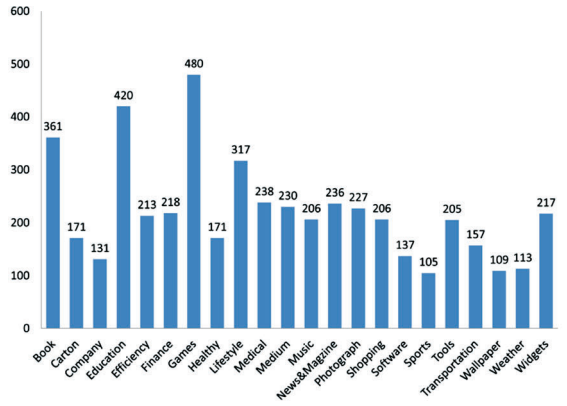
5.1. Data set and environment setup

5.1.1. Android app dataset.

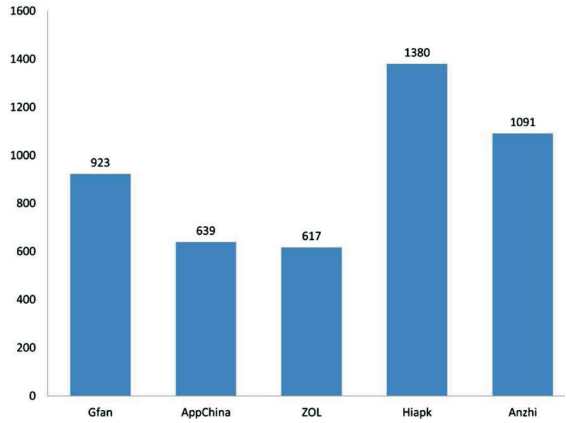
We used one Android app dataset to self-parameterize AndroGenerator in this paper, which is crawled from the popular Android markets over 2 months (December, 2012 through January, 2013). The Android app dataset consists of three markets and one dataset, 10 778 distinct Android apps. Specifically, we designed a tool for downloading Android apps from Google Play based on a java library named *androidmarketapi* [38], and this library allows access to official Android market servers. Then we use this tool to download the most popular free applications from 22 categories in Google Play, which cover most app categories as defined in Google Play, such as Game, Entertainment, and Productivity tools. Meanwhile, we manually downloaded applications in some alternative markets in China, like Gfan, Appchina, Hiapk, and ZOL. The category composition of Android apps is shown in Figure 6. In Figure 6, the *x*-axis represents different app categories in Google Play or different third-party Android markets, and the *y*-axis represents the number of Android apps that we downloaded from each app category or third-party Android market. We also utilize two sets of malware apps. The first malware dataset comes from Android Malware Genome Project [16], which contains more than 1200 malware samples that cover the majority of existing Android malware families, ranging from the ones that were added at their debut in August 2010 to recent ones that were added in October 2011. The second malware dataset comes from Contagio [39], which contains 116 malware samples from June 2012 to February 2015. Moreover, we executed each Android app in a controlled environment, which consists of 10 Android emulators; each app was executed in 5 min and collecting the generated traffic traces by *tcpdump* [33].

5.1.2. Real-world traffic traces.

We collected two different real-world mobile traffic traces to evaluate our approach. The first traffic trace (named **Trace 1**) consists of 5 days of mobile traffic collected from eight Android Mobile users and two iOS Mobile users in our lab. It mainly contains several popular mobile apps, such as *WeChat*, *QQ*, and *Taobao*, for data sent and received by the smartphone. All users are knowledge workers. Because of the poor 3G signal coverage, each of them is connected to Wifi in full signal conditions. The second traffic trace (named **Trace 2**) is collected



(a) Category composition of Android apps in Google Play



(b) App composition of third-party markets

Figure 6. App composition of our dataset.

from the backbone network of Hunan University, 12 and 13 January 2014 at Hunan University; **Trace 2** contains more kinds of mobile users and apps than **Trace 1**. Because the wireless traffic traces may contain traffic generated by laptops, we need to extract mobile traffic from this trace. For this, we used several known mobile traffic identification techniques such as mobile domain name [8], *User-agent* field in HTTP header [19], and ad traffic extraction. These techniques cannot identify all mobile traffic from the wireless traffic traces, but can make sure that the identified traffic is truly mobile traffic. The traffic traces used for evaluation are summarized in Table III.

5.1.3. Environment setup.

Our evaluation test environment consists of one workstations running the Ubuntu 10.04 operating system. The machine had 1.8 GHz Intel Xeon 4 processors, 16 Gigabyte of RAM, used kernel defaults for TCP/IP parameters, and the default receive window size was 64 KB. Our Ubuntu machine was used not only as AndroGenerator data server but also as client for generating requests and receiving the resulting data packets. We monitored the host during our tests to ensure the systems did not exhaust all available CPU or memory resources.

5.2. Comprehensiveness of traffic generation

5.2.1. Activity coverage of Android apps.

To evaluate the effectiveness of AndroGenerator, and the comprehensiveness of generated traffic, we use a performance metric, activity coverage, in our evaluation study. Activity coverage is defined as the quotient of the number of activities that is executed by AndroGenerator over the total number of activities. In our experiments, we analyze the activity coverage of the AndroGenerator on 15 popular Android apps, such as *Angry Bird*, *BBC News*, *QQ*, and *Kindle*, *Tumblr*. We also compare the activity coverage with three other well-known approaches, namely, *ProfileDroid* [9], *Monkey* [40], and *NetworkProfiler* [1]. *ProfileDroid* presented a manual execution that is a direct way to execute Android apps, *Monkey* is an Android testing tool that is able to generate random event stream to execute Android apps, and *NetworkProfiler* is similar to our work which also execute Android apps automatically, and extract fingerprints from the generated network traffic to identify Android apps. Figure 7 shows the comparison results of activity coverage.

In Figure 7, we find that *Monkey* achieves a lowest activity coverage among the four execution methods. This is because *Monkey* only uses random events to execute

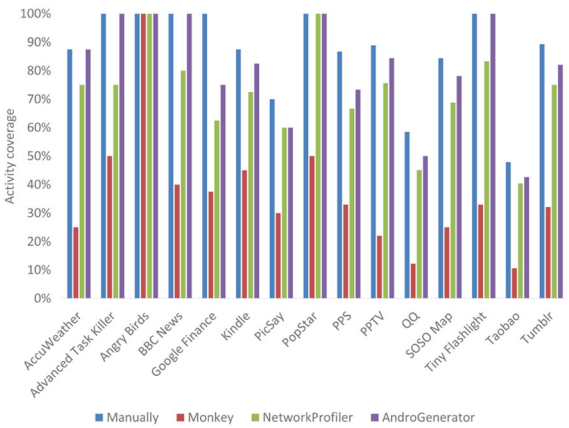


Figure 7. Comparison results of activity coverage.

Table III. Summary of traffic traces used in evaluation experiments.

Traffic trace	#TCP flows	#Packets	#MBytes	#HTTP flows	#Packets	#MBytes
Trace 1	78 322	3 998 438	684.95	51 432	3 392 762	411.38
Trace 2	507 982	11 487,598	9 539.01	358 991	10 430 739	6 678.26

Android apps, and the activity coverage would decrease when the number of activity increases. Manual execution achieves the highest activity coverage among the 15 apps, but this method cannot scale to large number of Android apps. On the other hand, Activity coverage of our approach is overall close to that of Manual execution. In particular, some apps, such as *Angry Birds* and *BBC News*, have the same activity coverage between our approach and manual execution. Finally, our approach is able to achieve higher activity coverage than *NetworkProfiler*, because our approach first obtains *Widget* information, then generates the corresponding execution event for each *Widget*, and finally use the Depth-first identification algorithm to execute Android apps. Based on comparison result, it is clear that our approach is effective and efficient for automatic executing of Android apps in terms of the comprehensiveness of the generated traffic.

5.2.2. Time of path identification.

As we show in Table IV, in the same amount of time (5 min in this case), our approach that utilizes depth-first search scheme can identify more execution paths than Dai's approach that uses breadth-first search on a variety of mobile apps.

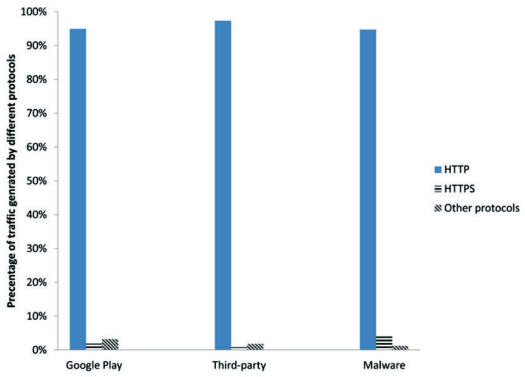
In Table IV, Column 2 presents the total number of paths identified by AndroGenerator. Column 3 represents the number of paths identified by NetworkProfiler. We find that when the number of paths identified by AndroGenerator is small (e.g., 1), NetworkProfiler would also achieve a similar value. However, when the number of paths increases, AndroGenerator would identify more paths than NetworkProfiler. AndroGenerator uses depth-first search scheme to explore the apps, which could identify paths of the apps faster than width-first search scheme (e.g., NetworkProfiler).

Table IV. The results of the number of path identified by AndroGenerator and NetworkProfiler.

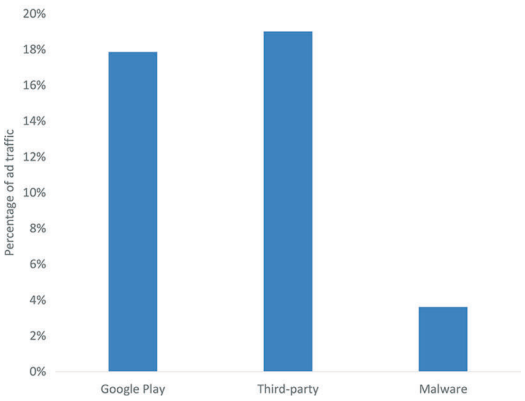
App	#Path identified by our approach	#Path identified by NetworkProfiler
AccuWeather	6	5
Advanced Task Killer	4	4
Angry Bird	1	1
BBC News	10	6
Google finance	8	6
Kindle	20	13
PicSay	7	6
PopStar	1	1
PPS	11	5
PPTV	15	8
QQ	22	14
SOSO map	12	5
Tiny Flashlight	6	3
Taobao	24	16
Tumblr	8	5

5.2.3. Composition of traffic trace.

After evaluating the activity coverage percentage of the 15 apps, we also study the application layer protocol and ad traffic composition of the generated traffic traces in Figure 8. It is well understood that the HTTP protocol should be used most of the time when Android apps access the Internet, and Figure 8(a) shows that the HTTP traffic accounts for majority of total traffic generated by the apps from three datasets. HTTPS is often used in the authentication phase of the session, such as *Facebook* login procedure, or encrypted data transmission. Therefore, HTTPS traffic only accounts for a small percentage in overall network traffic. In addition to those two kinds of traffic, traffic generated by other protocols also accounts for a small percentage, such as ftp or some other self-defined protocols. Figure 8(b) shows the percentage of ad traffic in each kind of apps; we find that three kinds of apps would generate ad traffic, and the percentage of ad traffic in Google Play is similar with third-party. We also find the percentage of ad traffic in malware is less than the former two kinds of apps. These results mean that ad traffic is common in each kind of app, and the majority of malware traffic generated by malware own not the ad libraries.

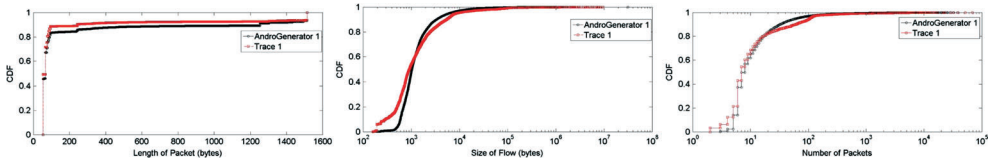


(a) Application-layer protocol composition of generated traffic



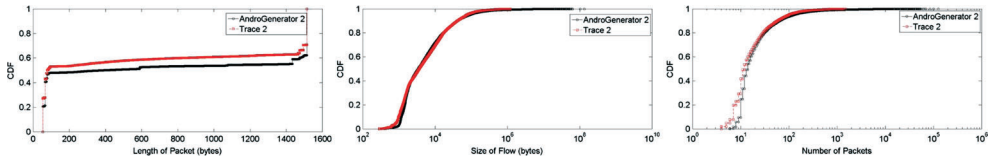
(b) Ad Traffic composition of generated traffic

Figure 8. Composition of generated traffic.



(a) Length of packet distribution (b) Size of flow distribution (c) Number of packet distribution

Figure 9. Comparison of empirical distributions extracted from **Trace 1** with distributions produced during AndroGenerator emulation (*x*-axes of (b), (c) are log scale).



(a) Length of packet distribution (b) Size of flow distribution (c) Number of packet distribution

Figure 10. Comparison of empirical distributions extracted from **Trace 2** with distributions produced during AndroGenerator emulation (*x*-axes of (b), (c) are log scale).

5.3. Results of traffic comparison

We first evaluate AndroGenerator's ability to reproduce mobile app traffic trace characteristics by comparing with the traffic traces collected from several of the most popular mobile apps, namely **Trace 1**. Then, we extract network characteristics from top 100 apps in Google Play, and use AndroGenerator to reproduce traffic traces, namely **AndroGenerator 1**. The detail information of **AndroGenerator 1** is shown in Table V. Figure 9 compares the size of packets, flows, and number of packets per flow frequency empirical distributions derived from the **Trace 1** dataset with the distributions generated by AndroGenerator. For length of packet, which is shown in Figure 9(a), there is a good overall match except at the length range of 100 to 1200 bytes. The non-matching segment of two curves is caused by the fact that **Trace 1** contains more small length packets (smaller than 100 bytes) than the generated traffic trace. Figure 9(b) compares flow size extracted from the **Trace 1** data set with the flow sizes generated by AndroGenerator. It is clear that they are very similar except at the smaller size of flows. Figure 9(c) plots the number of packet contained by each flow on a log scale for both of **Trace 1** and traffic trace generated by AndroGenerator. We observe a close match between the collected traffic trace and the AndroGenerator-generated traffic trace, and results for size of flows are similar.

Our second evaluation test was to compare the traffic traces generated by AndroGenerator against the traffic traces collected from all wireless traffic of Hunan University between a certain time interval, namely, **Trace 2**. Then, we extract network characteristics from a mixed app dataset, which contains apps from Google Play and third-party markets and malware, and use AndroGenerator to reproduce traffic traces, namely, **AndroGenerator 2**. The detail information of **AndroGenerator 2** is shown in Table V. Figure 10 shows the results of comparison between traffic traces generated by AndroGenerator and **Trace 2**. Figure 10(a) shows the comparison of packet length distribution between traffic traces generated by AndroGenerator and **Trace 2**, which has a good match at beginning and end of two curves because **Trace 2** contains more small length packets (smaller than 100 bytes). Figure 10(b) shows the comparison of flow size between traffic traces generated by AndroGenerator and **Trace 2**. Comparing the result with Figure 9(b), AndroGenerator can generate more realistic traffic traces for larger-scale traffic in terms of flow size. Figure 10(c) plots number of packet contained by each flow on a log scale for both of **Trace 1** and traffic trace generated by AndroGenerator. We observe a very good match between the collected traffic trace and the AndroGenerator generated traffic trace, and results for size of flows are similar.

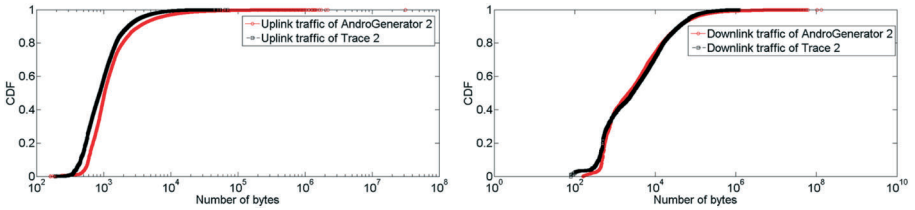
Figure 11 shows the comparison of the uplink (from apps to network) and downlink (from network to apps) traffic size between the traffic traces generated by AndroGenerator and **Trace 2**. From this figure, we can find that the sizes of uplink and downlink traffic between AndroGenerator and **Trace 2** are very similar to each other. In addition, the comparison results also demonstrate that the AndroGenerator not only has ability to generate network traffic that perfectly matches the real-world traffic in terms of overall traffic size, but the generated network traffic is also very similar to the real-world traffic in both uplink and downlink directions.

As we introduced earlier, most of the Android apps run over HTTP protocol. In the evaluation traffic traces, we found that traffic of other mobile app (e.g., iOS app) is also carried over HTTP. Moreover, AndroGenerator can

Table V. Summary of traffic traces generated by AndroGenerator.

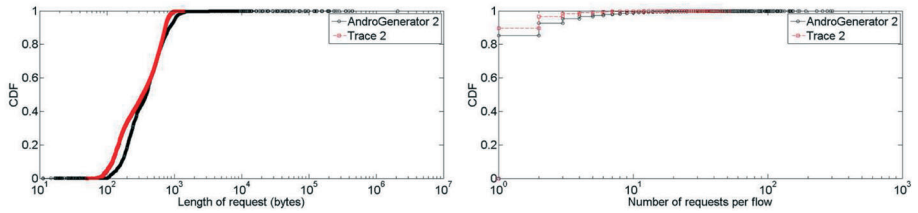
Traffic trace	#HTTP flows	#Packets	#MBytes
AndroGenerator 1	50 906	3 510 167	452.81
AndroGenerator 2	338 590	11 266 027	6 039.57

DOI: 10.1002/sec



(a) Uplink traffic (b) Downlink traffic

Figure 11. Comparison of uplink and downlink traffic between **Trace 2** and AndroGenerator (*x*-axes of both figures are log scale).



(a) Length of request distribution (b) Number of request distribution

Figure 12. Comparison of HTTP request characteristics extracted from **Trace 2** with HTTP request characteristics produced during AndroGenerator emulation (*x*-axes of both figures are log scale).

generate network signatures to simulate the live mobile traffic. Therefore, the third evaluation test was to compare the HTTP request packets generated by AndroGenerator with those generated in **Trace 2**, and Figure 12 shows the comparison results. In Figure 12(a), we found that there is a good overall match between traffic generated by AndroGenerator and **Trace 2** in length of request packets. This result also proves that our approach can generate realistic mobile traffic for identification of mobile app studies. Figure 12(b) shows AndroGenerator also matches real traffic well in the characteristic of number of request packets.

6. DISCUSSION

In this paper, we develop a comprehensive framework for generating realistic Android app network traffic traces. We find that capturing and reproducing essential characteristics of a traffic trace requires techniques for automated app execution, packet and flow characteristics extraction, and traffic generation. Our approach, AndroGenerator, uses these techniques to generate representative Android app network traffic based on several distributions characteristics of TCP flows, especially HTTP flows. Moreover, AndroGenerator implements on commodity operating systems subject to the communication characteristics of several appropriately configurable profiles. In the evaluation, we first show the activity coverage of AndroGenerator, and compare this result with three other well-known approaches. Comparison results show that our approach can achieve the best activity coverage among these approaches, which means our approach is able to trigger the largest number of network behaviors for Android apps. Second, we generated traces from AndroGenerator that match the network characteristics of packet length, flow size, and packet number per flow well when they are compared with two kinds of real-world traffic trace. In addition, the experimental results also show that AndroGenerator can generate similar results in terms of length of request and number of request packets per flow. These results suggest that AndroGenerator could be useful as a system for providing network operators and security analyzers some insight regarding how to conduct study of mobile traffic characterization, malware detection, and mobile app identification.

Because automatic execution does not work well when the widgets are customized in a non-standard manner, to address this issue, we will explore the possibility of using machine-learning techniques to train classification models from the well-known widgets, and then used these trained classifiers to identify which type of these custom-made widgets truly are. Second, automatic execution is particularly useful when user credentials or some meaningful information is required, and these credentials can easily be used for subsequent navigation. In practice, account registration and log in process for most Android apps is carried out via external websites. However, our work currently cannot handle with webpages. Therefore, whenever the mobile app requires user log-in using an external website, our current system cannot continue the app execution process. To address this problem, we integrate web-based GUI exploration techniques into AndroGenerator in future work, such as seleniumHQ [41]. The two aforementioned issues make some apps achieve relatively low activity coverage, but overall, the activity coverage results for the majority of Android apps show high accuracy. In addition, sometimes it is very useful to provide other meaningful inputs such as a city name or a zipcode for some specific Android apps. For example, the AccuWeather app asks for the location information when

it is hard for the app itself to determine the location. Automatic execution is quite stunted if the location information is not provided.

There exist four different types of components in an application, each defining a different interface to the system: *activities*, *services*, *content providers,* and *broadcast receivers*. Every application can declare several components of each type in the manifest. In this paper, we primarily focus on *activity* execution because it is shown on screen and interacts with users directly. However, other components can also generate network traffic when they run. For example, some apps would use *services* to communicate with remote servers. Some of the service components will automatically start when the apps start, other service components will start only if specific activities of the mobile apps are executed, such as music download service. In this paper, we have verified that the proposed AndroGenerator can achieve a high activity coverage, which indicates that it is likely that most of the services can be triggered when the app activities are covered. This verified result also indicates that specific background traffic generated by background service could be captured as well as the traffic generated by activities. Therefore, the specific background traffic characteristics can be analyzed. However, because it is hard to distinguish traffic generated by background services or activities, we analyze these traffic characteristics as a whole. Obviously, because it is impossible to make the activity coverage for all apps reach 100%, there is no guarantee that all the services of an app are triggered, especially those services that can only be triggered by some specific activities. In our ongoing and future work, we will investigate the relationship between activities and services in mobile apps. By this means, more services can be triggered during the automatic execution of mobile apps, and more app network behaviors can be captured.

Finally, AndroGenerator currently focuses on generating HTTP traffic. As shown in Figure 8, Android apps would generate other types of traffic, such as HTTPS and FTP. One possible future direction is to extend our approach to simulate more network protocols and cover network behaviors of Android apps as much as possible for the purpose of generating more realistic traffic.

ACKNOWLEDGEMENTS

This work is supported by the National Basic Research Program of China (973) under Grant 2013CB315805, the National Science Foundation of China under Grant 61173167, 61473123, the Jiangsu Science Project under Grant No. BY2013095-1-05, and the National Science Foundation of China under Grant 61173168.

REFERENCES

1. Dai S, Tongaonkar A, Wang X, Nucci A, Song D. Networkprofiler: Towards automatic fingerprinting of android apps. In *Proceedings of the 32nd IEEE International Conference on Computer Communications, Infocom 2013*: Turin Italy: IEEE, 2013; 809–817.
2. Kasperskey lab. first SMS trojan detected for smartphones running Android. Available from: http://www.kaspersky.com/news?id=207576158 [Accessed on 2010].
3. Lookout. update: security alert: Droiddream malware found in official Android market. Available from: http://blog.mylookout.com/blog/2011/03/01/security-alert-malware-found-in-official-android-marketdroid-dream/ [Accessed on 2011].
4. Security alert: Anserverbot new sophisticated Android bot found in alternative Android markets. Available from: http://www.csc.ncsu.edu/faculty/jiang/AnserverBot [Accessed on 2011].
5. Android basebridge. Available from: http://www.syman tec.com/security_response/writeup.jsp?docid=2011- 060915-4938-99&tabid=2 [Accessed on 2011].
6. Enck W, Gilbert P, Chun B, Cox L, Jung J, McDaniel P, Sheth A. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, USENIX OSDI 2010*. USENIX Association: Vancouver, BC, Canada, 2010; 1–6.
7. Enck W, Octeau D, McDaniel P, Chaudhuri S. A study of Android application security. In *Proceedings of the 20th USENIX Conference on Security, SEC 2011*. USENIX Association: San Francisco, CA, USA, 2011; 21–21.
8. Lever C, Antonakakis M, Reaves B, Traynor P, Lee W. The core of the matter: analyzing malicious traffic in cellular carriers. In *Proceedings of the 20th Network and Distriuted System Security Symposium, NDSS*: San Diego, CA, USA, 2013.
9. Wei X, Gomez L, Neamtiu I, Faloutsos M. Profiledroid: multilayer profiling of android applications. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking, MobiCom*. ACM: Istanbul, Turkey, 2012; 137–148.
10. Weigle M, Adurthi P, Hernández-Campos F, Jeffay K, Smith F. Tmix: a tool for generating realistic TCP application workloads in ns-2. *SIGCOMM Computer Communication Review* 2006; **36**(3): 65–76.
11. Sommers J, Barford P. Self-configuring network traffic generation, *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement,* IMC '04, New York, NY, USA, 2004; 68–81. ACM.
12. Medina A, Taft N, Salamatian K, Bhattacharyya S, Diot C. Traffic matrix estimation: existing techniques and new directions. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications,*

SIGCOMM. ACM: Pittsburgh, Pennsylvania, USA:, 2002; 161–174.
13. Le L, Aikat J, Jeffay K, Smith F D. The effects of active queue management on web performance, In: *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications SIGCOMM*, Karlsruhe, Germany, 2003; 265–276.
14. Khaled H, Azer B, John B. Measuring bottleneck bandwidth of targeted path segments. *IEEE/ACM Trans. Netw* 2009; **17**(1): 80–92.
15. Jain M, Dovrolis C. End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput. *IEEE/ACM Transactions on Networking* 2003; **11**(4): 537–549.
16. Zhou Y, Jiang X. Dissecting Android malware: characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy, S&P 2012*. IEEE Computer Society: Francisco, CA, USA, 2012; 95–109.
17. Monkeyrunner. Available from: http://developer.android.com/tools/help/monkeyrunner_concepts.html [Accessed on 2010].
18. Hierarchy viewer. Available from: http://developer.android.com/tools/help/hierarchy-viewer.html [Accessed on 2010].
19. Xu Q, Erman J, Gerber A, Mao ZQ, Pang J, Venkataraman S. Identifying diverse usage behaviors of smartphone apps. In *Proceedings of the 11th Internet Measurement Conference, IMC, 2011*. ACM: Berlin, Germany, 2011.
20. Vallina-Rodriguez N, Shah J, Finamore A, Grunenberger Y, Haddadi H, Papagiannaki K, Crowcroft J. Breaking for commercials: characterizing mobile advertising. In *Proceedings of the 12th Internet Measurement Conference, IMC 2012*. ACM: Boston, Massachusetts, USA, 2012.
21. Rolland C, Ridoux J, Baynat B. LiTGen, a lightweight traffic generator: application to P2P and mail wireless traffic. In *Proceedings of the 8th International Conference on Passive and Active Network Measurement*. Louvain-la-Neuve: Belgium, 2007; 52–62.
22. Hafsaoui A, Nikaein N, Wang L. Openairinterface traffic generator (OTG): a realistic traffic generation tool for emerging application scenarios, *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Washington, USA, 2012; 492–494.
23. Sohn K, Park C, Shin K, Kim J, Choi S. Online gaming traffic generator for reproducing gamer behavior, In *Proceedings of the 9th International Conference on Entertainment Computing*, Seoul, Korea, 2010; 160–170.
24. Machiry A, Tahiliani R, Naik M. Dynodroid: An input generation system for Android apps, *FSE*, Saint Petersburg, Russia, 2013; 224–234.
25. Yang W, Prasad MR, Xie T. A grey-box approach for automated GUI-model generation of mobile applications, *FASE*, Roma, Italy, 2013; 250-265.
26. Ravindranath L, Nath S, Padhye J, Balakrishnan H. Automatic and scalable fault detection for mobile applications, *MobiSys*, Bretton Woods, New Hampshire, USA, 2014; 190–203.
27. Rastogi V, Chen Y, Enck W. Appsplayground: automatic security analysis of smartphone applications, *CODASPY*, San Antonio, Texas, USA, 2013; 209–220.
28. Gilbert P, Chun B, Cox LP, Jung J. Vision: Automated security validation of mobile apps at app markets, *MCS*, Bethesda, Maryland, USA, 2011; 21–26.
29. Mahmood R, Esfahani N, Kacem T, Mirzaei N, Malek S, Stavrou A. A whitebox approach for automated security testing of Android applications on the cloud. In *2012 7th International Workshop on Automation of Software Test (AST)*: Zurich, Switzerland, June 2012; 22–28.
30. Hao S, Liu B, Nath S, Halfond WGJ, Govindan R. Puma: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services, Mobisys 2014*. ACM: Bretton Woods, New Hampshire, USA, 2014; 204–217.
31. Falaki H, Lymberopoulos D, Mahajan R, Kandula S, Estrin D. A first look at traffic on smartphones. In *Proceedings of the 10th Internet Measurement Conference, IMC 2010*. ACM: Melbourne, Australia, 2010; 281–287.
32. Grace MC, Zhou W, Jiang X, Sadeghi A. Unsafe exposure analysis of mobile in-app advertisements, In *WiSec*, Tucson, Arizona, USA, 2012; 101–112.
33. Tcpdump. Available from: http://www.tcpdump.org/ [Accessed on 2014].
34. Böhmer M, Hecht B, Schöning J, Krüger A, Bauer G. Falling asleep with angry birds, Facebook and Kindle: a large scale study on mobile application usage, *MobileHCI*, Stockholm, Sweden, 2011; 47–56.
35. Ye MJ, Xu K, Wu JP, Po H. AutoSig-automatically generating signatures for applications, In *CIT*, Xiamen, China, 2009; 104–109.
36. Park BC, Won YJ, Kim MS, Hong JW. Towards automated application signature generation for traffic identification, In *NOMS*, Salvador, Bahia, 2008; 160–167.

37. Kim HA, Karp B. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th Conference on USENIX Security Symposium, SSYM 2004,*. USENIX Association: San Diego, CA, USA, 2004; 19–19.
38. Android-market-api. Available from: https://code.google.com/p/android-market-api/ [Accessed on 2012].
39. Contagio. Available from: http://contagiodump.blogspot.com/ [Accessed on 2012].
40. Ui/application exerciser monkey. Available from: http://developer.android.com/guide/developing/tools/monkey.html [Accessed on 2011].
41. seleniumhq. Available from: http://docs.seleniumhq.org/ [Accessed on 2013].

DOI: 10.1002/sec