Hindawi Publishing Corporation
Mobile Information Systems
Volume 2016, Article ID 1740129, 11 pages
http://dx.doi.org/10.1155/2016/1740129

Research Article

QDroid: Mobile Application Quality Analyzer for App Market Curators

Jemin Lee and Hyungshin Kim

Department of Computer Science and Engineering, Chungnam National University, Daejeon, Republic of Korea

Correspondence should be addressed to Hyungshin Kim; hyungshin@cnu.ac.kr

Received 25 April 2016; Accepted 1 September 2016

Academic Editor: Yuh-Shyan Chen

Low quality mobile applications have damaged the user experience. However, in light of the number of applications, quality analysis is a daunting task. For that reason, QDroid is proposed, an automated quality analyzer that detects the presence of crashes, excessive resource usage, and compatibility problems, without source codes and human involvement. QDroid was applied to 67 applications for evaluation and discovered 78% more crashes and attained 23% higher Activity coverage than Monkey testing. For detecting excessive resource usage and compatibility problems, QDroid reduced the number of applications that required manual review by up to 96% and 69%, respectively.

1. Introduction

An increasing variety of applications (or apps) is becoming accessible through expanding app markets. Unlike enterprise software, mobile apps require that their developers consider performance, resource usage, and compatibility of their apps in diverse devices, in addition to whether they are bug-free [1–4]. Individual developers frequently have difficulties coping with these considerations because of testing complexity.

There are now 1.5 million apps on Google's Play Store, of which 14% are low quality apps, according to statistics [5], provided by AppBrain. Low quality apps have abnormal exits, large energy consumption, and compatibility problems, which damage the user experience (UX) and usability of the app market. In light of these issues, there is a pressing need for market curators to examine apps before publication. Unfortunately, it is a daunting task to manually examine a large number of apps in a timely fashion.

To tackle these challenges, many researchers have proposed various testing approaches that can cover large numbers of apps in a reasonable time without the need of source codes. Basically, these approaches for mobile apps can be broadly classified into static and dynamic types. Despite covering a large number of apps, static approaches [6–8] are hindered by commonly used app features such as code obfuscation, native libraries, and a complex SDK framework. For these reasons, recent works have focused on dynamic approaches, like Monkey, where the runtime properties of an app are examined by feeding random or deterministic events, such as tapping on a button and typing text [9–12]. These Monkey-based approaches have been used for specific goals, such as detecting crashes [9, 12, 13], privacy [10], and GUI violations [11]. However, all the prior research has focused on bugs, privacy, and security. There is no scalable approach yet that focuses on verifying multiple quality factors for market curators. By focusing only on a single analysis, the most significant issue of an app could be missed, such as excessive resource usage or a compatibility problem in differently sized screens.

In this paper, we propose an automated quality analyzer called QDroid that efficiently explores app pages and verifies the presence of crashes, excessive resource usage, and compatibility problems in differently sized screens without source code. We anticipate that this analyzer will be used by market curators to maintain app markets. Moreover, QDroid is proposed as open-source software (https://github.com/leejaymin/QDroid) to be widely used. To use it, market curators submit app binary files to the analyzer, and they can then obtain reports within a short amount of time. The QDroid reports can provide the number of crashes, the amount of

network traffic, CPU utilization, energy consumption, and compatibility in differently sized screens. Based on these reports, curators can decide whether to accept an app into the market or not. The proposed analyzer is constructed in two parts: dynamic exploration and verification of the three quality factors of a mobile app.

In the first part, dynamic exploration is used to identify app pages by executing the app. Generally, basic Monkey [14] is used to explore and test a large number of apps, but, to improve the coverage and efficiency of Monkey, we designed *Activity-Based-Monkey*, which interacts with each *Activity* rather than only considering a main entry point. As Activities are the main parts of mobile apps, an *Activity* roughly corresponds to a different screen or window in traditional GUI-based applications. Basic Monkey always starts the exploration in the *Main-Activity*, injects user-like GUI actions, and generates callbacks to invoke certain Activities. In contrast, the *Activity-Based-Monkey* starts the exploration in all Activities that can be independently invoked, regardless of the execution sequence. By doing so, the *Activity-Based-Monkey* enables exploration of as many app pages as possible within a reasonable time.

In the second part, verifying the three quality factors of a mobile app, the number of crashes, resource usage, and compatibility problems in differently sized screens are considered. These quality factors damage user experience with abnormal exits, large energy consumption, and poor design. While app pages are being explored by the *Activity-Based-Monkey*, the proposed Logger in QDroid automatically collects all data for each verification, such as exception type, CPU utilization, the amount of network traffic, energy consumption, and screenshot images in each *Activity*.

In contrast to an existing energy profiler [15], the proposed Summarizer in QDroid analyzes each quality factor by a devised algorithm without human involvement and then outputs an analysis report in which the number of crashes, reasons for each crash, app list of excessive resource usage, and Activity list of resolution problems are contained.

The goal of QDroid is not to completely eliminate verification by market curators. Instead, the aim is to substantially reduce the amount of manual effort that market curators must perform to remove low quality apps. By doing so, QDroid can help the market curators focus their time on the tasks that most require human involvement.

To prove the proposed approach, QDroid was implemented on a popular mobile platform and 67 apps collected from an official app market. QDroid was then evaluated by comparison with existing tools: basic Monkey [14] and Monkeyrunner [16].

From the experiment, QDroid discovered 78% more crashes and attained 23% higher Activity coverage (the number of screens explored) [12] than basic Monkey. When detecting compatibility problems in differently sized screens, QDroid attained a 17% lower false positive rate than Monkeyrunner. For discovering excessive resource usage, QDroid found 4 and 5 apps in CPU utilization and energy consumption, respectively.

The main contributions of this proposed approach and study can be summarized as follows:

(i) A new dynamic exploration, called *Activity-Based-Monkey*, is proposed. *Activity-Based-Monkey* identifies all Activities that can be independently invoked regardless of particular sequence and employs these Activities as entry points. This multiple entry points-based exploration enables as many app pages as possible to be visited within a reasonable time.

(ii) Evaluation schemes for each app quality factor are devised. These schemes enable a market curator to verify apps in terms of the presence of crashes, excessive resource usage, and compatibility problems in differently sized screens within a reasonable time and to efficiently reduce the number of steps requiring human judgment.

(iii) To demonstrate the proposed method, QDroid is compared to basic Monkey tools in terms of Activity coverage and effectiveness at discovering crashes. In addition, the amount of manual effort that can be reduced by QDroid is shown with regard to evaluating apps excessive resource usage and compatibility problems in differently sized screens.

2. Motivation and Design Requirement

The motivation for the study and development was originated from three aspects. The first aspect is the lack of information available for determining app quality from users perspective. App markets commonly provide five-star rating based on users' feedback as a quality indicator. However, previous works have revealed that such ratings are not relevant to app quality [9, 17, 18]. The major reasons are as follows. Users can rate apps with stars without any comments. This means that users can evaluate apps with only a moment's consideration. In addition, by filtering out any possible low reviews and giving aggressive incentive to users, developers can artificially increase the rating of their apps.

The second aspect is the inefficiency of existing approaches in the two representative app markets: Google's Play Store and Apple's App Store. Google and Apple have different approaches for managing app quality. Google does not verify app quality before app release. Instead, with their automatic tool, the curating company periodically removes apps which violate the Play Store's policy guidelines [19]. However, according to TechCrunch's report [20], Google has only removed 60,000 apps from the app market. In light of the 700,000 apps registered in that time [20], this number is too small to assume complete quality control. Moreover, users are consistently frustrated by low quality apps until the company's automatic tool is able to remove them. In contrast, Apple does not rely on automatic tools but has a small army of reviewers that make sure that each app meets the App Store review guidelines [21] before app release. Apple operates a more thorough review process than Google, but there are still problems. While Google does not charge developers to sell their apps, Apple has required developers to pay registration fee for such app reviews. Moreover, the human resource-oriented approach involves spending developer time.

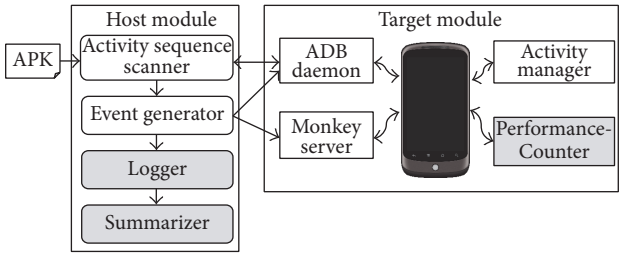

Figure 1: Overall architecture of the QDroid.

The third aspect is the lack of a comprehensive tool that can cover multiple quality factors. The two representative market operators, Google and Apple, only consider the security, privacy, and content of an app. Existing scalable dynamic approaches have focused on privacy [10] and bugs [9, 12, 13]. As far as is known, there is no scalable approach that can cover multiple quality factors such as excessive resource usage and compatibility in differently sized screens.

Considering these three aspects, there is a need for a system that satisfies the following requirements: firstly, it should reduce the manual effort for covering a large number of apps. In light of app submissions generated every day, it should be scalable. Secondly, the test procedure should be black-box based. App market curators do not have access to the source code because app developers submit compressed binary packages. Finally, it should complement existing methods by verifying excessive resource usage and compatibility problems in differently sized screens.

3. QDroid Architecture

The overall architecture of QDroid is shown in Figure 1. The whole structure consists of the host and the target module. The host module performs each quality analysis step and the target module is in charge of interacting with the host and measuring resource usage. The color of each component represents that certain functionality. The white colored components are in charge of the proposed dynamic exploration, *Activity-Based-Monkey*. The grey colored components are for verifying multiple factors.

To begin, market curators submit an app installer (or .apk file) to QDroid. QDroid then installs the app to an emulator and a smartphone. To generate different screen resolutions, the emulator is used. On the other hand, resource usage can be measured on a smartphone. Before the dynamic exploration, the *Activity-Sequence-Scanner* identifies no sequence Activities that can be independently invoked. To analyze an Activity sequence, the *Activity-Sequence-Scanner* employs Android Debug Bridge (ADB) and the *Activity-Manager*. ADB enables the host module to interact with the target, smartphone, and emulator. The *Activity-Manager* is in charge of controlling Activities.

Subsequently, the *Event-Generator* starts a dynamic exploration in no sequence Activities as entry points, injecting user-like GUI actions (tapping, typing text, callback, etc.) to interact with certain Activities. The *Monkey-Server* in the target module receives such events from the *Event-Generator* to operate the target.

While app pages are explored, the *Performance-Counter* continuously monitors the CPU utilization, amount of network traffic, and energy consumption of the app. At the same time, the *Logger* records all results during the entire exploration time. At the end, the *Summarizer* outputs an analysis report containing the number of crashes, reasons for each crash, a list of excessive resource usage instances, and an Activity list of resolution problems. The proposed dynamic exploration and verification schemes for multiple factors are, respectively, explained in Sections 4 and 5 in further detail.

4. Dynamic Exploration

In this section, the new dynamic exploration method, called *Activity-Based-Monkey*, is described. The existing dynamic approaches start the dynamic exploration in the *Main-Activity*. However, if many *Activities* are declared, the existing dynamic approaches spend a lot of time. Furthermore, it is technically difficult to explore all Activities. To overcome this limitation, the proposed dynamic approach distinguishes no sequence Activity in order to employ start point. Such no sequence Activity does not require any parameter from previous steps, such as database cursor and user inputs.

The proposed dynamic exploration consists of two steps. The first step is to reveal sequence dependency among *Activities*. The second step is to explore the app using particular events such as tapping, buttons click, and swipe. In contrast to the traditional Monkey, this exploration is started in each no sequence Activity. Each step is explained in the following.

The first step is to identify Activity sequences. This step is named Activity-Sequence-Scanning. Algorithm 1 is a Dynamic-Activity-Sequence-Scanning algorithm. This algorithm takes a list L of n Activities defined in `Manifest.xml` as an input. All Activities are declared in `Manifest.xml`. Such `XML` file can easily be extracted from the app installer (.apk) by decompression. The algorithm produces as output a list G of no sequence Activities it identifies. The more detailed process is as follows. At first, an *Activity* is picked from the list L if it is not empty. The algorithm then invokes a certain *Activity* to be shown on the screen. If this invocation is successful, this *Activity* is determined to be nonsequential. However, certain Activities may produce a crash due to the dependency of parameters. In that case, the algorithm terminates this *Activity*. This process is repeated until the list L is empty.

The second step is Event-Generation. This step explores app pages based on random UI events such as tapping, typing text, trackball, and navigator. In this paper, the UI extraction and the page equivalence that are provided by the smart Monkey [9, 11] are not considered. The *Event-Generator* simply feeds UI events to an app according to the number of events and types of UI events the user configures. Algorithm 2 details the operation of the *Event-Generator*. This algorithm takes the number of events and a list G of no sequence Activities as the input. At first, an *Activity* is picked from list G if this list is not empty. The algorithm then invokes

```
Input: List L of n Activities
Output: List G of no sequence Activities
(1) G ← {}
(2) e ← false
(3)  for Activity is remained in L do
(4)      a pick an Activity from List L
(5)      e ← startActivity(a)
(6)      if e is true then
(7)          append a to G
(8)          e ← false
(9)      else
(10)         forceStopActivity(a)
(11)     end
(12)  end
```

Algorithm 1: Dynamic activity sequence scanning.

```
Input: Number of events N, List of Activities G
(1)  while not all no sequence Activities are explored do
(2)      a ← pick an Activity from G
(3)      startActivity(a)
(4)      for i ← i + 1 to N do
(5)          e ← generate a new event
(6)          perform e
(7)      end
(8)  end
```

Algorithm 2: Event generation.

this *Activity*. The program feeds the UI event to this *Activity* according to the number of events the user configures. This process is repeated until list G is empty.

5. Verifying Multiple Quality Factors

This section describes each factor affecting app quality. Firstly, an explanation is provided for why these factors are chosen and how QDroid tests these factors. QDroid currently verifies five factors: number of crashes, CPU utilization, the amount of network traffic, energy consumption, and compatibility in differently sized screens. QDroid separates the verification code from the app page exploration code. This design has made it easy to extend to new verifications. Next, each quality factor is elaborated in more detail.

5.1. Crash. App crashes damage the user experience by incurring abnormal exits. Such crashes are caused by GUI bugs, network conditions, geolocation, and so forth. In this paper, QDroid is focused on detecting crashes by GUI bugs. The proposed tool is based on the logging messages in the mobile system used to detect crashes. Based on exception type and stack trace, QDroid counts total crashes. By doing that, the number of crashes detected is not exaggerated.

To reproduce crash, all GUI events generated by *Activity-Based-Monkey* are recorded. Also, we record delay time between two GUI events. With these events and timing information, we can revisit a certain crash. Not only can market curators decide whether a detected crash leads to significant degradation of a user experience by reproducing it, but also developers can improve their app quality.

```
Input: List L of n Activities
Output: List G of images
(1) resize images to the same resolution
(2) convert images to gray-scale
(3) normalize two images
(4) m ← manhattan-norm
(5) z ← zero-norm
(6) if m ≥ mThreshold ∧ z ≥ zThreshold
      then
(7)       return true
(8) else
(9)       return flase
(10) end
```

Algorithm 3: Image comparison.

5.2. Resource Usage. The proposed verifier automatically measures CPU utilization, the amount of network traffic, and the energy consumption of the app running on the smartphone. To measure resource usage during the app exploration process, a software based Performance-Counter running on the application layer in the mobile system was developed. To determine excessive usage, apps are grouped together by category. Then, outlier detection [22] is performed using a box plot. Finally, this verifier can provide a market curator with a list of outliers, and the curator can further examine those apps. This verifier cannot completely diagnose excessive resource usage. Rather, this tool aims to reduce the amount of work that market curators must perform.

5.3. Compatibility Problem in Differently Sized Screens. QDroid verifies that an app provides proper resolution in differently sized screens to ensure a quality user experience. According to Open Signal's report [23], mobile apps are being executed in more than 3,997 types of devices with more than 250 screen resolution sizes. In light of the number of diverse screen resolutions, resolution optimization is a daunting task for individual developers.

Resolution problems in differently sized screens are induced by the following three major causes: firstly, developers use physical units instead of logical units. Secondly, developers do not provide image resources optimized for high and low density screens. Thirdly, developers use absolute layout to arrange UI components rather than relative layout.

To detect suspicious apps having such problems, an image comparison algorithm was devised as shown in Algorithm 3. At first, this algorithm runs a certain app in two different resolution screens. Meanwhile, the proposed verifier collects each screenshot image according to each *Activity*. This algorithm takes the obtained screenshot images in differently sized screens as input. This algorithm produces as output a list G of images that need to be reviewed by the market curator.

To compare with a variety of image resolutions, different resolution images are resized to the same scale and converted

to grayscale. For direct comparison, two converted images are normalized to a range of pixel intensity values. After that, each image is calculated by applying two norms of linear algebra, *Manhattan* and *Zero*. The *Manhattan-norm* (the sum of the absolute values) presents how much the image is off. The *Zero-norm* (the number of elements not equal to zero) shows how many pixels differ. Finally, if the two values are greater than two thresholds set up by the user, the image comparison algorithm checks this image pair as a suspicious resolution problem. If not, nothing happens. The number of suspicious images can be controlled by adjusting the two thresholds.

6. Implementation

This section describes the implementation of QDroid. The entire QDroid consists of 11.7k lines of code, broken into host module 9.3 kloc, *Performance-Counter* 2.2 kloc, and modified parts of the mobile platform 0.1 kloc. Next, an explanation in further detail is given for how each module is developed.

6.1. Dynamic Exploration. The dynamic exploration is implemented in two parts: an Activity-Sequence-Scanner and an *Event-Generator*. This code is written by Python and host side accounts for 80% of total codes. For automated analysis, the install, uninstall, screen unlock, and reboot functions are implemented in the system. Basically, ADB and Monkey-Server provide the functionalities to control target devices. With the Python script, two tools are combined for automation. To implement the Dynamic-Activity-Sequence-Scanning algorithm, the *Activity-Manager* is employed. The *Activity-Manager* enables QDroid to launch certain Activities via ADB. This process is implemented by Python script. To develop the Event-Generator algorithm, the existing basic Monkey was modified. By doing that, QDroid is able to generate both specific and random events. The *Event-Generator* starts the exploration in a certain *Activity* rather than only starting at *Main-Activity*.

6.2. Performance-Counter. This component aims to profile hardware usage generated by dynamic exploration. To capture hardware usage in different levels, the Performance-Counter was implemented using C and Java. Figure 2 shows the internal structure of the Performance-Counter. The Performance-Counter profiles CPU utilization, network traffic, and energy consumption.

To extract CPU utilization and energy consumption from the Linux kernel, the `getCPUUsage` and `getPowerInfo` functions were developed by Java Native Interface (JNI). The `Proc` file system contains `jiffies` that are cumulative tick counts after system boot. The Performance-Counter can calculate CPU utilization by the following formulas:

$$\begin{aligned} \text{Total}_{\text{jiffies}} &= \text{idle}_{\text{jiffies}} + \text{use}_{\text{jiffies}} + \text{system}_{\text{jiffies}} \\ &\quad + \text{low}_p\text{riori}_{\text{jiffies}}, \\ \text{Utilization} &= \left(1 - \frac{\left(\text{idle}_{\text{jiffies}}\right)}{\left(\text{Total}_{\text{jiffies}}\right)}\right) \times 100\%. \end{aligned} \tag{1}$$

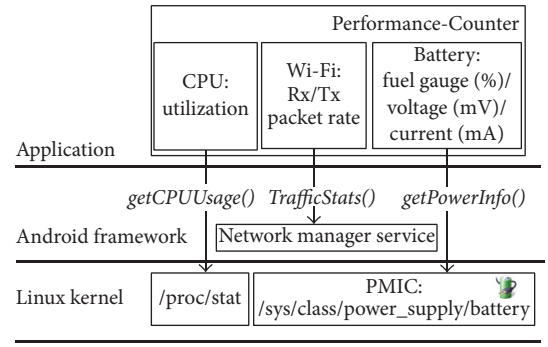

Figure 2: The structure of the Performance-Counter for measuring resource usage.

To compute energy consumption, the `getPowerInfo` function reads battery level, current, and voltage from the Power Management Integrated Circuit (PMIC). On the other hand, the network traffic can be known through `TrafficStats` API in the Android framework.

6.3. Modified Android. To make a completely automated process and some particular functions, 122 Lines of Code (LOC) in the platform components were modified. The modified mobile platform components are `Activity Manager`, `System Server`, `Monkey`, and `ADB`. To analyze the Activity sequences automatically, the `Activity Manager` was slightly changed to terminate an *Activity* which is frozen due to dependent parameters. The `system server` was also modified to keep the setting when the mobile system is rebooted during analysis. To implement the *Event-Generator* as mentioned in Section 4, Android `Monkey` was modified to start exploration in a certain *Activity*. `ADB` was also changed to keep the connection between host and target over Wi-Fi network. With network connection, multiple devices can interact with host PC.

6.4. Logger. The Logger developed by Python is in charge of recording analysis results. Such results constitute the mobile framework and kernel information. The Logger records the following information: list of no sequence Activities, list of explored Activities, screenshot images, deterministic events to reach specific *Activity*, exception and call stack for crash, and elapsed time for analysis.

6.5. Summarizer. The Summarizer developed by Python provides a final report based on the results of dynamic exploration. The Summarizer determines the number of unique crashes using exception and call stack recorded during the dynamic exploration. It receives resource usage from the *Performance-Counter* and then it calculates the mean of resource usage. Finally, the Summarizer finds resolution problems among the collected image pairs using the image comparison algorithm and it calculates Activity coverage.

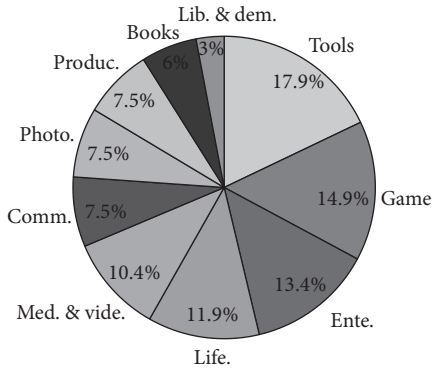

Figure 3: Distribution of collected apps.

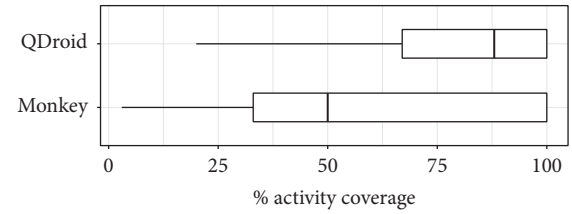

Figure 4: Activity coverage achieved by QDroid and Android Monkey.

Table 1: Fraction of crash corresponding to exception type.

Rank (fraction)	Exception
1 (51.2%) 21	nullpointerexception
2 (14.6%) 6	activitynotfoundexception
3 (12.2%) 5	illegalstateexception
4 (9.8%) 4	Activity Not Response (ANR)
5 (2.4%) 1	cursorindexoutofboundsexception
5 (2.4%) 1	arrayindexoutofboundsexception
5 (2.4%) 1	database.sqlite.sqliteexception
5 (2.4%) 1	securityexception
5 (2.4%) 1	unsatisfiedlinkerror

7. Evaluation

QDroid was evaluated by comparing it to the fuzzing tool called Monkey, for Activity coverage and the number of crashes discovered during dynamic exploration. In addition, QDroid was evaluated in terms of how much effort could be reduced while detecting excessive resource usage and compatibility problems in differently sized screens.

7.1. Methodology. 67 apps were randomly selected from the official app market. As shown in Figure 3, they are from sufficiently diverse categories. QDroid was evaluated on these apps to determine whether it could outperform existing tools and how much effort it could reduce. All experiments were done on Linux machines with 16 GB memory and an i7 processor. To measure resource usage, the HTC Nexus One smartphone was used.

For measuring Activity coverage and discovering app crashes, *Activity-Based-Monkey* and basic Monkey were executed on each app, respectively. To fairly compare with them, each tool was run with 2,000 random events. Additionally, QDroid automatically removes redundant crashes based on exception type and stack trace. In contrast to QDroid, basic Monkey does not eliminate the redundancy.

Respective automation capabilities for detecting excessive resource usage and compatibility problems were evaluated. Since QDroid automatically profiles resource usage during dynamic exploration, there is no extra process for detecting abnormal performance. To detect compatibility problems in differently sized screens, however, this experiment needed to gather screenshot images in each different screen. For that reason, each app was run twice on an emulator of different resolutions. The first configuration in the emulator was 720 × 1280 resolutions and 320 densities. The other configuration was 480 × 800 resolutions and 240 densities. To draw the same GUI components, the two emulators were run on the same mobile platform version, 4.0.3.

7.2. Activity Coverage and Discovering App Crashes. Among the 67 apps, 41 and 23 unique crashes were discovered by QDroid and basic Monkey, respectively. These results show that QDroid outperforms basic Monkey by nearly two times. This result obviously demonstrates that QDroid is better at detecting crashes than the existing tool. There is a reason for this improvement. In contrast to QDroid, basic Monkey always starts the exploration in *Main-Activity*. For that reason, the app crashes by the same bug during exploration. This phenomenon is known as the *Pesticide Paradox* in software testing [24]. To detect various crashes in several Activities, exploration is started at different entry points.

As shown in Table 1, the detected crashes are distributed in 9 categories. Interestingly, over 50% of the crashes are attributed to `java.lang.nullpointerexception`. To prevent such app crashes, the developer should focus on `null-point-error` testing before app submission. Over 14% of the crashes are caused by `activitynotfoundexception`. Commonly, `activitynotfoundexception` occurs when no apps are relevant to a particular request. For example, `Facebook` does not have an e-mail function, but `Facebook` can send an e-mail by forwarding this request to a certain e-mail app. If no e-mail app is installed, `activitynotfoundexception` is produced by this request. The remaining fraction of crashes stems from several causes, such as database, security, or freezing.

7.3. Coverage Result. Figure 4 shows Activity coverage for the 67 apps by QDroid and basic Monkey. Basic Monkey covered 3–100% of Activities, for an average of 57%. QDroid attained 20–100% of Activities, for an average of 80%. QDroid outperforms Monkey, achieving substantially higher coverage for 32 of the 67 apps. In 25 out of 32 apps, QDroid even achieves two times higher Activity coverage than basic Monkey. In three apps (`Finger Tap Piano`, `WolframAlpha`, and `BIG Launcher`), QDroid led to improvements of 11.4x, 13.5x, and 25.3x higher Activity coverage, respectively. These results

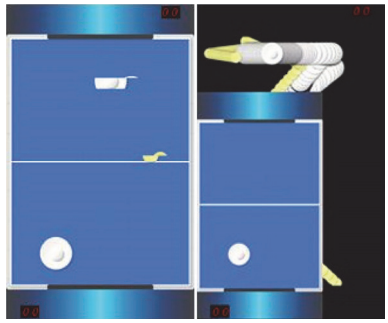

Figure 5: Example of compatibility problem image.

can be attributed to the use of multiple entry points, which only *Activity-Based-Monkey* can generate. In 15 out of 67 apps, however, QDroid did not make an improvement in Activity coverage. This is because QDroid strongly hinges on the number of entry points. In other words, if almost all Activities have a particular execution sequence to be invoked, *Activity-Based-Monkey* will also start the exploration in *Main-Activity*. In that case, there is no advantage to QDroid. Apart from using multiple entry points, *Activity-Based-Monkey* and basic Monkey are the same in that respect.

7.4. Resolution Problem. The effectiveness of the image comparison algorithm was evaluated by comparing it to Monkeyrunner [16]. *Activity-Based-Monkey* collected 206 screenshot image pairs for each *Activity* on two different screens. Then, resolution problems were detected among these image pairs manually.

Out of 67 apps, 3 apps have resolution problem. These apps are `Digger`, `Air Hockey`, and `Saybebe`. In the 3 apps, 10 images have a resolution problem. Figure 5 shows the resolution problem image of `Air Hockey`. Resolution problem images have a black area or the wrong sized GUI component.

To demonstrate effectiveness, QDroid was compared to the mobile platforms basic comparison tool, Monkeyrunner. For image comparison, the developer employs `sameAs` function in Monkeyrunner. `sameAs` takes a float as input. This float indicates the percentage of pixels that need to be the same. The default is 1.0, indicating that all the pixels must match. The proposed algorithm focuses on preventing false negatives instead of completely detecting all resolution problems.

Before comparing the proposed algorithm to Monkeyrunner, each algorithm's optimal threshold was investigated to ensure no false negative. Monkeyrunner's threshold is 0.66. This means that 34% of the distance between two images is allowed to be equal. The proposed algorithm additionally considers color differences between two images. Zero-norm's threshold is 0.67, similar to Monkeyrunner, and Manhattan-norm's threshold is 24.

The experimental results for the two methods are shown in Table 2. Monkeyrunner and the proposed algorithm attained 89 and 53 false positives, respectively. The proposed algorithm led to a 17% improvement in the false positive rate. Eventually, with the proposed tool, a market curator would have reviewed only 63 out of 206 image pairs. The reasons that these false positives occurred can be grouped into the following three categories.

Table 2: Results of the image comparison algorithm among 206 image pairs.

Method	# of image pairs	TP	FP	FN
Monkeyrunner `sameAs()`	206	10	89	0
The proposed algorithm	206	10	53	0

The first reason is dynamic contents. An app can represent different contents every time it is launched. For example, for `All-in-one-toolbox` in a productivity category, some of the Activities represent the hardware usage of the CPU, memory, and so forth, and such information is not consistent. In another example, `Highway-Racing` app deploys different vehicles every time the game starts. Moreover, the status bar and built-in GUI component also have dynamic contents such as battery level, time in the status bar, and radio connectivity. Therefore, such dynamic contents can induce false positives.

The second reason is advertisement. Many free apps contain ad-related Activities. For example, in `Music Searcher` app, all the Activities have ad-related parts. In general, free apps contain more Activities than paid apps. Ads present different content every time an *Activity* is launched. For that reason, ads are the primary reason for false positives.

The last reason is resizing. To compare with images among differently sized resolutions, images are resized to be the same resolution using the Python library. However, this resizing creates a gap between the mobile system and resizing. This is because the mobile system precisely deploys GUI components in the screen according to layout, but the resizing simply decreases dot-per-inch (DPI) to meet a particular resolution.

7.5. Resource Usage. QDroid automatically measured resource usage during dynamic exploration. CPU utilization, the amount of network traffic, and energy consumption are considered to be resource usage. Figure 6(a) shows the distribution of CPU utilization on 67 apps. The minimum, maximum, and average distribution are 14%, 94%, 46%, respectively. Eventually, this distribution is evenly spread. An app having high CPU utilization can hinder the response time of other apps. For that reason, a market curator should review such apps in the registration process to ensure a quality user experience. Figure 6(b) shows the distribution of the network traffic, the log scale. One of the apps produced more than 8000 Kb network traffic. There are some reasons for this. The first one is mobile ads. They can generate a lot of network traffic to show the ads. The second one is web-based content. To ensure app compatibility among different platforms, many developers employ web-based contents. Because of the data fee burden on users, a market curator should provide network traffic information via testing. In addition, frequently transmitting data over 3G or Wi-Fi leads to early battery depletion [25]. Figure 6(c) shows the distribution of energy consumption. This distribution ranges from 1144 mW

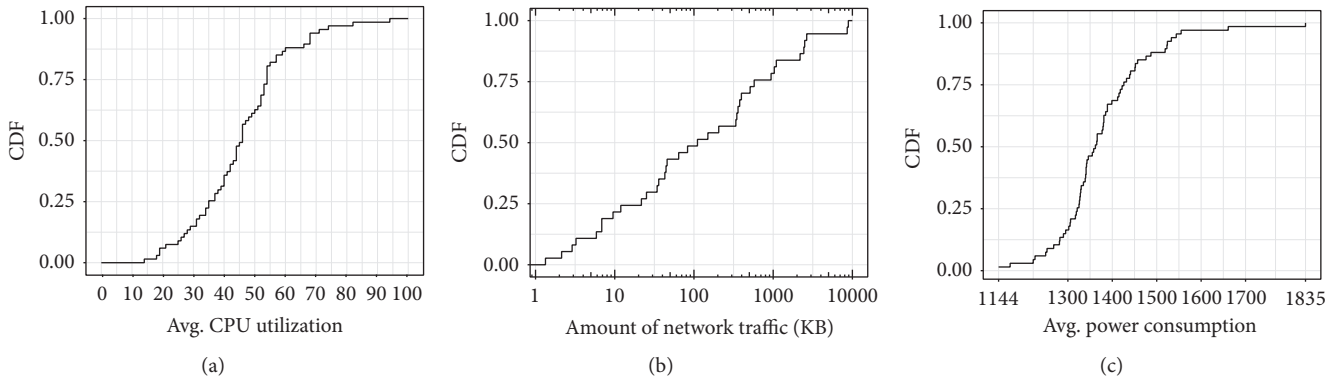

Figure 6: The distribution of resource usage in 67 apps; (a) CPU utilization; (b) network traffic; (c) power consumption.

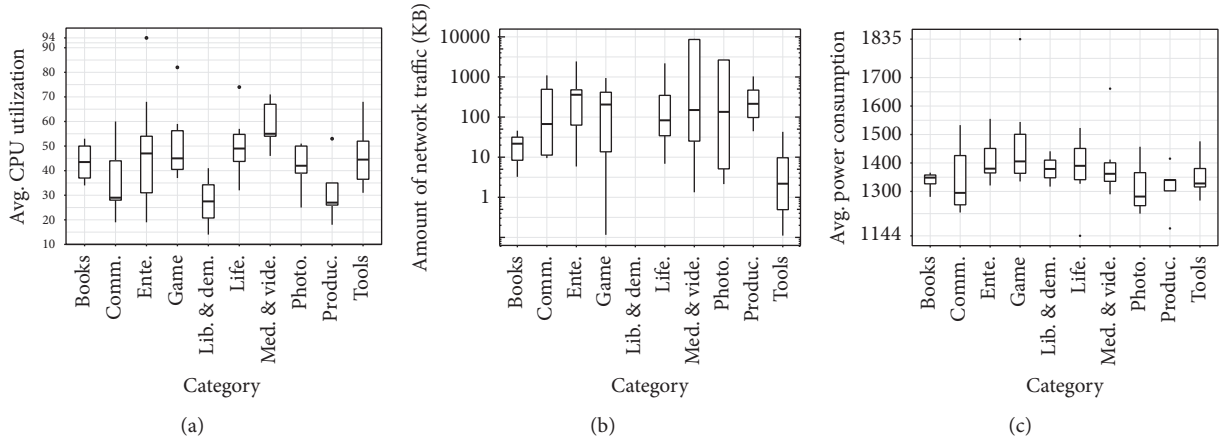

Figure 7: App performance outliers detected among 67 apps; the box plots of (a) CPU utilization, (b) network traffic, and (c) power consumption.

to 1835 mW. The average power consumption is 1378 mW and this distribution is evenly spread. Energy consumption is caused by CPU utilization and network traffic [26]. Also, anomaly energy consumption is caused by malicious apps [27] and wake-lock deactivation [28]. Market curators have to test energy issue to prevent it because high energy consumption restricts the duration of smartphone usage.

To find abnormal behavior, resource usage in each app was analyzed corresponding to each category. QDroid discovered outliers based on resource usage in the same category. These outliers need to be reviewed by the market curator, since their behavior is not normal compared to other apps. To determine outliers in the same category distribution, QDroid used a box plot method. The box plot is a useful graph for describing the behavior of the data in the middle as well as at the ends of the distributions.

The box plot uses the median and the lower and upper quartiles (defined as the 25th and 75th percentiles). If the lower quartile is $Q1$ and the upper quartile is $Q3$, then the difference $(Q3 - Q1)$ is called the interquartile range or IQR. A box plot is constructed by drawing a box between the upper and lower quartiles with a solid line drawn across the box to locate the median. The following quantities (called whisker) are needed for identifying extreme values in the tails of the distribution: lower whisker defined by $Q1 + 1.5 \times \text{IQR}$ and upper whisker defined by $Q3 + 1.5 \times \text{IQR}$. A point beyond two kinds of whiskers is considered as an outlier.

This paper focused solely on the upper whisker for outliers to detect abnormal behavior. The box plots of each resource usage are shown in Figure 7. Black circles in Figure 7 refer to outliers which are distant from normal resource usage. As shown in Figure 7(a), there are four total outliers in four different categories. In the network case, Figure 7(b), no outlier appears in any category. As shown in Figure 7(c), there are five total outliers in four different categories. Out of 5 outliers, 2 outliers are the same as CPU utilization outliers. That means that the root cause of the high energy consumption correlates with the underlying hardware usage. QDroid does not guarantee that all outliers are related to abnormal behavior. Since the goal is to reduce the number of apps needing human review, however, outlier based analysis is meaningful.

8. Related Work

In this paper, a novel app quality analysis framework based on a dynamic exploration and verification method is proposed. In this section, QDroid is compared to prior works for app page exploration and app quality verification.

Table 3: Recent tools for mobile application testing.

Tool name	Developed	Req.	Checked quality measures		
			Properties	Resource	Resolution
AMC [11]	Academy	Binary	Accessibility	—	—
VanarSena [9]	Academy	Binary	Crashes	—	—
DynoDroid [12]	Academy	Binary	Crashes	—	—
Caiipa [13]	Academy	Binary	Crashes	CPU, network, energy	—
Monkey [14]	Industry	Binary	Crashes	—	—
Monkeyrunner [16]	Industry	Script	Crashes	—	Auto
Robotium [29]	Industry	Script	Crashes	—	—
Robotium-Recorder [30]	Industry	Record	Crashes	—	—
TestDroid [31]	Industry	Record	Crashes	CPU, memory	Manual
Firebase-Robo-Test [32]	Industry	Binary	Crashes	—	Manual
QDroid	Academy	Binary	Crashes	CPU, network, energy	Auto

8.1. Mobile Application Page Exploration. The software testing research group has presented various techniques to efficiently explore program states. Several prior works have statically explored source code or binary code to discover energy bugs [6], app plagiarism [8], security problems [33], and privacy leaks [7, 34]. However, static exploration approaches are restricted by many features such as source code dependency, code obfuscation, native libraries, and a complex SDK framework.

Model-based exploration approaches [35–37] are another popular method. These approaches can test an app according to a prespecified model of GUI states and transitions. However, model-based exploration requires a developer to provide a GUI model of the app or source code. Providing such a detailed model is an onerous task that requires substantial knowledge of the application's behavior.

To address these limitations, several recent works [10–13] have focused on dynamic exploration for verifying the runtime properties of an app. However, these approaches are relatively more time-consuming during exploration. To improve exploration speed, grey-box approaches [9, 38] have been proposed which combine static and dynamic exploration or use binary instrumentation. The proposed exploration approach also employs the grey-box strategy to improve speed. No sequence Activities that can be independently invoked are identified before the exploration. By doing so, QDroid can start exploration in multiple entry points.

8.2. Mobile Application Verification. As mentioned earlier, two representative app markets, operated by Google and Apple, have managed app quality using their own approaches and have shown many limitations. As shown in Table 3, several researchers and companies have proposed a variety of testing tools for app quality verification. All the tools in Table 3 are based on dynamic exploration instead of static approach. This is because dynamic exploration is more applicable to the app markets as described earlier.

Those tools are categorized into two types: academic and industrial software. At first, AMC [11], VanarSena [9], DynoDroid [12], and Caiipa [13] were developed by universities as research projects. AMC [11] evaluated the suitability of UI properties to be used in vehicular apps. VanarSena [9] and DynoDroid [12] detected crashes by inducing faults or injecting UI and system events, respectively. Moreover, Caiipa [13] discovered new crashes by exposing apps to various external exceptional conditions, such as network conditions and location change. Unlike QDroid, none of these research works consider multiple quality factors which are excessive resource usage and compatibility in differently sized screens.

Next, Monkey [14], Monkeyrunner [16], Robotium [29], Robotium-Recorder [30], TestDroid [31], and Firebase-Robo-Test [32] were developed by open-source groups and companies as industrial testing tools. The Monkey and the Monkeyrunner tools are provided from the Android SDK. The Monkey [14] generates random events, including touch-screen presses, gestures, and other system-level events, and feeds them into a mobile app. However, the Monkey shows poor Activity coverage because it is quite naive testing. The Monkeyrunner [16] drives an Android device or an emulator according to a script written by a user. Robotium [29] is a test framework based on Junit for writing testing cases for an Android app. Robotium also executes an Android app according to a script a user can write similar to Monkeyrunner. Both tools heavily depend on manual test scripts. It is a daunting task to write scripts with regard to a large number of apps. In addition, these tools do not support multiple quality verifications.

Robotium-Recorder, TestDroid, and Firebase-Robo-Test are commercial tools for mobile app testing. Robotium-Recorder [30] is a record-and-replay tool, which allows users to record Robotium test cases and replay them on an Android device. However, record-and-replay testing is still a daunting task due to the number of registered apps on a daily basis. TestDroid and Firebase-Robo-Test are cloud-based infrastructures, which provide a number of mobile devices hosted in a datacenter for testing Android apps. TestDroid [31] requires a user to record events for replaying them on

thousands of mobile devices. Unlike TestDroid, Firebase-Robo-Test [32] can test mobile apps by just submitting an app binary similar to QDroid. With one operation, users can initiate testing of their apps across a wide variety of devices and device configurations. However, these cloud-based tools also do not consider multiple quality factors. Moreover, in light of a number of apps, all the commercial tools can charge enormous cost because the cost is calculated based on the number of used devices and usage time. Therefore, even though there are a variety of testing tools, market curators can benefit from QDroid to filter out low quality apps in a timely fashion.

9. Conclusion

In this paper, QDroid was presented, a tool that explores apps pages and verifies the presence of crashes, excessive resource usage, and compatibility problems in differently sized screens with no need for source codes. QDroid enables market curators to maintain market quality by removing flawed apps in a timely fashion. To demonstrate the effectiveness of QDroid, QDroid evaluated 67 apps in terms of crash detection, Activity coverage, and discovering excessive resource usage and compatibility problems. The results were then compared to basic Monkey. The experimental results showed that QDroid found more crashes and visited more Activities than basic Monkey. In addition, QDroid can substantially reduce manual efforts to test apps.

In the future, QDroid will be extended to include additional quality factors, to cover the full range. In addition, QDroid will be combined with a real user interaction model. By doing so, a useful quality standard can be devised, allowing users to choose the most adequate app for each particular situation.

Competing Interests

The authors declare that they have no competing interests.

Acknowledgments

This research was supported by the Basic Science Research Program through the National Research Foundation (NRF) of Korea funded by the Ministry of Education (2014R1A1A2059669) and the Human Resource Training Program for Regional Innovation and Creativity through the Ministry of Education and NRF of Korea (2014H1C1A1066721).

References

[1] A. I. Wasserman, "Software engineering issues for mobile application development," in *Proceedings of the FSE/SDP Workshop on the Future of Software Engineering Research (FoSER '10)*, pp. 397–400, ACM, November 2010.

[2] M. E. Delamaro, A. M. R. Vincenzi, and J. C. Maldonado, "A strategy to perform coverage testing of mobile applications," in *Proceedings of the ACM International Workshop on Automation of Software Test (AST '06)*, pp. 118–124, Shanghai, China, May 2006.

[3] H. Muccini, A. Di Francesco, and P. Esposito, "Software testing of mobile applications: challenges and future research directions," in *Proceedings of the 7th International Workshop on Automation of Software Test (AST '12)*, pp. 29–35, IEEE, Zurich, Switzerland, June 2012.

[4] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2013.

[5] AppBrain, "Percentage of low quality apps," http://www.appbrain.com/stats/number-of-android-apps.

[6] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps," in *Proceedings of the 10th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys '12)*, pp. 267–280, ACM, Lake District, UK, June 2012.

[7] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androi-dleaks: automatically detecting potential privacy leaks in android applications on a large scale," in *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (TRUST '12)*, pp. 291–307, Springer, Vienna, Austria, 2012.

[8] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: detecting cloned applications on android markets," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS '12)*, pp. 37–54, Pisa, Italy, September 2012.

[9] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan, "Automatic and scalable fault detection for mobile applications," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '14)*, pp. 190–203, ACM, Bretton Woods, NH, USA, June 2014.

[10] W. Enck, P. Gilbert, S. Han et al., "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems*, vol. 32, no. 2, pp. 5 : 1–5 : 29, 2014.

[11] K. Lee, J. Flinn, T. J. Giuli, B. Noble, and C. Peplin, "AMC: verifying user interface properties for vehicular applications," in *Proceedings of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '13)*, pp. 1–12, ACM, Taipei, Taiwan, June 2013.

[12] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for android apps," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE '13)*, pp. 224–234, ACM, August 2013.

[13] C.-J. M. Liang, N. D. Lane, N. Brouwers et al., "Caiipa: automated large-scale mobile app testing through contextual fuzzing," in *Proceedings of the 20th ACM Annual International Conference on Mobile Computing and Networking (MobiCom '14)*, pp. 519–530, ACM, Maui, Hawaii, USA, September 2014.

[14] Google, "Ui/application exerciser monkey," http://developer.android.com/tools/help/monkey.html.

[15] M. Kim, J. Kong, and S. W. Chung, "Enhancing online power estimation accuracy for smartphones," *IEEE Transactions on Consumer Electronics*, vol. 58, no. 2, pp. 333–339, 2012.

[16] Google, Monkeyrunner/monkeyimage, http://developer.android.com/tools/help/MonkeyImage.html.

[17] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '14)*, pp. 221–233, ACM, 2014.

[18] B. Stegner, "Why you should not trust app ratings on google play," http://www.makeuseof.com/tag/shouldnt-trust-app-ratings-google-play/.

[19] Google, Playstore's policy guidelines and practices, https://support.google.com/googleplay/android-developer/answer/113474?hl=en&rd=1.

[20] S. Perez, "Nearly 60 k low-quality apps booted from google play store in February, points to increased spam-fighting," http://techcrunch.com/.

[21] Apple, "App store review guidelines," https://developer.apple.com/app-store/review/guidelines/.

[22] W. Chang, *R Graphics Cookbook*, O'Reilly Media, Sebastopol, Calif, USA, 1st edition, 2013.

[23] O. Signal, Signal reports, http://opensignal.com/reports/.

[24] A. P. Mathur, *Foundations of Software Testing*, Addison-Wesley Professional, Boston, Mass, USA, 1st edition, 2008.

[25] Y. Cho, O. Mikhail, Y. Paek, and K. Ko, "Energy-reduction offloading technique for streaming media servers," *Mobile Information Systems*, vol. 2016, Article ID 7462821, 7 pages, 2016.

[26] A. Rice and S. Hay, "Measuring mobile phone energy consumption for 802.11 wireless networking," *Pervasive and Mobile Computing*, vol. 6, no. 6, pp. 593–606, 2010.

[27] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and S. Pastrana, "Power-aware anomaly detection in smartphones: an analysis of on-platform versus externalized operation," *Pervasive and Mobile Computing*, vol. 18, pp. 137–151, 2015.

[28] Y. Liu, C. Xu, S. C. Cheung, and J. Lü, "Greendroid: automated diagnosis of energy inefficiency for smartphone applications," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 911–940, 2014.

[29] Robotium, "User scenario testing for android," 2016, https://github.com/RobotiumTech/robotium.

[30] Robotium, Robotium recorder, 2016, http://robotium.com/.

[31] TestDroid, Testdroid, 2016, http://testdroid.com/.

[32] Google, "Firebase test lab for android robo test," 2016, https://firebase.google.com/docs/test-lab/robo-ux-test/.

[33] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: whitebox fuzzing for security testing: SAGE has had a remarkable impact at Microsoft," *Queue*, vol. 10, no. 1, pp. 20–27, 2012.

[34] B. Livshits and J. Jung, "Automatic mediation of privacy-sensitive resource access in smartphone applications," in *Proceedings of the 22nd USENIX Security Symposium*, pp. 113–130, Washington, DC, USA, August 2013.

[35] R. C. Bryce, S. Sampath, and A. M. Memon, "Developing a single model and test prioritization strategies for event-driven software," *IEEE Transactions on Software Engineering*, vol. 37, no. 1, pp. 48–64, 2011.

[36] X. Yuan and A. M. Memon, "Generating event sequence-based test cases using GUI runtime state feedback," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 81–95, 2010.

[37] X. Yuan, M. B. Cohen, and A. M. Memon, "GUI interaction testing: incorporating event context," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 559–574, 2011.

[38] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," in *Fundamental Approaches to Software Engineering*, vol. 7793, pp. 250–265, Springer, Berlin, Germany, 2013.