2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

Mimic: UI Compatibility Testing System for Android Apps

Taeyeon Ki, Chang Min Park, Karthik Dantu, Steven Y. Ko, Lukasz Ziarek
Department of Computer Science and Engineering
University at Buffalo, The State University of New York
Email: {tki, cpark22, kdantu, stevko, lziarek}@buffalo.edu

***Abstract*—This paper proposes Mimic, an automated UI compatibility testing system for Android apps. Mimic is designed specifically for comparing the UI behavior of an app across different devices, different Android versions, and different app versions. This design choice stems from a common problem that Android developers and researchers face—how to test whether or not an app behaves consistently across different environments or internal changes. Mimic allows Android app developers to easily perform backward and forward compatibility testing for their apps. It also enables a clear comparison between a stable version of app and a newer version of app. In doing so, Mimic allows multiple testing strategies to be used, such as randomized or sequential testing. Finally, Mimic programming model allows such tests to be scripted with much less developer effort than other comparable systems. Additionally, Mimic allows parallel testing with multiple testing devices and thereby speeds up testing time. To demonstrate these capabilities, we perform extensive tests for each of the scenarios described above. Our results show that Mimic is effective in detecting forward and backward compatibility issues, and verify runtime behavior of apps. Our evaluation also shows that Mimic significantly reduces the development burden for developers.**

***Keywords*—Mobile apps; UI compatibility testing; Parallel testing; Programming model;**

I. Introduction

This paper proposes Mimic, an automated UI compatibility testing system for Android. It supports what we call *follow-the-leader* model of testing—multiple testing devices are used in parallel but one device becomes a "leader" that performs a sequence of UI actions. All other devices follow the leader and perform the same sequence of UI actions. Using this testing model, Mimic reports UI compatibility problems occurred during a testing run, such as different UI paths taken, different UI structures displayed, different exceptions thrown, differences in UI performance, etc. In essence, the main focus of Mimic is *UI compatibility testing*.

This design choice of Mimic stems from several testing needs and the lack of a practical testing system that meets those needs. In particular, there are four common testing scenarios that call for UI compatibility testing as we detail in Section II—(i) *version compatibility testing*, where developers test their apps on different Android API versions, (ii) *device compatibility testing*, where developers test their apps on different Android devices, (iii) *third-party library testing*, where developers test new versions of third-party libraries with their existing apps, and (iv) *instrumentation testing*, where mobile systems researchers test the correctness of their bytecode instrumentation techniques [32], [17], [19] by comparing instrumented apps to original apps. All of these scenarios require UI compatibility testing, i.e., testers want to test and compare how the UIs of apps display and behave across different environments. We further detail each of these scenarios and the need for UI compatibility testing in Section II.

Mobile testing has several unique challenges including UI version compatibility and consistency across devices, app and OS versions. Mimic addresses the above challenges by providing the following two main features — an (i) easy-to-use *programming model* specifically designed for UI compatibility testing, and (ii) a *runtime* that manages multiple devices and app or Android versions. As mentioned earlier, it implements follow-the-leader testing model. The runtime also captures visual differences of an app's UI across different versions or devices using image processing techniques.

To the best of our knowledge, there is no previous work that focuses on UI compatibility testing for mobile apps. As we discuss in Section VII, existing systems, such as DynoDroid [27], A^{3}E [16], and others, focus on uncovering bugs within an app or examining the security or performance aspects of an app, rather than comparing how an app behaves across different versions or environments.

Our evaluation shows that Mimic is effective in finding UI compatibility problems in real Android apps and FicFinder data set. We have used Mimic to test 400 popular apps downloaded from Google Play on four different Android platform versions; we have also downloaded multiple versions of the same apps and tested them using Mimic. In various scenarios we have tested, Mimic has discovered that 15 apps have backward and forward compatibility problems across different Android versions (including well-known apps such as WatchESPN and Yelp). Mimic has also discovered that 5 apps throw different exceptions across different app versions. With FicFinder data set, Mimic has detected compatibility problems including 8 errors and 4 distorted UI issues, and performance problems such as four lagging UI problems, two memory bloat problems, and one battery drain problem. Section V discusses these and other findings in more detail.

II. Motivation

Prior research [16] shows how ineffective humans are at manually exploring app UIs. Humans can cover only 30.08%

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00040

of the app screens and 6.46% of the app methods according to the research. This further argues for automated testing of app UI.

There is little prior work on UI compatibility testing despite the need. For example, the FicFinder [35] work conducted an empirical study demonstrating 191 compatibility bugs and published a data set with 27 open-source apps that could be used to reproduce such bugs. Another paper [20] tests seven popular existing tools, and it states that none of the tools can handle the UI compatibility issues correctly. The paper mentions the need for a specific tool for UI compatibility testing. Unfortunately, there is no existing system that supports compatibility testing of UIs as discussed in Section VII. This section presents a set of scenarios under which specific UI testing functionality for mobile apps is required.

A. UI Compatibility Testing Scenarios

App developers and researchers frequently face the need for comparing app UI behavior across different app versions, Android versions, and/or devices. Here are four representative testing scenarios that require UI compatibility comparison. In the following sections, we define UI compatibility as consistent UI looks and behavior.

Forward-Backward Compatibility: According to Android Dashboard from Google [1], Android platform API versions from 10 to 27 are in active use as of July, 2018. Google has an aggressive release cycle of new Android APIs, and has been releasing a new version once or twice a year [2]. App developers need to ensure correctness across all previous Android versions (backward compatibility) as well as future releases (forward compatibility). This backward or forward compatibility testing requires UI compatibility testing across different Android versions.

Device Heterogeneity: There are many types of Android devices ranging from smartphones to tablets, with different screen sizes and resolutions. In order for apps to be UI compatible with a wide set of devices, developers need to test their apps on different types of devices.

Library Versioning: Most Android apps rely on third-party libraries for value-added functionality such as advertising, Google Play Services (e.g., Google Maps), visualization, image processing (e.g., OpenCV), and others. However, these libraries evolve in parallel with newer features being added to them over time. When these third-party libraries release a new version, developers need to test if their existing apps provide the same consistent user experience across different third party library versions. This requires UI compatibility testing across different versions.

App Instrumentation: Many researchers and companies are interested in using Java bytecode instrumentation techniques that can modify existing apps without having any source code. Examples include improving energy efficiency for always-on sensing [32], providing mobile deep links [17], or developing novel data management systems [19]. To test the correctness of such systems, researchers automatically transform existing Android apps with their instrumentation techniques (including the UI), and compare the behavior of the instrumented apps to the (uninstrumented) original apps. This requires UI compatibility testing.

B. Requirements for UI Compatibility Testing

Configurability: A basic requirement is the ability to easily run the same app on multiple devices using different app versions, Android versions, and third-party library versions for the purpose of comparing the UI across. Testers should be able to configure the set of devices they have, and set them up as required for their testing. They should also be able to install and configure the software platform on each of the devices to exactly set up the tests they would like to run.

Comparing UI Structures: Fundamental to UI compatibility testing is the ability to query UI structures and iterate through individual elements, e.g., buttons, menu elements, and text boxes. With this ability, a tester can ensure consistent UI experience by iterating through all elements, interacting with each of them, and comparing the experience with other runs of the same app in different environments.

Testing Modes: Some testers might want to thoroughly evaluate UI compatibility by iterating through all elements in a systematic manner, while other testers might want to perform a quick check of a subset of UI elements to ensure that their recent change has not affected the UI adversely. This advocates for both sequential and randomized UI testing.

Visual Inspection: A challenge when dealing with different form factors of mobile devices (e.g., screen size, resolution, and orientation) is the ability to accurately compare screens as viewed by the user. Thus, a testing framework needs to provide some form of automated comparison of the UIs displayed on different screens without requiring manual intervention, in order to enable scalable testing.

Interaction Performance: Even if two runs of an app (in different environments) look similar visually, there might be a difference in the performance such as greater lag between devices. This is an important problem to identify as this breaks consistent UI experience across app runs. A testing framework must be able to check for not just consistency of UI elements but to identify differences in performance parameters such as latency and memory use.

Motivated by these challenges, we have designed Mimic. The next section with describe the Mimic design in detail.

III. Mimic Design

Figure 1 shows the Mimic architecture. Mimic takes as input a set of apps and a Mimic script. The Mimic runtime that runs on a desktop that (described in subsection IV-A) executes the script, and runs specified tests on the set of devices connected. Mimic is designed to be able to scale the number of apps tested as well as the number of devices used. This architecture is managed by the Mimic programming model which provides methods for the setup of the runtime, configuring the set of devices that will be used for testing, specifying UI tests to be run, and the specification of desirable properties of the test runs that need to be logged. While it might be possible to

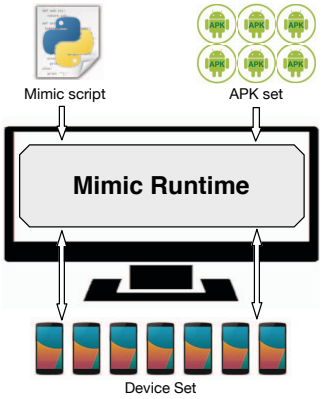


Fig. 1. Mimic Architecture

```
1 from mimic import Mimic
2 # Key: device type,
3 # Value: tuple(# serial, Android version, apk under test)
4 experiment_settings = {
5   "leader": (("0361676e094dd2ed", "4.4", "demo_v1.0.apk"),),
6   "follower": (("072dca96d0083871", "4.4", "demo_v1.1.apk"),)
7 }
8 Mimic(experiment_settings)
```

Fig. 2. Example Mimic Configuration

write similar scripts by a tester who understands the nuances of Android devices and tools, *Mimic makes this easier with a small set of APIs that provide high-level abstractions and callbacks specifically designed for UI compatibility testing*.

A. Mimic Programming Model

Our programming model provides easy abstractions to configure and set up testing environments with multiple devices, Android versions and app versions. It abstracts away this complexity by providing a single-device, single-version illusion, where testers write their testing logic as if they were using a single device and a single version of an app. Our implementation described in Section IV handles the differences of multiple devices and versions. Further, our programming model provides the ability to *compose* testing different aspects of Android apps, by providing an expressive, callback based abstractions for writing tests.

Abstractions for Test Configuration: We provide a simple dictionary abstraction for testers to express their requirements. Figure 2 shows a code snippet that initializes two devices to compare two different versions of an app. In line 5, the script configures a leader device by providing three parameters—a device serial number, an Android version, and an app file name. In line 6, the script configures a follower device, also with three parameters. In line 8, the script loads the configuration and initializes the Mimic runtime. When the Mimic runtime executes this script, it finds two devices using the serial numbers, installs Android 4.4 on both devices, and installs the respective versions of the app on the appropriate devices for testing.

Abstractions for Testing: Mimic provides an event-driven programming model and abstractions, so that testers only need to handle important events related to UI compatibility testing. These abstractions and the event-driven programming model simplify the process of writing a test script. Table I shows the set of five callbacks that the Mimic programming model

```
1 def onUITreeChanged(tree):
2   return tree.sort(sortType="random")
3
4 def handleNextUIObject(ui):
5   if ui.clickable:
6     ui.click.wait(3000) # the unit is ms
```

Fig. 3. Randomized Testing Example

provides and Table II lists the set of methods for configuration and UI testing. A tester can utilize these callbacks and interfaces for UI compatibility testing.

The most important abstraction is `UITree`; when an app launches or a user action occurs on an already-opened app, the app shows a new app window (or *activity* in Android's terminology). An activity is represented as a tree with three types of UI elements: (i) *the root*, which is the activity object itself, (ii) the middle UI elements that are containers of other UI elements, and (iii) the bottom-most leaves of UI elements. We capture this using our `UITree` abstraction. It contains a list of UI elements within an activity. We encapsulate each UI element with our `UIObject` abstraction, and by default, a `UITree` keeps all elements in an activity in its list in a random order. However, testers can modify the list membership and the ordering. Furthermore, `UITree` provides useful helper methods and abstractions. For example, a tester can sort the order of `UIObjects` using `sort()`, find a specific `UIObject` with `select()`, and check if all `UIObjects` are tested with `completedTest`. Table III shows the summary of `UITree`.

Along with `UITree`, we provide two callbacks that testers need to implement. These callbacks are designed to allow testers to handle only important events relevant to UI compatibility testing. Primarily, there are two categories of events that UI compatibility testing is interested in handling. The first category is new UI tree events—handling an event of this kind allows testers to perform specific actions when a new UI tree comes up (i.e., when a new activity comes up). For example, a tester might want to measure how long it has taken to make a transition from a previous activity to a new activity; or, another tester might want to analyze the UI elements within an activity and skip the testing for previously-tested UI elements. Our programming model defines a callback (`onUITreeChanged()`) to signal the appearance of a new UI tree. Testers can implement this callback and write their testing logic pertinent to an entire activity.

The second category of events is related to handling individual UI elements. Actual testing only happens when a tester interacts with a UI element and performs a certain action, e.g., a button click. Thus, we provide a callback (`handleNextUIObject()`) that gets invoked with the next UI element to process within `UITree`. This is possible because `UITree` keeps a list of `UIObjects`, and when an invocation of `handleNextUIObject()` returns, it gets called again with the next `UIObject` in the list.

Figure 3 and Figure 4 show two examples of how testers can use our programming model. These are common strategies that testers use. First, Figure 3 shows a randomized testing strategy, where a tester randomly sorts all UI elements in a new UI tree (lines 1-2) and performs a click on the first UI

TABLE I
CALLBACKS OF MIMIC PROGRAMMING MODEL

Callback Interface	Description
`UITree onUITreeChanged(UITree tree)`	Called when there is a new `UITree` event.
void `handleNextUIObject(UIObject ui)`	Called when there is a new `UIObject` to test. A `UIObject` specifies one UI element such as `Button` or `Text`.
void `onBatteryChanged(Battery battery)`	Called when the battery level of an app under test has changed. A `Battery` represents battery usage statistics for an app under test.
void `onHeapUsageChanged(Heap heap)`	Called when the heap usage of an app under test has updated. A `Heap` represents heap usage for an app under test.
void `onTestingTimeChanged(TestingTime time)`	Called when the total testing time has updated.

TABLE II
MIMIC PROGRAMMING INTERFACE: [1] denotes system-wide functionality, and [2] denotes device-specific functionality.

Programming Interface	Description
void `terminate(msg)` [1]	Terminate all testing with the given message.
void `detach(serial)` [1]	Remove the device of the given serial number from Mimic.
void `attach(serial)` [1]	Add the device of the given serial number to Mimic.
bool `diffUITree(threshold)` [1]	Return True if the leader screen and one of follower screens are different over the given threshold.
DeviceInfo `info()` [2]	Retrieve device information such as the screen width, height, rotation, product name, Android version, etc.
void `click()` [2]	Perform a button click.
void `longClick()` [2]	Perform a long click action on the object.
void `drag(x,y)` [2]	Drag the UI object to other point.
void `swipe(direction)` [2]	Perform a swipe action.
void `press(buttonName)` [2]	Press the given button. Supported home, back, recent, volumeUp, volumeDown, power, etc.
void `wait(time)` [2]	Wait the given time for the next action.
bool `contains(**kwargs)` [2]	Return Ture if this object contains a mapping for the given keyworded variables. It can be called with `Device`, `UITree`, and `UIObject`. Supported keywords: text, className, description, packageName, resourceId, etc.
void `screenshot(fileName)` [2]	Take a screenshot and save it with a given name.

TABLE III
THE UITREE ABSTRACTION

Attribute/Method	Description
device	`Device` instance on which this `UITree` is running.
previousUITree	The previous `UITree` of this `UITree`.
completedTest	Flag for weather all `Objects` in this `UITree` are tested or not.
loadingTime	Load time of this `UITree` (from performing a UI action to loading this `UITree`).
bool `contains(**kwargs)`	Return Ture if this `UITree` contains a mapping for the given keyworded variables. Supported keywords: text, className, description, packageName, resourceId, etc.
UIObject `select(**kwargs)`	Return the `UIObject` to which the given keyword variables are mapped in UITree, or None if the UITree contains no mapping for the given keyword variables.
UITree `sort(sortType)`	Returns a new `UITree` with those `UIObjects` in sorted order based on the given sortType.

```
1 def onUITreeChanged(tree):
2   if tree.previousUITree.completedTest != True:
3     tree.device.press('back')
4   return tree.sort(sortType="untested")
5
6 def handleNextUIObject(ui):
7   if ui.clickable:
8     ui.click.wait(3000) # the unit is ms
```

Fig. 4. Sequential Testing Example

element in the list for the UI tree (lines 4-6). It also shows an example use of `wait()`, which allows testers to wait a certain period of time for the next action. Second, Figure 4 shows a sequential testing strategy, where a tester tests every UI element on every activity. If a UI element action makes the app to transition to a new activity, then the testing script presses the back button to go back to the previous activity if the previous activity still contains more UI elements to test (lines 1-4). These examples show how concisely a tester can express testing logic.

In Section V, we show that simple randomized testing needs 11 lines of code using our programming model, while 178 lines of code is needed using Android UI Automator. The main difference comes from the fact that we provide high-level abstractions in the form of callbacks and take care of all the plumbing work necessary. Using Android UI Automator,

```
1  def onHeapUsageChanged(heap):
2    with open(heap.device+"_heapUsage.log", "a") as log:
3      log.write(str(Mimic.currentUItree) + " " + \
4      str(Mimic.currentUIObject) + " " + \
5      str(heap.total_heap_allocation) + "\n")
6
7  def onUITreeChanged(tree):
8    if tree.loadingTime > 200: # the unit is ms
9      Mimic.terminate("GUI lagging")
10
11   return tree.sort('untested')
```

Fig. 5. Performance Testing Example

testers have the burden of checking if there is a new UI tree, getting all the UI elements from the tree, checking their properties (e.g., whether or not they are clickable), etc. This makes a significant difference in terms of development effort.

Abstractions for Performance Monitoring: Previous research [26] has shown that are three predominant types of performance bugs in Android apps. They are GUI latency, energy leaks, and memory bloat. Mimic programming model provides mechanisms for monitoring the resources related to these bugs and provides callbacks so the test writer can easily specify testing actions to compare how their apps behave across different devices and versions. Specifically, our programming model provides three callbacks: `onHeapUsageChanged()`, `onBatteryChanged()`,

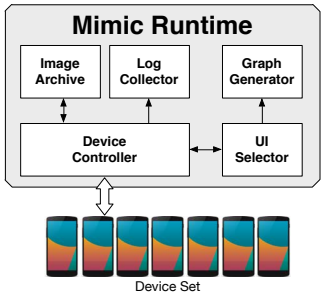


Fig. 6. Mimic Runtime Components

and `onTestingTimeChanged()`, which the programmer can leverage to invoke testing.

`onHeapUsageChanged()` is invoked when the heap usage of an app has changed. This is configurable, and by default, it is triggered every 10kB. `onBatteryChanged()` is called when the battery level changes. `onTestingTimeChanged()` is invoked when the total testing time changes. This is also configurable, and by default, it is triggered every 1 second. These methods allow the tester the ability to log these changes along with other state such as the UI tree to compare later. Figure 5 shows an example, where the code logs how much the heap usage has changed, what is the state of the UI, and what is the current UI object being interacted with.

Helper Methods for UI Difference Monitoring: An essential aspect of UI compatibility testing is comparing how an app displays its UI across different versions and devices. In order to support this, our programming model provides two methods for capturing and comparing screenshots. `screenshot()` method takes a screenshot. `diffUITree()` method returns true if any of the followers is displaying a different UI from that of the leader. `diffUITree()` takes a threshold as an input parameter, which indicates the percentage of difference. For example, `diffUITree(5)` returns true when any of the follower's UI is more than 5% different from the leader's. We use image processing to implement this as we describe in Section IV-B.

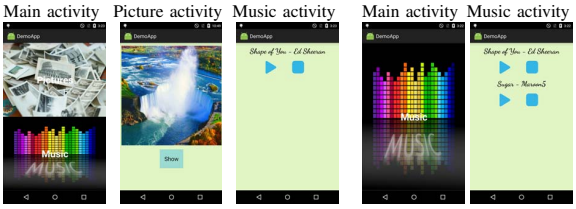
IV. Mimic Implementation

This section describes the Mimic runtime as well as how we enable visual inspection. Our prototype is roughly 2,500 lines of Python code.

A. Mimic Runtime

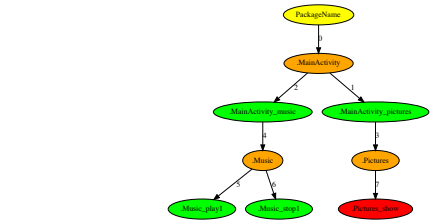
The Mimic runtime provides the environment for the execution of a Mimic script. It runs on a computer, controls all Android devices connected to it via USB, and displays test results. Figure 6 shows the components of the runtime. The main component is *Device Controller* that executes a Mimic script and controls the entire flow of testing. While executing a script, it interacts with testing devices using Android tools (mainly, `adb` and UI Automator [3]), and leverages other components. This section describes the flow of execution and the components of the runtime.

Initialization: Device Controller first installs an Android OS images as well as testing apps on different devices and

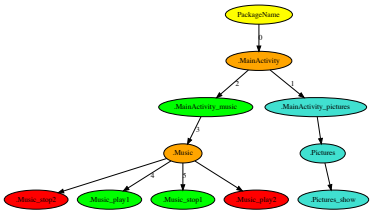


(a) Demo app 1.0* (b) Demo app 1.1

Fig. 7. Two Different Versions of Demo Apps. * denotes the leader device.



(a) Graph for Demo app 1.0



(b) Graph for Demo app 1.1

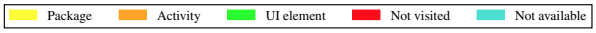


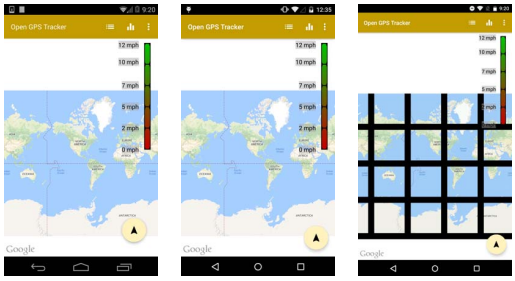
Fig. 8. Graph Representations for Demo Apps in Figure 7. The numbers denote the visiting nodes order.

configures each device. *Android Image Archive* stores stock Android OS images; our implementation currently supports ten images ranging from Android 2.3.6 to 6.0.1.

Testing Logic Execution: After initialization, Device Controller launches testing apps on all devices. It then periodically monitors each device for UI tree and resource status changes (e.g., battery level). If any change is detected, Device Controller invokes the appropriate callback provided by our programming model. Device Controller monitors UI tree changes using Android UI Automator, which enables the inspection of UI elements and their hierarchy. Device Controller uses `adb` to monitor resource status changes. Our periodicity of monitoring is currently 0.1 seconds, and this is configurable.

When Device Controller detects a new UI tree (i.e., when a new activity comes up on a device), it interacts with *UI Selector* to filter out unnecessary UI elements. We filter UI elements because there are many that are just containers for other UI elements and are not typically necessary for testing. For example, a simple button in Android that turns Wi-Fi on or off requires multiple UI elements, e.g., a text box, an interactable button, and a container that groups all UI elements together. A UI testing system would only be interested in the button that is clickable. The UI selector removes non-leaf UI elements as well as non-clickable ones from the UI hierarchy.

Testing Result Generation: After executing testing logic,



Anroid 4.4* Anroid 5.0 Anroid 6.0.1

Fig. 9. Open GPS Tracker on Different Android Versions. * denotes the leader device.

TABLE IV
UI DIFFERENCE INSPECTION USING THREE METHODS. * denotes methods that Mimic uses.

Sample Testing Image Type	Color Histogram*	Template Matching	Feature Matching*
Different UI Color	47.86%	6.5%	21.69%
Different UI Position	2.6%	90.31%	31.56%
Absence of UI	16.57%	9.97%	31.1%

Device Controller provides testing results in two forms. First, it uses *Log Collector* to capture all logs that testing apps generate. Device Controller can display individual logs as well as deltas among the logs from different devices. Second, Device Controller uses *Graph Generator* to display all UI transitions that occurred during testing. This UI transition graph helps testers visualize how different app instances behaved. Figure 7 and Figure 8 show visualizations of an example using a demo app we have developed. There are two versions of this app. The first version (v1.0) has three activities, Main, Picture, and Music. However, the second version (v1.1) only has two activities, Main and Music, and there are two more buttons in the Music activity. We intentionally inject one run-time error that causes the app to crash when the Show button on the Picture activity is clicked.

Figure 8 shows an example run. Using our leader-follower model, both versions execute the exact same sequence of UI events. For v1.0, the unvisited node (the Show button) is due to the run-time crash error that we inject; since the app crashes, the button is not explored. For v1.1, the two unvisited nodes (the second Play button and the corresponding Stop button) are due to our leader-follower model; since the leader does not have those buttons, they are not explored in the follower. Using these graphs, testers can easily discover points of run-time errors and compare UI differences across different versions.

B. Enabling Visual Inspection

As mentioned earlier, our programming model provides `screenshot()` and `diffUITree()` to take and compare screenshots. In our implementation, `diffUITree()` computes a measure of change between the leader and the followers. We do this by (i) taking a screenshot across testing devices, (ii) masking the top status bar and the bottom navigation bar because each Android version has the different size of status and navigation bars, and (iii) calculating the difference between the screenshots using both color histogram difference [7] and feature matching [6].

TABLE V
STATISTICS OF 400 APPS

Category	# of Apps	Example	Average App Size
Lifestyle	42	XFINITY Home	15.0M
Entertainment	49	Vimeo	14.2M
Travel	56	TripAdvisor	13.1M
Sports	53	ESPN	20.0M
Personization	52	Paypal Cash	11.9M
Education	43	Bookshelf	14.0M
Tools	55	Go Security	9.9M
Photography	50	Open Camera	17.4M
Total	400	N/A	14.4M

To calculate the UI difference, we have experimented with three popular methods from OpenCV [9]—histogram difference [7], template matching [12], and feature matching [6]. Using original screen image of Starbucks app, we create three different sample image types that are possible due to version differences, and we compare each sample image type with the original image using three methods mentioned. Table IV shows percentage differences from this comparison. While results of template matching show an inaccuracy on all three sample image types, color histogram difference and feature matching have given us satisfactory results either on color difference or on UI feature difference. Mimic takes higher percentage difference between color histogram difference and feature matching.

While we have used particular image processing mechanisms, it is also easy to create other custom mechanisms to quantify visual difference between two screens. Figure 9 shows an example of display differences across different Android versions from 4.4 to 6.0.1. In this example, the leader screenshot is about 11.84% different from the image on Android 5.0, and about 65.01% different from the image on Android 6.0.1. By using both `diffUITree()` and `screenshot()`, a tester can efficiently detect the display distortion.

V. EVALUATION

In this section, we demonstrate Mimic's capabilities in three ways. To demonstrate effectiveness of our programming model, we compare the lines of code necessary to implement different testing strategies using Mimic and Android UI Automator. Second, we evaluate Mimic in different scenarios—forward compatibility testing, backward compatibility testing, different app version testing, and instrumentation testing. For each testing scenario, we have used separate 100 Android apps due to different aspects that we want to evaluate. For each testing scenario, we explain the details of how we have picked the apps. Table V shows the statistics of the total 400 apps. Third, we conduct an experiment to show that Mimic correctly catches compatibility differences. For this, we use 14 apps from the FicFinder [35] data set, which are real apps downloaded from open-source repositories such as GitHub with verified UI compatibility problems.

We have used 10 Android devices for our experiments—three Nexus S (512MB RAM) running Android versions (2.3.6, 4.0.4, and 4.1.2), two Galaxy Nexus (1GB RAM) running (4.1.1 and 4.2.1), four Nexus 5 (2GB RAM) running (4.4, 5.0, 5.1, and 6.0), and one Galaxy S4 (2GB RAM) running Android 4.4.

TABLE VI
A SUMMARY OF DEVELOPMENT EFFORT. [1] denotes it runs on a single device independently, [2] means that it runs across all testing devices.

Testing Logic	Framework	LoC	Testing Environment
Randomized testing[1]	UI Automator	178	Non-automated
Sequential testing[1]	UI Automator	199	Non-automated
Randomized testing[2]	Mimic	11	Automated
Sequential testing[2]	Mimic	13	Automated

A. Development Effort

We demonstrate the effectiveness of our programming model by comparing the lines of code necessary to implement randomized testing and sequential testing. The randomized testing strategy we implement picks one clickable UI element (e.g., a button or a text input box), performs an action on it (e.g., a button click), and repeats it until either there is no clickable UI element or the app crashes. The sequential testing strategy we implement tests each and every UI element one by one. We have implemented both strategies using Android UI Automator and Mimic.

Mimic programming model enables compact descriptions of testing strategies—our randomized testing implementation requires 11 lines of code and our sequential testing implementation requires 13 lines of code. In contrast, UI Automator requires 178 lines of code for randomized testing and 199 lines of code for sequential testing, which are 16x and 15x higher, respectively. Even worse, it does not have any support for device management, environment initialization, etc.; it requires testers to manually set up a testing environment. Table VI shows this result.

B. Forward Compatibility Testing

To demonstrate that Mimic is useful to test forward compatibility, we have downloaded popular 1000 real apps from Google Play, and chosen 100 apps that target Android API 19. We then run the selected 100 apps across four Android API versions from 19 (Android 4.4) to 23 (Android 6.0) three times. We have designated API 19 as the leader, and API 21, API 22, and API 23 as followers.

We use three methods in our experiment—*base-random*, *Mimic-random*, and *Mimic-sequence*. The *base-random* method *does not* use Mimic; it implements the randomized testing strategy described earlier in subsection V-A using UI Automator. Since it does not use Mimic, randomization is done on each device *independently* without using our leader-follower model. The *Mimic-random* method uses Mimic and implements the same randomized testing strategy. However, since it uses Mimic, all UI actions are synchronized across different devices. The *Mimic-sequence* method uses Mimic and implements the sequential testing strategy described earlier in subsection V-A. During the experiment, we have captured all logs that each app generates through our Log Collector. We have collected 1,200 log files and analyzed the log files.

Table VII shows the summary of errors detected by our experiment. All the errors reported in this section have caused the testing apps to crash. 7, 16, and 21 errors from nine apps are uncovered by the base-random method, the Mimic-random method, and the Mimic-sequence method, respectively. 9 apps(out of 100 apps) have thrown at least one error.

More specifically, Yelp threw `NetworkOnMainThreadException` across four Android API versions—`NetworkOnMainThreadException` is thrown if an app performs a long task such as a networking operation on its UI thread (also called the main thread). The cause of the crashes from IP Webcam and CCleaner is not properly handling new features such as `Runtime Permissions`. Three apps, Home Exercise Workouts, Weather Kitty, and Livestream, threw `NoSuchMethodError` at run time since they invoke methods that newer versions of Android no longer support. For example, `Livestream` app has crashed on API 23 because this app uses Apache `HTTPClient`, which is deprecated effective API 23 [4].

We make two observations. First, comparing the base-random and the Mimic-random methods, we observe that Mimic's follow-the-leader model of testing is often more effective than independent testing. This is because 5 out of 9 apps report more errors when using Mimic and 2 out of 9 apps report the same number of errors. Second, comparing the Mimic-random and the Mimic-sequence methods, we observe that the Mimic-sequence method is more effective in all cases. This is simply because the sequential testing tests every UI element one at a time, and hence has a better chance of triggering an error.

C. Backward Compatibility Testing

To show that Mimic is useful for backward compatibility as well, we have run the same experiment as mentioned in subsection V-B. Only differences are that we have initialized the leader to use API 23 (Android 6.0), and followers to use API 19 (Android 4.4), API 21 (Android 5.0), and API 22 (Android 5.1), and selected 100 apps that target API 23.

Table VIII shows the summary of errors detected by our experiment. 5, 6, and 13 errors from 6 apps (out of 100 apps) are found by the base-random method, the Mimic-random method, and the Mimic-sequence method, respectively. The apps shown in Table VIII have experienced at least one error. All of the errors caused the apps to crash, since they are `NullPointerException` and `NoSuchMethodError`. Once again, `NoSuchMethodError` has been caused by the use of deprecated APIs. For example, Map of NYC Subway app uses `getDrawable()` which is not available on the Android platform below API 22. Overall, we can also see the similar patterns that we have observed for the forward compatibility experiment.

D. App Version Compatibility Testing

In order to evaluate Mimic's capability for app version compatibility testing, we have selected 2,097 apps that are top 100 apps in each category from Google Play. We have monitored these apps for two weeks, and chosen 100 out of 233 apps that have at least two different versions. For the simplicity of presentation, we simply call earlier versions of

TABLE VII
THE DETAILS OF DETECTED ERRORS FOR FORWARD COMPATIBILITY. * denotes Android API version on the leader.

App Name	Android API Version												Exception Type
	Base-random				Mimic-random				Mimic-sequence				
	19*	21	22	23	19*	21	22	23	19*	21	22	23	
Yelp				✔	✔	✔	✔	✔	✔	✔	✔	✔	NetworkOnMainThreadException
College Sports Live		✔		✔		✔	✔	✔		✔	✔	✔	NullPointerException
Home Exercise Workouts				✔		✔	✔	✔		✔	✔	✔	NoSuchMethodError
Fast Cleaner		✔								✔	✔	✔	NullPointerException
Photo Lab						✔	✔	✔		✔	✔	✔	NullPointerException
Weather Kitty							✔	✔			✔	✔	NoSuchMethodError
IP Webcam				✔								✔	Mishandling runtime permissions
CCleaner												✔	Mishandling runtime permissions
Livestream				✔				✔				✔	NoSuchMethodError

TABLE VIII
THE DETAILS OF DETECTED ERRORS FOR BACKWARD COMPATIBILITY. * denotes Android API version on the leader.

App Name	Android API Version												Exception Type
	Base-random				Mimic-random				Mimic-sequence				
	23*	22	21	19	23*	22	21	19	23*	22	21	19	
truTV		✔								✔	✔	✔	NullPointerException
Whitetail Deer Calls				✔		✔	✔	✔		✔	✔	✔	NoSuchMethodError
Cheapflights										✔	✔	✔	NoSuchMethodError
Map of NYC Subway				✔			✔	✔			✔	✔	NoSuchMethodError
WatchESPN			✔				✔				✔		NullPointerException
Collage Maker				✔								✔	NoSuchMethodError

TABLE IX
THE DETAILS OF DETECTED ERRORS. Both the stable and new apps are tested on Android 4.4.

App Name	Stable Version			New Version			Exception Type
	Base-random	Mimic-random	Mimic-sequence	Base-random	Mimic-random	Mimic-sequence	
TD Jakes Sermons	✔	✔	✔	✔	✔	✔	InflateException
Venmo		✔	✔		✔	✔	IllegalArgumentException
Volume Booster		✔	✔		✔	✔	SecurityException
Countdown Timer & Stopwatch & Caller ID		✔	✔		✔	✔	NullPointerException
Wonder Buy			✔				NullPointerException

TABLE X
THE DETAILS OF DETECTED ERRORS FROM 10 INSTRUMENTED APPS. Both versions of apps are tested on Android 4.4.

App Name	Non-instrumented Version		Instrumented Version		Exception Type
	Mimic-random	Mimic-sequence	Mimic-random	Mimic-sequence	
GasBuddy	✔	✔	✔	✔	ClassCastException
Stamp and Draw Paint 2	✔	✔	✔	✔	ClassCastException
Big 5 Personality Test			✔	✔	AssertionError
Text To Speech Reader			✔	✔	AssertionError
Gun Vault				✔	AssertionError
Street Art 3D			✔	✔	NullPointerException
Viaplay GameCenter-F			✔	✔	NullPointerException
Q Recharge			✔	✔	NullPointerException
Funny Face Photo Camera				✔	NullPointerException
Japan Offline Map Hotels Cars				✔	NullPointerException

apps as stable versions, and later versions of apps as new versions in this experiment. We have run the 100 stable versions and new versions of apps on the same Android version, 4.4, in order to show that Mimic is effective in discovering errors that occur across different versions of apps.

Table IX shows the result. In total, four types of errors have been found in five apps. For TD Jakes Sermons app, the error happens during the initialization, thus it is reported in all testing logic. For Wonder Buy, the error has been found in the stable version, but not in the newer version, which indicates that the developer has fixed the bug for the newer version. We observe that the two methods using Mimic (the Mimic-random and the Mimic-sequence methods) discover more errors across different versions than the base-random method.

E. Instrumentation Testing

To demonstrate Mimic's ability to compare runtime behavior between non-instrumented and instrumented versions, we have used Reptor [24] and its data set. Reptor is a system we have developed previously. It is a bytecode instrumentation tool enabling API virtualization on Android. To demonstrate the capability of Reptor, it uses 1,200 real apps, instruments the apps, and verifies the runtime behavior between the original and the instrumented version of apps using a randomized testing strategy. This randomized strategy they use is in fact the same as the base-random method described in subsection V-B.

For our experiment, we have used the same 1,200 apps and randomly selected 100 apps of the 1,200, and run the apps on Android 4.4 two times. We have also used the result from the Reptor paper for those 100 apps, and use it as the base-random comparison point in this experiment. In the first run, we have run the apps with the Mimic-random method. In the second run, we have used the Mimic-sequence method. We have also recorded heap allocation sizes using `onHeapUsageChanged()` described in Section III in order to show heap usage comparison between the non-instrumented and instrumented apps.

TABLE XI
THE DETAILS OF DETECTED COMPATIBILITY BUGS. The leader (Nexus 5) running on Android 6.0.1. * denotes that the bug are confirmed on multiple devices. E and U stand for errors and UI distortion respectively.

App Name	Category	LoC	APK Version No.	Bug Type	Results	Device (Android Version)
Open GPS Tracker	Travel & Local	12.1K	1.5.0	U	✔	Nexus 5/6.0.1
ConnectBot	Communication	17.2K	1.8.6	E	✔	Galaxy Nexus (4.1.1)
AnkiDroid	Education	45.2K	2.5alpha48	E	✔	Galaxy S4 (4.4)
c:geo	Entertainment	64.8K	2015.11.29	E	✔	Nexus S (2.3.6)
AnySoftKeyboard	Tools	23.0K	1.6.134	E	✔	Galaxy S4 (4.4)
QKSMS	Communication	56.4K	2.4.1	E	✔	Neuxs 5 (5.1)
BankDroid	Finance	22.8K	1.9.10.1	E	✔	Galaxy S4 (4.4)
Evercam	Tools	14.8K	1.5.9	E	✔	Nexus S (4.1.2)*
ChatSecure	Communication	37.2K	14.0.3	U, E	✔	Galaxy Nexus (4.2.1)
AntennaPod	Media & Video	38.9K	1.4.1.4	U	✔	Nexus 5 (5.1)
Brave Android Browser	Personalization	20.2K	1.7.4	U	✔	Nexus 5 (4.4)
IrssiNotifier	Communication	3.5K	1.7.14	E	✘	
CSipSimple	Communication	59.2K	1.02.03	U	✘	
Bitcoin Wallet	Finance	17.8K	4.45	E	✘	

TABLE XII
THE DETAILS OF DETECTED PERFORMANCE BUGS. The leader (Nexus 5) running on Android 6.0.1. * denotes that the bug are confirmed on multiple devices. G, B and M stand for GUI lagging, battery leak, and memory bloat respectively. U stands for unknown due to not enough information given.

App Name	Category	LoC	APK Version No.	Bug Type	Results	Device (Android Version)
Open GPS Tracker	Travel & Local	12.1K	1.5.0	G	✔	Galaxy Nexus (4.1.1)
AnkiDroid	Education	44.6K	2.1beta6	M	✔	Nexus S (4.0.4)*
AntennaPod	Media & Video	38.7K	1.4.0.12	G	✔	Nexus 5 (4.4)*
CSipSimple	Communication	59.2K	1.02.03	B	✔	Galaxy S4 (4.4)
Brave Android Browser	Personalization	20.2K	1.7.4	G, M	✔	Nexus 5 (4.4)*
K-9 Mail	Communication	60.8K	5.106	G	✔	Nexus 5 (5.1)
WordPress	Social	73.1K	4.8.1	U	✘	

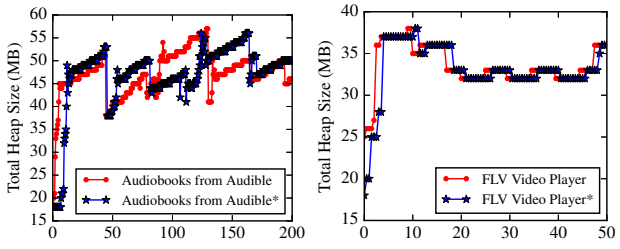


Fig. 10. Heap Usage for Four Selected Apps (*Instrumented Apps). The x-axis shows testing time in seconds.

Table X shows the details of the result. Mimic has caught 10 errors that were not originally reported in the Reptor paper using their randomized testing strategy (i.e., the base-random method described in subsection V-B), which can miss some of the UI paths. Two errors are found in both non-instrumented and instrumented versions. Eight errors are detected in only instrumented apps.

Figure 10 shows the heap usage results over testing time. The results help app developers to understand memory utilization of their apps. From the results, we can see that Mimic effectively verifies the runtime behavior in terms of error detection and performance comparison.

F. FicFinder Data Set Results

Compatibility Problems: The FicFinder data set [35] consists of 27 large-scale open-source Android apps with various bugs and problems. Out of the 27 apps, 14 apps are known to have UI compatibility problems, and we have conducted an experiment to evaluate whether or not Mimic can flag those apps as problematic. We have used the randomized testing strategy and `diffUITree()` function with 10% threshold. We have tested each of 14 apps for 5 minutes.

Table XI shows the results. We have detected 8 errors and 4 distorted UI issues from 11 apps. For example, c:geo uses `ShowcaseViewBuilder` that does not work on API 14 or below, and it crashes on Nexus S running API 10. Evercam also crashes on Nexus S (API 16), Nexus 5 (API 19), etc., due to a third-party library uses called `SplunkMint`. The library has a compatibility problem with Android versions from API 16 to API 19. Four UI distortion issues have structural UI problems that do not render UIs correctly across different Android versions. Mimic did not flag three apps (IrssiNotifier, CSipSimple, and Bitcoin Wallet), since their problems are specific to particular devices (namely, Blackberry and Zenfone 5). Since we have not tested on those devices, Mimic has not reported any problem.

Performance Problems: We have also used the FicFinder data set to test Mimic's capability to detect performance problems (i.e., lagging UIs, energy leaks, and memory bloats). Out of 27 apps, 7 apps are known to have performance problems, and we have used those apps in our experiment to see if Mimic flags those apps as problematic. We have used the randomized testing strategy, and also implemented `onBatteryChanged()` and `onHeapUsageChanged()` to record battery and heap usage. We report cases where the difference in battery or heap usage is more than 5%. To detect battery bloat, we have run this experiment up to 30 minutes.

Table XII shows the results. The results show that Mimic flags 7 apps out of 8 apps as problematic. Mimic detects four lagging UI problems, two memory bloat problems, and one excessive battery drain problem. We consider GUI lagging if an app has a UI that cannot be loaded within 5 seconds. From our inspection of AntennaPod code and logs, AntennaPod lags whenever it downloads new episodes. AnkiDroid and Brave Android Browser throw `java.lang.OutOf-`

TABLE XIII
COMPARISON TO EXISTING TOOLS. Program., Hetero., and No Instr. stand for programmability, heterogeneity, and no instrumentation required.

Name	Type	Program.	Hetero.	No Instr.
Mimic	Black	✔	✔	✔
MonkeyRunner [8]	Black			✔
Caiipa [25]	Black			✔
DroidFuzzer [36]	Black			✔
PUMA [22]	Black			
RoboTest [10]	Black	✔		✔
Robotium [11]	Black	✔		✔
Espresso [5]	Grey	✔		✔
AndroidRipper [13]	Black			
A^3E [16]	Black			✔
Dynodroid [27]	Black			✔
EvoDroid [28]	White			✔
ConcolicTest [15]	Black			
VarnaSena [31]	Black			

`MemoryError`. CSipSimple causes about 16% battery drain for 30 minutes on Galaxy S4. We could not find the root cause of the excessive battery drain, but we realized that CPU user time of CSipSimple is 965s 13ms from the logs that Mimic captured. Mimic does not flag WordPress as problematic—the WordPress issue tracker reports that WordPress (4.8.1 version) has GUI lagging issue. However, there is not enough information to determine for certain that there is indeed a performance problem.

VI. DISCUSSION

In this section, we discuss implications of our design choices as well as potential improvements to Mimic.

Mimic may not be suitable for all types of apps: Although our evaluation shows that Mimic is effective in finding potential UI compatibility problems across different versions and devices, it is not the case that Mimic is suitable for all types of apps. For example, sensor-centric apps such as games are not suitable for testing on Mimic since Mimic does not test sensor input. For future work, we are exploring how to enable automated testing for sensor-centric apps.

Mimic does not support system events: Some UI actions can be triggered by system events, for example, a calendar alarm pop-up can be triggered by a system alarm event. Currently, Mimic does not support the testing of such UI events. However, supporting such UI events is potentially possible by extending our current implementation of Mimic; with the Android Activity Manager (available via `am` command), we can send system events to an app. However, this could lead to a vast space to explore for testing as Android has a large set of system events. Intelligently reducing the size of this space is also our ongoing work.

VII. RELATED WORK

In this section, we compare existing mobile app testing systems with Mimic on a few important aspects and discuss how Mimic occupies a unique position in the design space of mobile app testing. In Table XIII, we compare Mimic to other systems in four aspects—testing type, programmability, support for heterogeneity, and whether or not app instrumentation is required. We discuss each aspect in this section.

Testing Type: There are mainly three types of testing approaches, depending on the availability of source code. The first type is black-box testing which does not require any source code. Fuzz testing [8], [25], [36], [22], [11], [37], [10] and model-based testing [13], [16], [27], [18], [14], [30], [33] belong to this category, but they focus on testing a single app. Mimic also belongs to this category, but its focus is on using multiple devices and versions and testing UI compatibility. The other two approaches are white-box testing and grey-box testing. White-box testing requires full access to source code. Grey-box testing does not require source code but assumes some knowledge about apps being tested, e.g., IDs of UI elements. Typical symbolic execution [29], [34], [21] uses white-box testing, and there are other systems that take either a white-box approach [28] or a grey-box approach [5]. Since white-box and grey-box approaches require knowledge about apps being tested, they are not suitable for certain scenarios such as testing instrumented apps.

Programmability: Many existing systems [10], [11], [5] only allow testers to use predefined methods of input, e.g., a monkey-based random input generation method. These systems do not provide any programmable API.

Support for Heterogeneity: Other systems [8], [25], [36], [22], [10], [11], [13], [16], [27], [28], [15], [5], [31], [23] do not provide support for handling multiple devices or versions. In these systems, testers need to manually set up and configure their testing environments, if they want to leverage those systems for UI compatibility testing.

App Instrumentation: A few existing systems [31], [22], [13], [15] require app instrumentation to enable testing. While instrumentation allows deep inspection of an app without requiring its source code, it modifies the app code which might result in producing different behavior (e.g., heisenbugs).

VIII. CONCLUSIONS

In this paper, we have described a new UI compatibility testing system for Android apps called Mimic. Mimic supports *follow-the-leader* model of parallel testing, where we designate one device to perform a sequence of UI actions; all other devices follow this leader and perform the same sequence of UI actions. This model is useful for UI compatibility testing across different Android versions, device types, and app versions. Testing is made easy by Mimic through a concise programming model for writing tests. The programming model allows testers to quickly set up testing environments, and to express their own testing logic. After executing testing logic, Mimic reports UI compatibility problems such as different exceptions thrown, different UI paths taken, differences in resource usage, etc. Our evaluation with a few hundred Android apps downloaded from Google Play shows that Mimic can effectively detect real errors and report UI compatibility problems across different Android or app versions.

IX. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported in part by the generous funding from the National Science Foundation, CNS-1350883 (CAREER) and CNS-1618531.

REFERENCES

[1] Android Dashboard. https://developer.android.com/about/dashboards/index.html, October 2017.

[2] Android Release Notes. https://developer.android.com/topic/libraries/architecture/release-notes.html, October 2017.

[3] Android UI Automator. https://developer.android.com/topic/libraries/testing-support-library/index.html, Jan 2017.

[4] Apache HTTP Client Removal. https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html, September 2017.

[5] Android Espresso. https://developer.android.com/training/testing/espresso/index.html, Apr 2018.

[6] Feature Matching. http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_matcher/py_matcher.html, August 2018.

[7] Histogram Comparison. http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_histograms/py_histogram_begins/py_histogram_begins.html, August 2018.

[8] MonkeyRunner. https://developer.android.com/studio/test/monkeyrunner/index.html, April 2018.

[9] OpenCV. https://opencv.org/, August 2018.

[10] RoboTest. https://firebase.google.com/docs/test-lab/robo-ux-test, April 2018.

[11] Robotium. https://github.com/RobotiumTech/robotium, April 2018.

[12] Template Matching. http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_template_matching/py_template_matching.html, March 2018.

[13] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012.

[14] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. Mobiguitar: Automated model-based testing of mobile apps. *IEEE Software*, 2015.

[15] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, 2012.

[16] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. *SIGPLAN Not.*, 48(10), Oct. 2013.

[17] T. Azim, O. Riva, and S. Nath. ulink: Enabling user-defined deep linking to app content. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, 2016.

[18] Y.-M. Baek and D.-H. Bae. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, 2016.

[19] S. Chandrashekhara, T. Ki, K. Jeon, K. Dantu, and S. Y. Ko. Bluemountain: An architecture for customized data management on mobile systems. In *Proceedings of the 23th Annual International Conference on Mobile Computing and Networking*, MobiCom '17, 2017.

[20] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? (e). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 429–440, Washington, DC, USA, 2015. IEEE Computer Society.

[21] S. Ganov, C. Killmar, S. Khurshid, and D. E. Perry. Event listener analysis and symbolic execution for testing gui applications. In *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '09, 2009.

[22] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. 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 '14, 2014.

[23] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, 2013.

[24] T. Ki, A. Simeonov, B. P. Jain, C. M. Park, K. Sharma, K. Dantu, S. Y. Ko, and L. Ziarek. Reptor: Enabling api virtualization on android for platform openness. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, 2017.

[25] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, MobiCom '14, 2014.

[26] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, 2014.

[27] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, 2013.

[28] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, 2014.

[29] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes*, 37(6), Nov. 2012.

[30] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon. Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, 2014.

[31] 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, 2014.

[32] H. Shen, A. Balasubramanian, A. LaMarca, and D. Wetherall. Enhancing Mobile Apps to Use Sensor Hubs Without Programmer Effort. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, 2015.

[33] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, 2017.

[34] Y. Su, Y. Yu, Y. Qiu, and A. Fu. Symfinder: Privacy leakage detection using symbolic execution on android devices. In *Proceedings of the 4th International Conference on Information and Network Security*, ICINS '16, 2016.

[35] L. Wei, Y. Liu, and S. C. Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

[36] H. Ye, S. Cheng, L. Zhang, and F. Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, MoMM '13, 2013.

[37] L. L. Zhang, C.-J. M. Liang, W. Zhang, and E. Chen. Towards a contextual and scalable automated-testing service for mobile apps. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications*, HotMobile '17, 2017.