Automation of Android Applications Functional Testing Using Machine Learning Activities Classification

Ariel Rosenfeld*
Dept. of Computer Science and Applied Mathematics, Weizmann Institute of Science
Rehovot, Israel

Odaya Kardashov
Dept. of Computer Science, Bar-Ilan University
Ramat-Gan, Israel

Orel Zang
Dept. of Computer Science, Bar-Ilan University
Ramat-Gan, Israel

ABSTRACT

Following the ever-growing demand for mobile applications, researchers are constantly developing new test automation solutions for mobile developers. However, researchers have yet to produce an automated functional testing approach, resulting in many developers relying on a resource consuming manual testing. In this paper, we present a novel approach for the automation of functional testing in mobile software by leveraging machine learning techniques and reusing generic test scenarios. Our approach aims at relieving some of the manual functional testing burden by automatically classifying each of the application's screens to a set of common screen behaviors for which generic test scripts can be instantiated and reused. We empirically demonstrate the potential benefits of our approach in two experiments: First, using 26 randomly selected Android applications, we show that our approach can successfully instantiate and reuse generic functional tests and discover functional bugs. Second, in a human study with two experienced human mobile testers, we show that our approach can automatically cover a large portion of the human testers' work suggesting a significant potential relief in the manual testing efforts.

KEYWORDS

Android Applications Testing, Mobile Testing Automation, Activities Classification

1 INTRODUCTION

Mobile devices are becoming a key component in our lives, with more than half of the world's population now owning one [4]. More than five million applications have been developed so far [5], making them the main productivity feature of these devices. As mobile devices become more popular, thus arises the need for efficient techniques for testing their applications. In order to support todays applications' short development cycles driven by agile development methodologies, software testing researchers have dedicated their efforts to produce automated testing tools for mobile applications, mostly for the Android Environment [15].

A recent survey by Linares-Vásquez et al. [21], which was conducted among 102 experienced Android developers, indicates that the mobile testing automation vision has yet to come to fruition. Despite the variety of testing automation tools produced by the academia, researchers have yet to convince the developers about their benefits, resulting in most developers relying on manual testing. The reasons reported in the survey include the lack of reproducible test cases and lack of debugging support. Indeed, there has been a minor adoption of automation APIs which enable a developer to write code for application functional testing and executing this code in test specific scenarios. The problem with this approach is that these tests are hand-coded for specific applications and specific scenarios, and each new application requires spending many resources to reuse these tests. In addition, these tests require high maintenance since every change in the application should be reflected in the pre-coded tests.

In this paper, we present a novel approach for automatic testing of Android applications in order to find as many functional bugs as possible. Our approach is based on the premise that different activities in an Android application share a similar interface structure. In order to use this similarity for our benefit, we use machine learning techniques to classify each activity in the application into one of seven pre-defined activity types which are identified in this work. For each classified activity, we then run specialized tests, at user interface level, that were coded to utilize the fact that we know the activity's structure and desired behavior. We have implemented this approach and developed a tool in Java for the TestProject[1] test automation framework that uses the Appium[2] open-source framework as a bridge between the mobile device and our code. TestProject allows a developer to build, deploy and execute automated testing by utilizing popular open-source frameworks for both Web and Mobile applications. The platform includes hundreds of add-ons for automated testing which are freely available. Our developed tool, named ACAT, standing for "Activities Classification for Application Testing", will be available to install as an add-on via TestProject add-ons store.

To evaluate our approach, we conducted an experiment in which we executed our add-on on different applications. We found that the ACAT add-on shows great ability in exploring the application and testing its key components without prior knowledge about the application. This lets the developer focus on the development of the application and not on writing standard tests or relying on a

*Corresponding author, arielros1@gmail.com

MOBILESoft '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5712-8/18/05…$15.00
https://doi.org/10.1145/3197231.3197241

[1] https://blog.testproject.io/
[2] http://appium.io/

resource consuming manual testing. The use of machine learning for testing applications is, to the best of our knowledge, a novel approach which has yet to be fully explored.

The rest of this paper is organized as follows. Section 2 discusses related works. Section 3 presents our approach. Section 4 describes our experiment and its results and Section 5 provides a discussion about the results. Finally, Section 6 provides conclusions and future work.

2 RELATED WORK

Producing a tool that will allow testing any arbitrary mobile application automatically is an extremely challenging problem, perhaps nearly as difficult as the underlying general software testing automation task. Throughout the years, there has been an extensive study in the field of testing automation of *desktop applications*. However, a recent study by Hu et al. [18] has discovered that many of the mobile application bugs are unique and tend to be different from the ones presented in traditional desktop applications, mainly due to the inherent difference in architecture and development methodologies. Thus, traditional approaches for desktop applications testing cannot be naturally translated into mobile applications.

The need for efficient mobile application testing methods has yielded many testing automation APIs, including tools like Appium[3], Selendroid[4] and Robotium[5], to name a few (a recent survey is available at [17]). These APIs allow a developer to write testing scripts in her programming language of choice, and later run these scripts over and over again to check the application in different user behavior scenarios. The main limitation of such tools is that these manually constructed scripts are coded for a designated application in mind, therefore the developer is bound to invest ample time in reusing these tests for new applications and to accommodate for changes in the functionality of existing applications.

In order to reduce the need for writing redundant testing scripts for mobile applications, which share many common characteristics, significant research efforts have been focused on the development of automated testing techniques and algorithms to allow for testing applications automatically. These techniques, which are commonly defined in the literature as automated input generation (AIG) techniques, strive to generate as much *relevant inputs* as possible for the designated application in order to explore the application with maximum coverage. This is achieved by generating user interface events, which simulate a user behavior including actions like clicks, scrolls and swipes, while searching for runtime crashes which are caused by uncaught exceptions thrown in the code.

These techniques, essentially, are seeking to perform a so-called *stress testing*, a type of mobile application testing which tests the robustness (error handling) of the application under heavy, irregular input. To our knowledge, this is the first work that tries to cope with the challenge of automating the process of Android applications *functional testing*, which, on the contrary, is intended to make sure that the application behaves as the designer expects under normal input and expected user usage. Hence, next, we mainly survey AIG techniques that aim to detect *runtime errors* and achieve good code coverage, as this is the closest research area to our work.

In this paper, as with most of recent papers in the field, we focus on the Android platform [15]. The choice to use the Android platform is mainly due to the fact that it is the most common mobile operation system on the market to date and due to its open-source nature that allows the academic community to get full access to the applications and the platform source code. Moreover, the large variety of Android models and versions on the market make the test automation task significantly important.

According to a recent study by Choudhary et al. [15], which has conducted a comprehensive overview of the main existing Android AIG tools that have been proposed and developed in academic papers, we can categorize each of these tools into one of three approaches:

- **Random exploration approach:** Tools which use this approach generate user interface events in a *random fashion*, executing them one by one on the application user interface. The main use of this approach is to test the application robustness, as most of these events are ones that the average user is not likely to perform. Random test input generators are easy to use and are particularly suitable for so-called "stress testing". On the downside of this approach, random tests are prone to get stuck with repetitive events and are not likely to get a good coverage of the application, due to their random nature. In this category we can find tools like Monkey [8], Dynodroid [23] and DroidFuzzer [28].
- **Model-based exploration approach:** Tools which use this approach build a model of the application's GUI in order to utilize it for building a sequence of user interface events that maximize the exploration coverage. The model is usually a finite state machine whose states are the application different screens and its transitions are the different possible user interface events. These tests trigger all the different possible user interface events (i.e., the finite state machine's transactions) on all the different screens (i.e., the finite state machine's states) and ends when all the user interface events that can be triggered are leading to already visited screens. The advantage of this approach is that it tends to get a good coverage of the application as triggered user behaviors are unique. The limitation of this approach is that only changes in the application GUI are reflected as new states in the model. However, many times user interface events change the internal state of the application, thus making these models miss and avoid certain exploration routes. In this category we can find tools like GUIRipper [11], A^3E-Depth-First [13] and Swifthand [14], which uses machine learning techniques to learn a model of the application's GUI and guide the generation of user input sequences based on this model.
- **Systematic exploration approach:** These tools generate unique user behaviors by dynamically analyzing the application's *source code*. The strength of this approach is that it can leverage the source code to generate tests to reveal previously uncovered application behavior. The downside of the approach is significant scalability concerns. In this category we can find tools like Sapienz [25], EvoDroid [24], A^3 E-Targeted [13] and ACTEve [12].

[3] http://appium.io/
[4] http://selendroid.io/
[5] https://github.com/RobotiumTech/robotium

All of the above tools have been successfully deployed in different experiments to have shown to produce a good coverage of the applications' state-space. Nevertheless, they all share a common prominent limitation: these tools aim to find only *technical* bugs and defects in the application, meaning real-time application crashes which are caused by uncaught exceptions thrown in the code [15]. However, many of the bugs presented in today's applications are related to the program logic (e.g., a login screen that can be bypassed without entering valid username and password, an email-composing screen that allows sending emails to an invalid email address, etc). Hu et al. [18] present an empirical study of common Android bugs. The authors find that logical bugs are about 10 times more prevalent than technical bugs.

More troubling is the relatively minor impact the above tools have on the Android developer community. The survey by Linares-Vásquez et al. [21] demonstrates that most Android developers still rely on manual testing, which is an expensive and time-consuming process. The few who do not use manual testing are relying on automation APIs which, as discussed above, is a limited automation solution. The authors go beyond the dry numbers and explore the reasons for this lack of adoption of new testing tools. As one might suspect, due to their randomness nature, these tools are only good at finding corner cases which would not normally be encountered by a user. The developers who participated in the survey claimed that these tools work as a solution only for low-quality fragile code, but rarely help to improve the quality of the application. Moreover, even runtime errors which are detected by these tools are very hard to reproduce, combined with the lack of intuitiveness in the generated event streams making debugging the software correctly highly complex. The authors conclude that for the purpose of allowing future testing tools to make a bigger impact among the Android community, researchers should consider developer preferences and workflows when designing and evaluating their approaches.

Another recent work by Linares-Vásquez et al. [22] serves as a perspective paper which survey the current state-of-the-art mobile testing tools available for developers. The authors highlight the limitations of current model-based testing approaches which have traditionally focused on *individual models* instead of *multi-model* or *domain models* representations. These more sophisticated representations are crucial for understanding an application GUI and use cases, as well as for enabling automation tests to exercise complex inputs and behaviors. They also explain that domain models should be extracted by examining common traits between applications that exist in similar categories in order to derive common event sequences and GUI-usage patterns.

In this work, we suggest to implement these recommendations by using machine learning techniques that can enable automated tools to infer the *domain model* of each of the application's screens, which, in return, allows it to derive a set of generic functional test cases for that screen which fits the model. These auto-generated functional test cases allow one to automatically test the designated application using a more genuine and human-like usage model as Android developers have been shown to prefer. As a result, the developer may discover new, previously unexplored, functional bugs in the screen. In addition, these functional test cases are easy to reproduce and have an expressive description, thus potentially making the debugging process easier. Overall, our approach can be categorized as a sub-category of the model-based exploration approach: Domain Model-based exploration. Note that companies and individuals who provide testing or consulting for mobile applications may find this approach particularly beneficial, as they often have access to similar applications in different domains.

3 APPROACH

3.1 Motivation for Activities Classification

The key element of our approach stems from the difference between testing desktop and mobile applications. While desktop applications come in an endless amount of shapes and forms, the structural scope of mobile applications is naturally more limited [3]. An Android application is, at its core, a series of different screens which are connected using user interface buttons. The official Android development guide defines each of these screens as an "Activity"[6], which is a single window in the application. An Android activity is a group of different user interface elements from the Android development kit which are organized in a hierarchic structure. While these elements vary in their specific purposes, we can categorize them into two main groups: 1) Elements which are directly visible to the user on the screen and allow him to interact with them by hand gestures, such as clickable buttons, lists of items which can be scrolled up and down and text fields; 2) Elements that are not directly visible to the user on the screen, but rather control the layout of other user interface elements in the activity, such as arranging them horizontally in a single column or vertically in a single row.

In a session titled "Structure in Android App Design" given at the Google I/O 2013 developer conference [2], Nagel and Fulcher discuss common patterns in designing activities for Android Applications. They introduce various structures of elements arrangement, explaining that using common and more simple activities structures will help making the application more predictable and understandable to the user and thus, more pleasing to use. The fact that many different Android activities share the same structure suggests that these activities may require similar treatments as for their testing. Therefore, by classifying an activity into a certain class of activities, we can derive which tests should be performed automatically. The problem can be naturally translated into a Multiclass classification problem [10], which is a common branch of machine learning. Namely, given an instance of an activity, we seek to classify it into one of seven pre-defined classes.

3.2 A Study of Activities Types

Narrowing down all of the possible activities into a finite list of types is an open question for future study, as it would require a more comprehensive study. In the scope of this work, we performed a preliminary study of 100 Android applications from the Google Play store, by manually searching for common patterns, structures and behaviors in the different activities. Based on our preliminary study, we identified 7 activity types which can be divided into two groups: 1) Activity types which have been the most common ones among the 100 studied applications. 2) Activity types which have a notable structure and a naturally-anticipated functionality.

Activity types in the first group:

[6] https://developer.android.com/reference/android/app/Activity.html

- **Splash Activity:** Splash activity is the screen displayed when opening the activity, which usually displays an image or text while the application is loading in the background. Most of the applications that we examined had a splash activity. When a splash activity exists, it is always the first screen in the application. Nevertheless, there is still a need to classify the first activity in the application. For many lightweight applications, this screen may be displayed for only a fraction of a second, which may result in incorrectly classifying the second screen of the application as a splash screen if the classification is done in a naïve way. The test for this activity will be to make sure that the application can advance from this screen to the next one.
- **Advertisement Activity:** The vast majority of the Android applications are free to download due to many reasons listed in [1]. Instead, for making profits, the Android developers commonly incorporate advertisements in their applications. These advertisements can pop up anywhere and at anytime in the application, which makes the classification challenging. Identifying these activities is important as clicking on these advertisements during testing will likely exit the designated application. Thus, after classifying an activity as an advertisement activity we need to carefully close it and make sure that we stay within the scope of the application.
- **Login Activity:** Many modern applications require a valid username and password in order to use most of the application's services. Therefore, these applications contain a screen which allows the user to enter its username and password to connect to their server(s). We designed a test for this activity that verifies that the user cannot bypass the login screen by leaving the text fields empty, or by entering incorrect credentials (e.g., using random strings). After that, the test verifies that the login screen can be passed by using valid details (the test should be provided with valid username(s) and password(s)).
- **Portal Activity:** Many of today's communications media, such as websites, newspapers and TV channels, have a designated mobile application that allows the user to access the media content on her phone. The portal activity is the "hub" screen of these applications, and thus we designed a test that verifies that the screen can be swiped left and right in order to reach different sections of the portal and that an article can be opened from this screen.

Activity types in the second group:

- **Mail Activity:** Mail applications, which are very common on mobile devices [6], have a well defined purpose with a limited number of possible actions. This allows us to design global tests for every mail application. The mail activity is the "hub" screen of these applications, with functionalities as managing the inbox mails and sending new mails. We designed a test for this activity that browses through the inbox mails, tries to open a mail from the list (at random) and scrolling through the mails content.
- **Browser Activity:** Web browsers are one of the most important applications, as they allow the user to access websites on their mobile device [9]. These applications share a very specific functionality which can be translated into a uniform test which verifies that the user can reach several websites through the activity, use the back, forward and home buttons and opening a new tab.
- **To Do List Activity:** To-do list applications usually share a common purpose which is to keep track of the user personal list of tasks. Thus, the test for this activity verifies that the user can add new tasks to the list and check them as done.

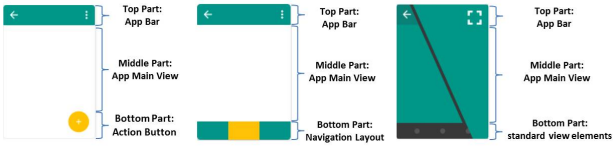

Figure 1: Screen divisions across common activities templates

In this paper we have decided to focus on these 7 common activity screens as we identified in our preliminary study. However, note that our approach can be readily amended with additional activities, using the same procedure discussed in Section 3.4. Although introducing a new activity type is not an instant process, it does pay off considering the potential reuse of testing scripts in many applications, as demonstrated in our experiment (Section 4).

3.3 Building the Features Vector

Each activity can be characterized by a large number of features, which are all related to the user interface elements it contains such as the different classes of the elements, their set of attributes, their relative location in the activity, the number of elements presented in the activity, etc. Additionally, an activity can be characterized if it contains a navigation drawer, which is a panel that displays the application's main navigation options on the left edge of the screen. It is hidden most of the time, but it is revealed when the user swipes a finger from the left edge of the screen or, by clicking on a designated button.

While constructing the features vector, we had to decide which elements are the most informative and may differ between different types of activities. In our preliminary study, as described in Section 3.2, we noticed that an activity can be identified mostly by its visible elements, namely, the elements which the user can interact with directly. This correlates to the fact that the official Android development guide specify that almost all activities interact with the user. As a result, user interactive elements are assumed to adequately represent the activity. Furthermore, by examining the basic activity templates from the Android studio activity design guideline[7], we noticed that each activity screen can be artificially divided into 3 parts: the top, the middle and the bottom. We use the following heuristic division of the screen: 20%-60%-20% from top to bottom, as depicted in Figure 1.

Therefore, we focus on the following interactive elements groups which can appear in each of the three activity's parts:

- Clickable elements: Elements that are responsive to the user touch click.

[7]https://developer.android.com/studio/projects/templates.html

- Horizontal swipeable elements: Elements that can be swiped by the user left and right.
- Vertical swipeable elements: Elements that can be swiped by the user up and down.
- Text field elements: Elements that the user can type text into them.

We use the number of elements from each of the above element groups in each of the three activity parts as the first set of features. Namely, the first group of features contains 12 features, where each represents the number of the elements of each of the 4 element categories presented above in each of the 3 parts of the screen.

The second group of features contains 2 features. The first one is the number of general elements on the screen, no matter what class they are or where they are located. The second is the number of 'long-clickable' elements on the screen, meaning elements that respond to the user holding them for a couple of seconds, such as an image element that holding it for a couple of seconds will mark the image and display options such as saving or sharing it. Based on our experience in applications development and our activities study, these features are less prominent and thus we do not divide them into different parts of the screen.

The final group contains one feature, which is a boolean variable set to true if the activity contains a navigation drawer. We can determine if an activity contains a drawer by checking if it contains an element of class DrawerLayout, which is the default navigation drawer the Android development kit provides. However, this feature is still hard to derive as some applications implement a different drawer than the default one, and as a consequence they do not contain a DrawerLayout element. To overcome this problem, when we are scanning for clickable elements on the screen, we check the resourceID of each one of them. If the resourceID contains one word from a pre-defined constant list of words that indicates a drawer button, such as "drawer", "menu", "sidebar" and so on, we assume that this button opens a drawer menu in the activity and thus the activity contains a drawer.

Overall, we use 15 features as described above.

3.4 Constructing a Dataset

In order to construct a dataset to train and test our classifier, as discussed in Section 3.5, we needed to obtain a large set of Android applications' activities and perform feature extraction and labeling. Extracting the features of an activity is a hard task, due to the fact that many of the elements in an activity are invisible and cannot be identified just by observing the activity display on the device. To overcome this problem, we use the "TestProject Elements Spy" tool, which allows developers to scan and inspect the user interface elements of an Android activity. We have implemented an automated script which extracts the features of a given Android application, as defined in Section 3.3, using the Elements Spy tool and saves them to the dataset file in a textual format. Although the extraction of the features was automated, building a dataset of activities was still a long process, as it took significant time to connect into the Appium Server, load up the application on the device and extract the features. We searched the Google play store for relevant applications by using appropriate search terms (e.g., 'Mail', 'Browser', 'To do', etc) and picking the ones with the largest number of downloads. We then download each application into our device, and manually labeled each of the different activities to the activities types we have defined. Most of the activities screens were easily classified to one of the types we have defined, the rest are of different types which we did not model in this paper, and therefore were omitted. This process resulted in a dataset consisting of 80 activities, taken from 50 different applications from the Google play store. A pie chart representing the relative distribution of the activities types in the dataset is shown in Figure 2.

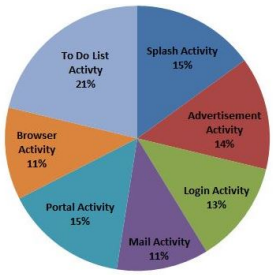

Figure 2: Relative distribution of the activities types in the dataset

Classifier	Accuracy
Decision Tree	63.75%
K-Nearest Neighbours	77.5%
Logistic Regression	77.5%
Random Forest	82.5%
Multi-Layer Perceptron	83.75%
KStar	86.25%

Table 1: Accuracy of activity type prediction using different classification models.

3.5 The Classifier

In order to construct a classifier we used Weka [7] which is a suite of machine learning software written in Java and is widely used in the machine learning community. In order to train our classifier, we ran a 10-fold classification process on our dataset with different classification algorithms, while measuring the accuracy of each one. Using a 10-fold validation procedure, we built 10 classifiers from 10 equal sized splits of our dataset, each one is divided into 90% training data and 10% test data. Then, we did an average of the performance of these 10 classifiers on the whole dataset. The final model we are keeping is a one based on the whole dataset. Table 1 shows this averaged accuracy of the 10 classifiers on the dataset over activity type prediction using different classification models:

As we can see from Table 1, using the instance-based KStar classifier [16] we have managed to achieve a high classification accuracy of 86.25%, while other classic methods such as decision trees or multi-layer perceptron averaged only 77% accuracy. Thus, our model of choice for this work is KStar. KStar uses an entropic distance measure as a similarity function to determine which of the training instances are the most similar to the test instance. We performed a grid-search over the possible k parameter values and found that $k = 20$ produced the best results.

3.6 Creating Generic Functional Test Cases

Correctly classifying an activity to an activity type enables one to derive what it is the "expected behavior" from the activity. For example, a *login activity* is normally expected to advance to the next activity only once the user fills in a valid username, a valid password and clicks on the sign-in button. While conducting our preliminary activities types study, as described in Section 3.2, we came to realize that although each instance of a certain activity type can be different (i.e., different elements' position, elements' size and specific flow of events), they all follow a rigidly defined functionality. This means that given the correct classification of an activity we can derive "generic" functional test cases for it. Interestingly enough, we have also noticed that each of the identified functionality is reflected in the GUI in roughly the same way. This means that classifying an activity not only allows us to derive potential generic test cases, but also determine the expected GUI change which follow it. For example, classifying an activity as a login activity allows us to derive a test case in which we click on the sign-in button without filling in a username and a password, and determine that the correct outcome for this test case should be that no other activity can be reached unless a valid username and a valid password have been filled by the user.

With this idea in mind, we have implemented several functional test cases for each activity type and formulated the expected GUI changes that should follow each of them. These test cases come from our activities types study, in which we listed all of the functionalities we identified to be repeat in activities sharing the same type. Table 2 gives an extensive overview of the ACAT tool's functional test cases, providing the GUI events each test triggers and the expected GUI changes, which the tool is checking in order to determine the success\failure status of each test case. Note that the "Compose Mail Activity" and "Portal's Article Activity" were not part of the classification model as there is no need to involve our classifier to identify them. For example, classifying an activity as a "Mail Activity" allows us to automatically derive that the activity which follows clicking the compose new mail button will be "Compose Mail Activity".

Designing activity tests is a complex process. These tests cannot be hard-coded for a specific application in mind as they should fit different activities which share the same type in as many as possible different applications. Different programmers develop different applications, each have her own way and style of designing the application. For example, the developer may refer to different elements on the screen by using varying *resourceID*s. As a result, when manually coding a test for an Android application, referring to the correct resourceIDs is not a problem. Unfortunately, this is infeasible in our case as we propose tests for general activities which we do not know the resourceIDs for their elements in advance. In order to overcome this problem, we built lists of associative words for different elements we expect to find in a test. When searching for a specific element on the screen, e.g., a close button for an advertisement, we iterate over the clickable elements on the screen. For each clickable element we check if its resourceID contain at least one word from the list: ["close","discard","shut","hide","no"]. If so, we assume that this button is the close button of the advertisement. From our experiments (see Section 4), this method is proving to be very efficient, as the resourceIDs developers give their elements tend to be very predictable. The rational for the above is that developers themselves want to give informative names to their elements, as this will help simplify the code maintenance.

Algorithm 1 ACAT's Main Algorithm

```
1:  procedure AutomatedFunctionalTesting(firstActivity)
2:      currentActivity ← firstActivity
3:      while currentActivity has not been tested do
4:          activityFeatures ← extractActivityFeatures(currentActivity)
5:          activityClassification ← classifyActivity(activityFeatures)
6:          activityGenericTest ← deriveGenericTest(activityClassification)
7:          for each functionalTestCase ∈ activityGenericTest do
8:              testCaseSteps ← functionalTestCase.getTestCasePreliminarySteps()
9:              for each step ∈ testCaseSteps do
10:                 step.executeStep()
11:             assert functionalTestCase.getTestCaseExpectedGUIChange()
12:         currentActivity ← getCurrentActivity()
13:     return Test'sReport
```

Algorithm 1 shows the skeleton of the main algorithm of ACAT. It takes as input the first activity of an application and keeps running on each new discovered activity. For each of these reached activities, the algorithm first extracts the features described in Section 3.3, classifies the activity based on these features using the KStar classifier which is described in Section 3.5 and selects the Generic Test which is associated with this activity classification. Then, for each of the individual functional test cases contained in this general test, which are described in Table 2, the algorithm first executes the preliminary steps of this case and then asserts the current GUI state against the expected GUI change of the current test case. Finally, the algorithm returns a comprehensive report which includes the activities classifications, the steps executed on each activity, the functional test cases failed/passed status and an explanation for those test cases which failed (the expected GUI change against the GUI received during the test).

4 EMPIRICAL EVALUATION

In this empirical evaluation, our first and most important goal is to demonstrate the ACAT's ability to discover real functional bugs presented in applications from the Google Play Store. However, applications usually go through rigorous testing before they are uploaded to the application store, meaning that it is unlikely that our tool would be able to detect overlooked functional bugs. Therefore, we have designed a three-part experiment which will allow us to demonstrate this point: 1) An experiment which measures the ability of ACAT to execute the generic functional test cases on different applications; 2) An experiment which measures the ability of ACAT to detect "planted" bugs in open source applications; 3) An experiment which measures the ACAT ability to automatically cover human testers' work.

4.1 Part 1 - Generic Functional Test Cases Execution Evaluation

4.1.1 Experimental Design. We randomly picked 24 Android applications from the Google Play Store, 6 applications from each application category presented in our model (i.e., mail clients, news and magazines, browsers and productivity applications). These applications were not part of our training dataset used for training our

Activity Type	Test Description	Test GUI Events	Expected GUI Change
Splash Activity	Users can advance the splash activity to the application main activity or view	1. Wait for activity or view to change	Activity name or GUI changed after some time
Ad Activity	Users can close the advertisement	1. Click on the close button	Activity name or GUI changed after clicking the close button
Login Activity	Users cannot advance the login screen without filling in a username and a password	1. Click on the login button	Activity name remained the same after clicking the login button
	Users cannot advance the login screen with an invalid username and an invalid password	1. Fill in a random username 2. Fill in a random password 3. Click on the login button	Activity name remained the same after clicking the login button
	Users can advance the login screen with a valid username and a valid password (a valid combination of username and password must be supplied by the user to the tool as an input argument)	1. Fill in a valid username 2. Fill in a valid password 3. Click on the login button	Activity name changed after clicking the login button
Mail Activity	Users can browse through the inbox mails	1. Swipe up the list of mails 2. Swipe down the list of mails	GUI changed after swiping the list
	Users can open a mail from the inbox mails	1. Click on a random item in the mails list	GUI changed after clicking on a mail
	Users can open the compose new mail form	1. Click on the compose new mail button	Activity name changed after clicking on the compose new mail button
Compose Mail Activity	Users cannot send a mail without filling in a recipient mail address	1. Click on the send mail button	Activity name remained the same after clicking the send button
	Users cannot send a mail with an invalid recipient mail address	1. Fill in an invalid recipient mail address (e.g., InvalidAddress@gmail) 2. Click on the send mail button	Activity name remained the same after clicking the send button
	Users can send a mail with a valid recipient mail address (the user's self mail address must be supplied by the user to the tool as an input argument)	1. Fill in a random unique ID in the mail's subject 2. Fill in the user's self mail address as the recipient mail address 3. Click on the send mail button 4. Refresh the mails inbox list by swiping it down 5. Wait for a couple of seconds 6. Search for the random unique ID entered before in the inbox mails' subjects	The unique ID entered in the mail's subject was found in the inbox mails list
Portal Activity	Users can browse through the articles list	1. Swipe up the list of articles 2. Swipe down the list of articles	GUI changed after swiping the list
	Users can switch between the portal's sections	1. Swipe left the center of the screen 2. Swipe right the center of the screen	GUI changed after each click
	Users can open an article	1. Click on an article name from the list	Activity name or GUI changed after clicking the article name
Portal's Article Activity	Users can scroll the article up and down	1. Swipe up the article's content 2. Swipe down the article's content	GUI changed after swiping the article's content
	Users can share the article using other social applications	1. Click on the share article button	Activity name or GUI changed after clicking the share article button
	Users can return to the portal's main hub	1. Click on the back navigation button	Activity name or GUI changed after clicking the back navigation button
Browser Activity	Users can access the web using the activity	1. Fill in a valid website URL in the address bar text field 2. Click enter 3. Wait for a couple of seconds	GUI changed after timeout
	Users can scroll the web page up and down	1. Swipe up the web page 2. Swipe down web the web page	GUI changed after swiping the web page
	Users can return to the previous page	1. Click on the back navigation button	Activity name or GUI changed after clicking the back navigation button
	Users can open a new tab	1. Click on the tabs button 2. Click on the add new tab button	GUI changed after clicking the add new tab button
To Do List Activity	Users can browse through the tasks	1. Swipe up the tasks list 2. Swipe down the tasks list	GUI changed after swiping the tasks list
	Users can add tasks to the tasks list	1. Fill in a task name in the text field 2. Click the add task button	GUI changed after clicking the add task button
	Users can check tasks from the tasks list	1. Click on the check task button	GUI changed after clicking the check task button

Table 2: Description of all the functional tests implemented in the ACAT tool

model, nor were they used as references in the implementation process of our tool. In this part of the experiment, we seek to evaluate the ability of our tool to successfully translate the generic functional test cases scripts to new unseen applications. While conducting this experiment we would also want to test the classification accuracy of our model on this new set of activities.

We ran the ACAT testing tool on the applications and stored it's auto-generated reports. Finally, we extracted the results of each run's report, which includes the classification of each activity, the number of functional test cases successfully executed and a short description for each of the tools actions invoked. Furthermore, for each functional test which could not be executed, we manually extracted the reason for its failure for statistical analysis of the errors' source.

Application Category	Application Name	Classification Accuracy	Functional Test Cases Executed Percentage
Mail Clients	Email TypeApp	100%	100%
	MailDroid	100%	66%
	Email mail box	100%	33%
	Zoho Mail	100%	83%
	Yandex.Mail	100%	66%
	Email - email mailbox	100%	100%
News and Magazines	CNN	100%	83%
	Sky News	100%	100%
	Mirror	100%	100%
	USA Today	0%	————————
	CBS News	100%	100%
	Euronews	100%	100%
Browsers	APUS Browser	0%	————————
	Fastest Mini Browser	100%	100%
	Browser for Android	100%	100%
	Boat Browser	100%	75%
	Mercury Browser	100%	75%
	DU Browser	100%	50%
Productivity	MyLifeOrganized	100%	33%
	TODOList	100%	66%
	Simple To Do	0%	————————
	Checklist	0%	————————
	Listing it!	100%	33%
	Tasks: Todo list	100%	100%

Table 3: Classification accuracy and tests executed percentage of the applications in the experiment

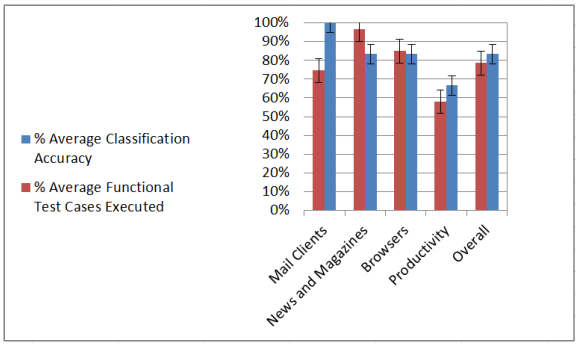

Figure 3: Classification accuracy and functional test cases executed percentage average among the different applications categories

4.1.2 Results. Table 3 and figures 3 and 4 present the results of the experiment. We identify 3 major trends: 1) The overall classification accuracy average, 83%, is consistent with the 86% accuracy achieved by our model in the initial evaluation. 2) The overall functional test cases executed properly percentage is high, averaging 79% of applicable functional tests which were reused and executed successfully on these new applications. 3) The functional test cases which could not be reused properly failed to be executed only because they were not generic enough. This failure is rarely attributed to the inability of the ACAT to successfully identify the activity's desired functionality.

4.2 Part 2 - Detecting Planted Bugs Evaluation

4.2.1 Experimental Design. As discussed before, we cannot expect to find functional bugs in applications which are available to download on the Google Play Store. Therefore, for this part of the experiment, we used 2 open source Android applications in which we artificially "planted" bugs. These applications were not used in this study thus far.

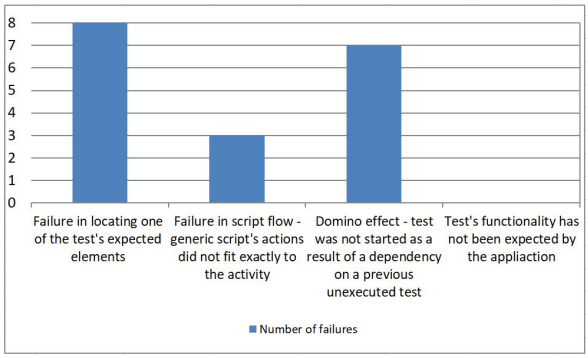

Figure 4: Number of Functional tests execution failures among the different failures classification

- **"K-9Mail"** - An email client application. We focused on the following activities:
  - "**MessageList**" - A mail activity which contains the following bugs:
    - A user cannot open an email's content from the inbox list.
    - A user can send an email without recipient's address.
    - A user cannot send a valid email.
    - A user can send an email with an invalid recipient's address (this bug already existed in the original code).
  - "**setup.AccountSetupBasics**" - A login activity which contains the following bugs:
    - A user can sign in without filing in a username and a password.
    - A user can sign in with an invalid username and an invalid password.
    - A user cannot sign in with a valid username and a valid password.
- **"CrimeTalk Reader"** - A portal application to browse "CrimeTalk" articles. We focused on the following activity:
  - "**MainActivity**" - A portal activity which contains the following bugs:
    - A user cannot swipe the screen left and right in order to browse the portal's different sections.
    - A user cannot click on the menu's different tabs in order to browse the portal's different sections.
    - A user cannot open an article from the activity.

We ran the ACAT on the original activity, which has not been tampered with, and on the faulted version thereof. Finally, we extracted the results of each run's auto-generated report, which includes the number of functional bugs discovered, as well as their description, and the classification of each activity as well.

4.2.2 Results. Table 4 presents the results of the experiment. While examining them, we can identify 2 major trends: 1) The ACAT was able to classify correctly the 3 unseen activities. 2) The ACAT managed to discover all of the "planted" functional bugs. Moreover, the ACAT was able to find a functional bug (A user can send an email with an invalid recipient's address) which was already part of the original version of the application, without tampering with

Application and Activity Name	Original/Faulted	Activity Classified as:	Functional Bugs Discovered
K-9Mail: MessageList	Original	Mail Activity	1 bug was found: # Sending an email with invalid recipient address – Failed
	Faulted		4 bugs were found: # Opening an email from the inbox mails list – Failed # Sending an email without recipient address – Failed # Sending an email with invalid recipient address – Failed # Sending and receiving a valid mail – Failed
K-9Mail: setup.AccountSetupBasics	Original	Login Activity	No bugs were found
	Faulted		3 bugs were found: # Login without username and password – Failed # Login with wrong username and password – Failed # Login with valid username and password – Failed
CrimeTalk: MainActivity	Original	Portal Activity	No bugs were found
	Faulted		3 bugs were found: # Browsing through the portal's sections by swiping it left and right – Failed # Switching between the portal's tabs – Failed # Opening an article – Failed

Table 4: Functional bugs discovered in the different activities by the ACAT tool

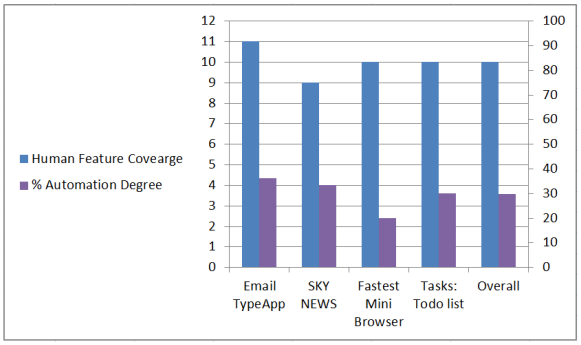

Figure 5: Human feature coverage and automation degree among the different applications

its code (it worth to mention that this bug also existed in 2 of the mail client applications used in the first part of the experiment, which the ACAT successfully discovered). In the rest of the original activities, it has correctly proclaimed that there are no functional bugs.

4.3 Part 3 - Feature Coverage Evaluation

4.3.1 Experimental Design. Evaluating the effectiveness of an automated mobile testing approach by code coverage has been widely considered a standard practice by researchers and practitioners alike. However, recent studies ([19], [20], [29]) have started to question the effectiveness of code coverage as a test's quality measuring metric, a suspect which has been confirmed by Linares-Vásquez et al. [21]. In their survey, the authors found that code coverage is not perceived by Android developers as an important measure of a test's quality. The developers point out *feature coverage* (i.e. the number of an application's features a test covers) as a better metric which can capture the effectiveness of a test. In light of these findings, for this part of the experiment we seek to evaluate our approach using feature coverage while comparing it to manual testing – the most common testing practice to date [21].

For this part of the experiment, we recruited two human participants, a male aged 26 and a male aged 20, with proven experience in developing Android applications and are familiar with the Android infrastructure. Both participants are students at Bar-Ilan University (Israel), one is a graduate student and the other is an undergraduate student, yet both of which are majoring in Computer Science.[8] Each participant has been presented with the same 4 installed applications, taken from the first part of the experiment (Section 4.1): Email TypeApp, Sky News, Fastest Mini Browser and Tasks: Todo list, one from each category. The participants have also been given the description of each application taken from the Google Play Store. Participants were instructed to assume the role of a human tester and were asked to perform manual testing for the given applications. Specifically, participants were requested to explore the applications' functionality without any time restriction and list the features that they have tested.

[8]Both students do not co-author this paper.

In order to perform a meaningful interpretation of the results we introduce the following notation:

$$HumanFeatureCoverage = Tester1Features \cup Tester2Features$$

$$HFC = HumanFeatureCoverage, TFC = ToolFeatureCoverage$$

and define

$$AutomationDegree = \frac{TFC \cap HFC}{HFC}$$

Specifically, for each examined application we calculate the automation degree, measured as the ratio of feature coverage achieved by the intersection of ACAT and human, to the total feature coverage achieved by human.

4.3.2 Results. Figure 5 presents the results of the human experiment. The left Y-axis shows the human feature coverage (quantity of features), and the right Y-axis shows the automation degree (in percentage). The automation degree of the ACAT varies from 20% to 36%, with a mean of 30% and a standard deviation 7%. These results indicate that the ACAT automatically covered a large portion of the human testers' work. Specifically, these results suggest that about a 1/3 of the manual testing time could be potentially reduced, which is a significant relief in the manual testing efforts.

5 DISCUSSION

Our results depicted in Section 4.1.2, Section 4.2.2 and Section 4.3.2 demonstrate the potential of our approach. This experiment shows that our proposed approach, as implemented by the ACAT tool, can successfully perform functional testing to new applications without prior knowledge about them. This is contributed to two main ideas implemented in our tool:

- The activity classification approach, which enables this tool the power to derive activity-based functional test cases for examining the activity's expected behavior. Note that these tests can only be executed in the correct activity.
- The generic functional testing scripts which translate these expected behaviors into application-specific testing actions. This translation is a challenging concept due to the inherent differences between applications. Nevertheless, we have managed to solve this difficulty by implementing the approach describes in Section 3.6. The first part of our experiment support this claim while Table 5 further clarifies this concept by showing the relation of generic script entities to

Test Description	Test Actions		
	Generic Application	Application #1 - TypeApp	Application #2 - Email - email mailbox
Users can browse through the inbox mails	1. Swipe up the list of mails 2. Swipe down the list of mails	1. Swiped up an element with resource ID: message_list 2. Swiped down an element with resource ID: message_list	1. Swiped up an element with resource ID: list 2. Swiped down an element with resource ID: list
Users can open a mail from the inbox mails	1. Click on a random item in the mails list	1. Clicked on a button with resource ID: message_list	1. Clicked on a button with resource ID: list
Users can open the compose new mail form	1. Click on the compose new mail button	1. Clicked on a button with resource ID: new_mail_tab	1. Clicked on a button with resource ID: action_compose_navi
Users cannot send a mail without filling in a recipient mail address	1. Click on the send mail button	1. Clicked on a button with resource ID: sendmail	1. Clicked on a button with resource ID: icon_send_mail
Users cannot send a mail with an invalid recipient mail address	1. Fill in an invalid recipient mail address 2. Click on the send mail button	1. Typed: "Testing@InvalidAddress" a text field with resource ID: RecipientEditTextView_to 2. Clicked on a button with resource ID: sendmail	1. Typed: "Testing@InvalidAddress" in a text field with resource ID: to 2. Clicked on a button with resource ID: icon_send_mail
Users can send a mail with a valid recipient mail address	1. Fill in a random unique ID in the mail's subject 2. Fill in the user's self mail address as the recipient mail address 3. Click on the send mail button 4. Refresh the mails inbox list by swiping it down	1. Typed: "Hello Mate! Mail_Unique_ID:5248" in a text field with resource ID: subject 2. Typed: "orelzang@gmail.com" in a text field with resource ID: RecipientEditTextView_to 3. Clicked on a button with resource ID: sendmail 4. Swiped down an element with resource ID: message_list	1. Typed: "Hello Mate! Mail_Unique_ID:688647" in a text field with resource ID: subject 2. Typed: "orelzang@gmail.com" in a text field with resource ID: to 3. Clicked on a button with resource ID: icon_send_mail 4. Swiped down an element with resource ID: list

Table 5: Relation of generic script entities to specific application's elements

specific application's elements (from our experiment). The second part of the experiment demonstrate that in a case of existing functional bugs, these testing scripts can indeed discover them and present them to the developer in the auto-generated report.

In addition, our experiment demonstrates another underlying idea behind our approach; instead of testing an application as a "whole unit", as done by previous works (see Section 2), it might be better, or at least grant a certain advantage, to consider each activity in the application on its-own, a "self-entity" if you will, with specific desired functionality. Thus, an application testing could be viewed a series of scenario tests, designed for each activity on its own.

Recall the about 20% of the functional tests could not be executed or failed in the process. Deeper investigation into these failures reveals an interesting insight: testing scripts failed only because they were not generic enough. Namely, they failed due to their inability to locate the test's required elements in the activity or a wrong estimation of the test's flow (e.g., a portal's article activity in which the share article button is only visible after opening a certain menu). Specifically, *none* of them failed as a result of an activity which was not supposed to meet the test's desired functionality in the first place, which supports our underlying assumption that activities which share the same pre-defined type should share the same expected functionality.

When presenting a new approach, it is worth to discuss its limitations. Since a system may have an infinite number of possible runs, checking the behavior of an application against our expectations is limited to those executions that we actually carry out. Thus, our approach is limited to the activity types and the functionalities which have been pre-defined. The 7 activities types identified in this work were developed for a "proof of concept" of our approach, the full intended product will contain more types and tests. Additionally, one should consider the preliminary scope of our experiment (26 applications), as well as the fact that the evaluated applications were downloaded from categories which were expected to fit our model. In order to mitigate these shortcomings we are currently working with TestProject[9] R&D team to expand our approach for more activities types, along with enabling the testing algorithm to be less dependent on hard-coded test cases.

6 CONCLUSIONS

This paper introduces a novel approach for automating Android applications functional testing using machine learning techniques. The use of such machine learning enabled us to classify each of the application activities into a specific type, which in turn allowed us to successfully test various expected functional behaviors of the different screens. We tested our tool on 26 randomly selected applications, demonstrating its ability to automatically perform functional testing by translating generic testing scripts into application-specific actions. We also compared our tool to experienced human mobile testers and found out that it can automatically cover a large portion of the human testers' work suggesting a significant potential relief in the manual testing efforts. Our tool, which we named ACAT, is shown to find more functional bugs in an application, as opposed to only real-time crashes, which brings about new possibilities for developing more sophisticated testing tools. To our knowledge, this is the first attempt to develop a machine-learning-based automatic tool for testing such functionalities. Functional testing is a task which can so far only be done by hard coding these tests for each new application, or worse by manual testing, as it requires higher level of reasoning than developing automated dynamic analysis tools. We are currently working with TestProject in order to integrate the ACAT tool in the Project framework, utilizing their database of thousands of mobile applications patterns. The ACAT tool will be available to install as an add-on via TestProject Add-ons store.

[9] https://blog.testproject.io/

ACKNOWLEDGMENTS

This paper is an extended version of a short HVC demonstration paper [26] and a full arXiv paper [27].

REFERENCES

[1] 2013. The History of App Pricing, And Why Most Apps Are Free. http://flurrymobile.tumblr.com/post/115189750715/the-history-of-app-pricing-and-why-most-apps-are. (2013).
[2] 2013. Structure in Android App Design. https://www.youtube.com/watch?v=XpqyiBR0lJ4. (2013).
[3] 2015. The Simplified Open Framework for Innovative Android Applications - Structure of an Android App. http://sofia.cs.vt.edu/sofia-2114/book/chapter2.html. (2015).
[4] 2017. Digital in 2017: Global Overview. https://wearesocial.com/special-reports/digital-in-2017-global-overview. (2017).
[5] 2017. Number of apps available in leading app stores as of March 2017. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. (2017).
[6] 2017. Ten best email apps for Android. http://www.androidauthority.com/best-android-email-apps-579368/. (2017).
[7] 2017. The Weka collection of machine learning algorithms for data mining tasks. http://www.cs.waikato.ac.nz/ml/weka/. (2017).
[8] 2018. The Monkey UI Android testing tool. http://developer.android.com/tools/help/monkey.html. (2018).
[9] 2018. Ten best Android browsers of 2018. http://www.androidauthority.com/best-android-browsers-320252/. (2018).
[10] Mohamed Aly. 2005. Survey on multiclass classification methods. *Neural Networks* 19 (2005).
[11] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. 2012. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 258–261.
[12] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 59.
[13] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 641–660.
[14] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI testing of Android apps with minimal restart and approximate learning. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 623–640.
[15] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for Android: Are we there yet?. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 429–440.
[16] John G Cleary, Leonard E Trigg, et al. 1995. K*: An instance-based learner using an entropic distance measure. In *Proceedings of the 12th International Conference on Machine learning*, Vol. 5. 108–114.
[17] Jerry Gao, Xiaoying Bai, Wei-Tek Tsai, and Tadahiro Uehara. 2014. Mobile application testing: a tutorial. *Computer* 47, 2 (2014), 46–55.
[18] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM, 77–83.
[19] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 435–445.
[20] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 654–665.
[21] Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2017. How do developers test android applications?. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 613–622.
[22] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 399–410.
[23] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 224–234.
[24] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 599–609.
[25] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 94–105.
[26] Ariel Rosenfeld, Odaya Kardashov, and Orel Zang. 2017. ACAT: A Novel Machine-Learning-Based Tool For Automating Android Application Testing. In *Haifa Verification Conference*. Springer, 213–216.
[27] Ariel Rosenfeld, Odaya Kardashov, and Orel Zang. 2017. Automation of Android Applications Testing Using Machine Learning Activities Classification. *arXiv preprint arXiv:1709.00928* (2017).
[28] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. Droidfuzzer: Fuzzing the Android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*. ACM, 68.
[29] Yucheng Zhang and Ali Mesbah. 2015. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 214–224.