Automated Test Selection for Android Apps Based on APK and Activity Classification

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 24, 2020, accepted October 5, 2020, date of publication October 8, 2020, date of current version October 26, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3029735

Automated Test Selection for Android Apps Based on APK and Activity Classification

LUCA ARDITO[1], (Member, IEEE), RICCARDO COPPOLA[1], (Member, IEEE), SIMONE LEONARDI[1], MAURIZIO MORISIO[1], AND UGO BUY[2]

[1]Department of Control and Computer Engineering, Politecnico di Torino, 10138 Turin, Italy
[2]Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607, USA

Corresponding author: Simone Leonardi (simone.leonardi@polito.it)

ABSTRACT Several techniques exist for mobile test automation, from script-based techniques to automated test generation based on GUI models. Most techniques fall short in being adopted extensively by practitioners because of the very costly definition (and maintenance) of test cases. We present a novel testing framework for Android apps that allows a developer to write effective test scripts without having to know the implementation details and the user interface of the app under test. The main goal of the framework is to generate adaptive tests that can be executed on a significant number of apps, or different releases of the same app, without manual editing of the tests. The frameworks consists of: (1) a Test Scripting Language, that allows the tester to write generic test scripts tailored to activity and app categories; (2) a State Graph Modeler, that creates a model of the app's GUI, identifying activities (i.e., screens) and widgets; (3) an app classifier that determines the type of application under test; (4) an activity classifier that determines the purpose of each screen; (5) a test adapter that executes test scripts that are compatible with the specific app and activity, automatically tailoring the test scripts to the classes of the app and the activities under test. We evaluated empirically the components of our testing framework. The classifiers were able to outperform available approaches in the literature. The developed testing framework was able to correctly adapt high-level test cases to 28 out of 32 applications, and to reduce the LOCs of the test scripts of around 90%. We conclude that machine learning can be fruitfully applied to the creation of high-level, adaptive test cases for Android apps. Our framework is modular in nature and allows expansions through the addition of new commands to be executed on the classified apps and activities.

INDEX TERMS Android testing, test selection, app classification.

I. INTRODUCTION

User interface and functional testing are notoriously tedious and costly processes. Even when automated testing techniques are adopted, testers have to manually write test scripts susceptible to human errors and requiring constant maintenance, to be aligned with the evolution of the Software Under Test (*SUT*). When a full test suite has been written, small changes in the user interface or the functionality of the Application Under Test (AUT) may make it necessary to correct or re-design a significant number of test cases to ensure that their behaviour is still correct. In the Android domain, this issue is exacerbated by the rapid refresh cycle of apps and their GUIs, fueled by the constant evolution of the Android operating system and its design guidelines. Several empirical works in literature have highlighted the amount of maintenance typically required by Android test suites [1]–[3]. In contrast, several survey-based studies have linked the intrinsic difficulties in test script development to the relatively low adoption of scripted testing frameworks among Android developers [4], [5].

According to many development best practices, Android applications have to follow several standard patterns in their design and functional structure [6]. Research has also tried to identify semantic categories of applications, according to the structure and components inside the screens (or *Activities* in Android parlance). The characteristics of the app GUIs can allow to create clusters of application types and screen categories, to abstract the application behaviour and exploit such high-level description for the creation of generalized test cases [7], [8].

The experience reports from the literature of the Android domain suggest that mobile testers would benefit from a generalization of test scripts, to untie them from the implementation details of individual applications. High-level test scripts can be adapted to different apps of the same typology;

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

moreover, they can be run against the same app even if the implementation details change during its lifecycle.

Therefore, our work aims at building a novel testing framework capable of allowing a developer user to write straightforward, robust and adaptive tests that can be launched against different apps based on its class, without having to manually re-adapt them to the specific properties of the individual GUI. To that extent, we define a novel test scripting language, offering 33 high-level commands generalizing (sets of) common interactions with typical Android apps.

The framework leverages a State Graph Model of the GUI built through a systematic exploration of the app's components, and machine learning algorithms to identify the class of the application and to classify every activity that is met. In contrast with other Android test scripting tools (such as Appium [9] or Espresso [10]), which require the developer to identify explicitly the elements of the GUI that have to interact in each test case, our framework supports writing down app-independent commands that relieve the tester from exploring every screen's layout structure.

The main contribution of this paper is an extensible and low-weight framework for the development of high-level, readable and application-independent test suites. As a secondary contribution, we provide two novel means of classifying Android apps and Activities, enhancing—in terms of performance–state-of-the-art classifiers.

The remainder of the paper is organized as follows: Section II provides background information about the Android layout and activity structure, the different testing strategies for Android apps, and related work on machine learning strategies to classify Android app. Section III provides a high-level description of the testing framework that we propose. Section IV provides details about the individual components of the framework: the State Graph Modeler, the APK classifier, the Activity classifier, and the Test Adapter. Section V reports the results of the evaluations taht we performed for all the components of the architecture, in isolation. Section VI discusses the results and the current limitations of the approach. Section VII gives our conclusions and outlines future work directions.

II. BACKGROUND AND RELATED WORK

This section provides background information about the structure of typical Android apps and the available approaches for modelling and testing them. We also discuss similar findings in the literature, highlighting the differences with our work and the existing gaps in the literature.

A. ANDROID APPLICATION STRUCTURE

Android apps can be divided into three different categories, according to the target deployment platform. First, *Native* apps are developed using components that are part of the Android framework. These apps are deployed and run on Android devices only. Second, *Web-based* apps are typical are intended to be loaded and displayed on a mobile browser. Third *Hybrid* apps contain native components that can be used to load web-based contents at run time.

Native Android apps are made up of several components, each conforming to a specific lifecycle driven by the Android operating system. Four kids of components exist: *Activities*, *Content Providers*, *Services* and *Broadcast Receivers*. The latter three components are responsible for managing data, operations in the background, and message exchange between different apps. Activities are the main components of Android apps since they are in charge of populating the user interface with the required widgets. The population of the user interface is performed according to specifications that are given via code, in the Activity class itself, or via static XML layout files, that are provided in a specific folder of an Android project. The GUIs of Android apps are consist of so-called *Views*, arranged on the screen according to different Layouts; each view is provided with a set of properties that govern the displaying of the view. Views (e.g., buttons) can be associated with callbacks executed in response to user interactions (e.g., button presses).

The Android PacKage file (APK) is a compressed archive (similar to a jar or zip archive) containing a compiled program for Android and additional assets belonging to the app, such as widgets, screen layout specifications and strings in different languages (for localization purposes). The APK, digitally signed with a certificate, is the format in which the applications are distributed for the Android system. The most important files and folders contained in the archive are summarized in Table 1.

Every APK must contain a file named *AndroidManifest.xml*, which contains essential information about the app itself that is needed by various agents, such as the Google Play store, the build tools, and the Android operating system. Some of the most critical items described in the manifest file include the hardware requirements needed to run the app, the package name (which is generally the same as the project namespace), and details about the list of components (including activities) in the application. Most importantly, the manifest file contains information about the *permissions* that the app needs from the Android operating system. Permissions are needed to have controlled access to some features or areas of the device that are deemed as sensitive or vulnerable (e.g., the user's emails and contacts, the device's camera, etc). For many of the required permissions, the user is prompted to manually consent to their use when the permission is needed.

B. TESTING STRATEGIES FOR ANDROID APPS

Several testing approaches can be applied to mobile applications. Linares-Vasquez *et al.* gave a characterization of the current state of the art of Android testing, dividing testing tools and techniques into three different categories [5].

Automation APIs/Frameworks provide testers/developers with interfaces for writing GUI tests with manually-written scripts with a JUnit-like syntax. These testing techniques give full control to the testers for exercising complex use cases against the app GUI, at the cost of high development and

TABLE 1. Content of a decompiled Apk file.

Name	Description
assets	A directory containing all the multimedia files required by the app
res	A directory containing all the resource files, such as drawables, string definitions and layout XML files
lib	A directory containing compiled libraries used by the app
META-INF	A directory containing the app signature and other meta-information
AndroidManifest.xml	A file describing essential information about the app (e.g., OS compatibility and permissions)
classes.dex	The compiled Android application code file
resources.arsc	A file with precompiled strings, colors and styles to optimize performance

maintenance costs. Test cases written with Automation APIs are also significantly prone to fragility issues, meaning that they need relevant maintenance effort when the application is subject to typical maintenance [3]. Examples of Automation APIs and Frameworks are the official Espresso [11] and UI Automator [12] tools, Calabash [13], Ranorex [14], Robolectric [15] and Robotium [16].

Record and Replay tools relieve testers from manually encoding sequences of interactions into test scripts, enabling the automated recording of testing sequences, that can be re-executed at any time. However, these tools expose several limitations because they cannot provide significant defect-finding power if the developers manually insert no assertions or if the testing environment has no knowledge of the user interface that is tested. Even though no script development effort is needed by Record and Replay tools, the tests are not runnable against other AUTs if not explicitly adapted. Examples of Record and Replay testing tools for Android apps are the Espresso Test Recorder (which comes embedded in recent releases of the Android Studio IDE) [10], RERAN [17] and Barista [18].

Automated test input generation techniques have been largely investigated by the most recent studies in the literature. These approaches automatically generate sequences of inputs with the app, with a particular goal (e.g., achieving high coverage or finding a high number of defects). Automated test input generation techniques can be further divided into three different categories:

- **Random Exploration Strategy**: Random independent UI events are generated and applied to the application under test to find some faults. The advantage of this method is that it is possible to generate many test sequences with little effort, making it suitable for stress testing. The main drawbacks lie in the fact that targeted inputs cannot be generated, and that the tools are not aware of how much coverage is provided for the application under test. Also, random input generation can lead to redundant test sequences. Examples of random-based input generation tools are Monkey [19], Dynodroid [20], and Cadage [21].
- **Model-Based Exploration strategy**: These tools systematically generate and explore a symbolic model of the app, to investigate its behaviour. The model can be seen as a finite-state machine, where the states represent screens of the app, and the transitions represent events triggered by interactions with the GUI. All the states and transitions are typically generated dynamically while interacting with the application, and stopping when all possible routes lead to already explored states. On the one hand, the main advantage of the model-based approach is reaching full GUI coverage of the app without many redundant test sequences. On the other hand, internal behavioural changes that have no effects on the GUI may not be registered by the models. Testing tools that leverage the model-based exploration strategy to test Android apps are MobiGUItar [22], A3E [23], Swifthand [24], QUANTUM [25], and GuiRipper [26].
- **Systematic Exploration Strategy**: This category of tools refer to a family of more sophisticated techniques (such as genetic algorithms) that are used to guide the exploration towards previously uncovered code. These techniques seek to create input sequences capable of unveiling undesired behaviours. Compared to Random and Model-based strategies, these can achieve greater coverage and target more defect-prone areas of the application. The main limitation of those techniques lies in the scalability of the algorithms. Examples of testing tools adopting a systematic exploration strategy are AndroidRipper [27], CrashScope [28] and EvoDroid [29].

C. RELATED WORK

Existing research recognizes the importance of an approach similar to ours; anyway, they are not comprehensive in developing a framework involving all the aspects of GUI modelling, classification, and high-level script generation. In [22], Amalfitano *et al.* explore the state graph modelling phase with the MobiGuitar tool. They build a state-machine model where each state is a specific state of the app's screen, and each transition is a UI event. After creating the state graph breadth-first, MobiGuitar generates test cases for each path given the initial state. Then, it crosses out every possible incoming and outgoing edge for every node.

Yang *et al.* [30] have explored the application classification phase, with *Lacta*. Their approach, however, requires access to the full source code of the application. From the code, Lacta can identify the app's category. Dong *et al.* decompile the app's code and count the number of occurrences for each API function call and then classify the app through a Multinomial Naive Bayes classifier [8]. Hamedani *et al.* decompile

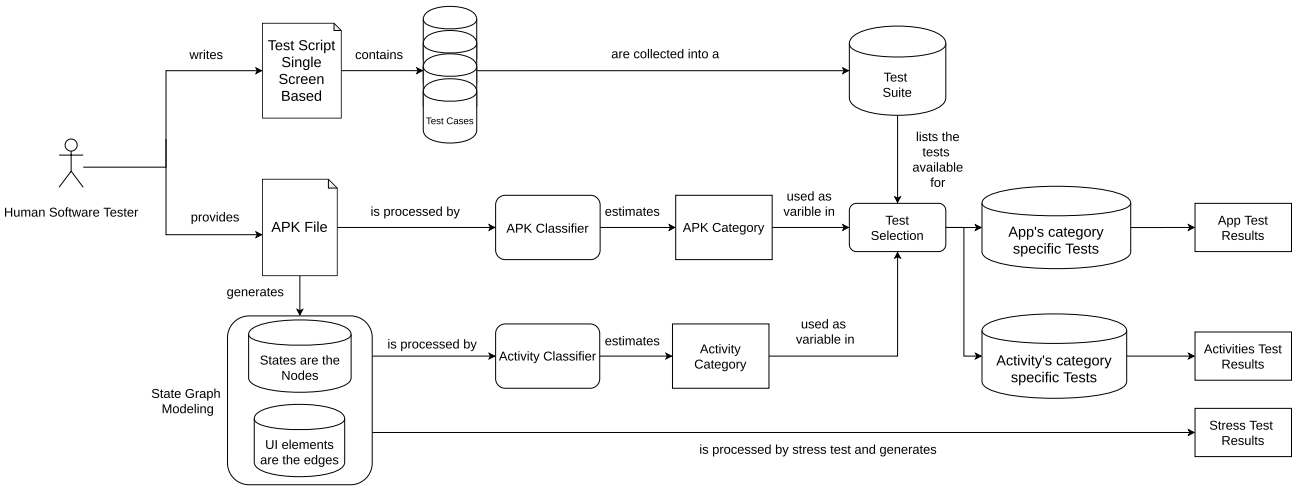


FIGURE 1. The complete testing process workflow.

the code to retrieve the list of called APIs, but they also exploit the app's intents and hard-coded strings to classify the app's category [7].

The activity classification phase has not been extensively explored yet by existing research projects. The main work towards this direction is the one by Rosenfeld *et al.*, which retrieves the list of swipable, clickable, and edited text elements to perform classification [31]. Hu *et al.* developed AppFlow that converts the entire screen layout into a single string and uses a screenshot and Optical Character Recognition (OCR) techniques to identify useful text [32].

Related to the test scripting phase, instead, a direct comparison can be made between our work and the already cited work of Hu *et al.* [32]. They allow the user to write specific test scripts where conditions with some semantic values can be imposed.

With this work, we improved accuracy on both APK and Activity classification. We developed a comprehensive approach not yet explored in literature. Finally, we created a new testing framework based on these classifiers.

III. TESTING FRAMEWORK

Figure 1 shows an overview of the testing process according to our framework. The individual modules and their implementation details will be detailed in the following section.

- **Test Script Writing.** In the first step of the testing process, the human tester can create a set of test scripts (i.e., a *test suite*). The test scripts, written in our scripting language, are not aimed at any specific mobile app but are generally applicable to broad classes of apps and activities. Therefore the tester writes scripts that will be fired when specific types of screens and apps are being tested. The test suite can contain tests for different kinds of activities, meaning that they will be executed only when a specific kind of activity is encountered.
- **APK Classification.** This module analyzes the APK file of the application under test, retrieving useful and symbolic information that can be used to classify the app into a specific category. These results are then used in the testing phase, to understand which test suites have to be executed against the AUT.
- **State Graph Modelling.** This module builds a finite state machine model of the app's GUI. This element is then used as a functional map for the execution of the test scripts. The creation of the GUI model can be considered, by itself, a form of stress test for the GUI of the app, since during the exploration of the GUI it is possible to detect crashes, bugs, freezes and other forms of defects in the application.
- **Activity Classification** The activity classifier is run at each state that identified by the State Graph Modeler, to understand the kind of the activity.
- **Test Adapter.** The information about the specific categories of APK and activity is used to filter out the tests from the test suite that are designed to be run on them. Each test case is then fired, and the test results are reported based on the assertions that are contained in the test. At the end of each test execution, the exploration of the application is backtracked to the state preceding the test execution.

IV. TOOL IMPLEMENTATION

In this section, we provide implementation details about key modules of our framework. Specifically, we detail the implementation choices that we made for state graph modelling, for both the classifiers and for the automated selection and execution of test cases.

A. STATE GRAPH MODELLING

The main purpose of the State Graph Modelling module is to build a reliable, functional, and logical map of the application. Such a map allows us to proactively learn what screen widgets

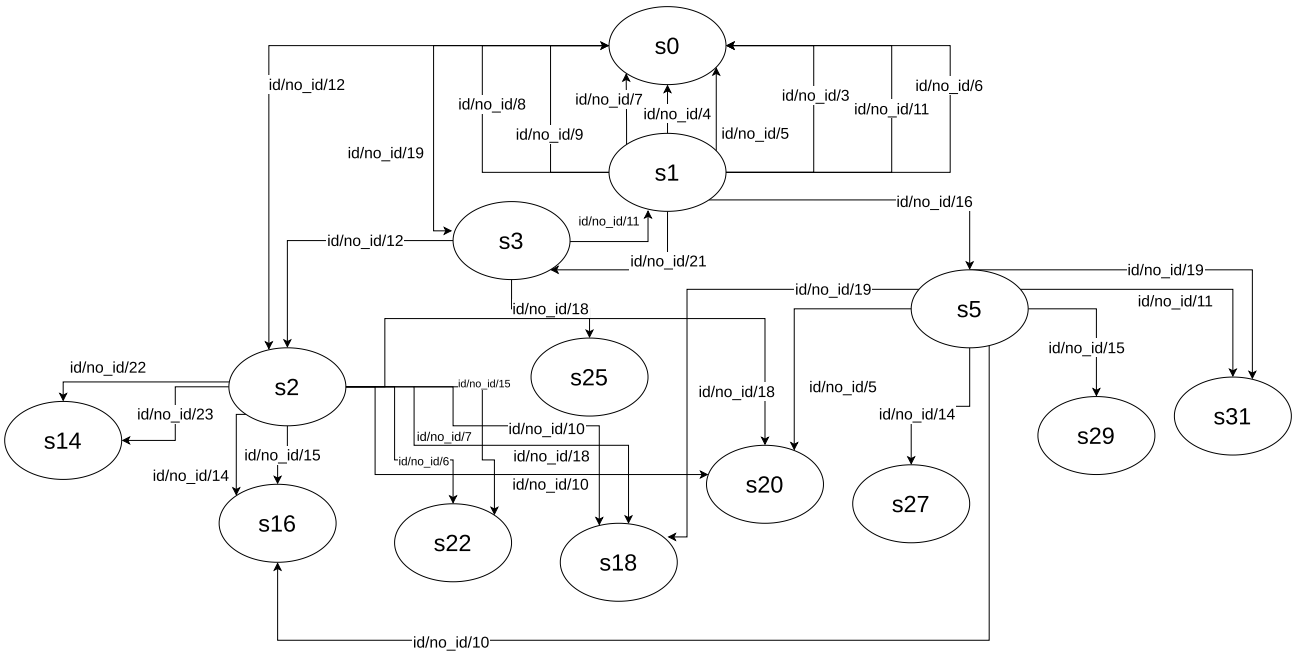


FIGURE 2. Graphical visualization of the state graph model obtained by crawling the android stock messaging app.

are available in the application and what transitions can be triggered by interacting with them.

Building automatically a state graph is equivalent to drawing a formal model, as a Finite State Machine, since the application can only be in one of a finite set of states at a given time. The state graph considers all the possible transitions out of a state, i.e., exploring all the possible UI elements in a given screen of the application. The construction of such a graph is a relevant aid for functional testing, as it provides a systematic way to explore all features of the applications.

The State Graph Model (from now on, SGM) that we adopted is a Directed Graph, with its main elements defined as follows:

- **Nodes:** Each node represents the *state* of an *activity*. By *state* of activity, we take into consideration the current values of the set of attributes associated with each UI element contained in the activity. For instance, this value could indicate whether a button is enabled, or if a toggle button is on or off. This implies that the same activity/screen can be represented multiple times, in multiple states of the app.
  From an implementation point of view, each node contains the full layout hierarchy of the activity and the values for all the available attributes. The node also contains a reference to the predecessor node, a list of all possible successor nodes, and the node's *roadmap*, that is, the list of all edges that need to be traversed to reach the node from the entry point of the graph (most typically, the Main activity shown to the user at the startup of the app). By keeping the predecessor nodes and the roadmap in a node, we can replicate the chain of interactions needed to reach the node from the startup screen of the app, hence guaranteeing replication of the created test sequences.
- **Edges:** Each edge corresponds to a specific UI element that the user can select to reach a new screen state. As every widget is represented by a multitude of attributes, we selected an id for each UI element, where the id is unique in the context of a single activity.

Figure 2 represents a sample State Graph Model, obtained by crawling the stock Android messaging app. To build the SGM, we crawled the states of the app using a FIFO queue to store the set of states that were discovered. Five essential steps are performed during the State Graph Modelling procedure:

1) **Start State Building:** The start state s_0 consists of the main activity of an application. The UI elements contained in the activity are set to their initial state as evinced by monitoring the activity with the Android Debug Bridge (ADB), which allows an app to be controlled from a desktop or laptop computer. This state corresponds to the start node of the SGM. This node is then inserted in the FIFO queue of nodes to be explored.
2) **Dequeue Phase:** The first available unexplored state is dequeued from the queue and reproduced on the Android device being monitored, starting from state s_0 (the startup screen);
3) **Exploration Phase:** Once the state to explore is reproduced, its layout structure is explored with the ADB, to save all the attribute values and to identify all possible interaction with every available UI element. All elements, both clickable and non-clickable, are interacted at this point: the reason for such operation is to provide

a form of UI testing of the app already in the phase of graph building, to investigate if defects or crashes can be reached by interacting with any element of the GUI;

4) **New-State-Check Phase:** After the interaction with a GUI element, we check whether the app is now in a new screen. If so, the new state is appended to the FIFO queue;
5) **Merging Phase:** once an iteration of the exploring phase is completed, we check whether some states inserted in the FIFO queue are equivalent to previously encountered states (i.e., all the attributes in the screen hierarchy are the same). If this is the case, only the previously created state is kept in the queue.

B. APK CLASSIFICATION

This section describes the first classification step of the proposed architecture. The APK classifier is in charge of assigning one of a set of app categories to a given APK. The approach we followed is a black-box approach where only the APK compiled package of the application is needed and not its source. It is based on Deep Neural Networks and it is similar to existing approaches in the literature, e.g., AndroClass system [7]. The main differences lie in the feature vectors and parameters that we use.

1) COMPOSING THE FEATURE VECTOR

The first decisions taken for the APK classification step was the objective features of the application that would bias toward a given app class. The following elements were taken into consideration when building our feature vectors:

- **Android API Calls:** Several methods in the literature have used Android OS API calls as features to perform classification of Android apps: LACTA [33] gathers information from the semantics of the code identifiers; AndroClass [7] and ClassifyDroid [8] refer to the full list of the possible Android OS API methods called by the app. Both approaches lead to tens of thousands of features, resulting in very sparse data. For these reasons, we are using only Android OS API calls, building a binary feature vector where each cell represents one of the possible Android OS API Classes. The value in each vector cell indicates whether the corresponding API call is present in the APK file or not.
  To find the methods (and classes) referred by the application, we extract the Java bytecodes executed by Android's Dalvik virtual machine from an APK file. The bytecodes are typically contained in one or more classes.dex files. Each of these dex files contains a *methods_ids* section, where identifiers for all API and user-defined methods are stored. Once the list of methods is obtained, it is compared with the full collection of 4339 Android API classes (as of this writing), which can be retrieved from the publicly-available Android.jar file.
- **Permissions:** Requested permissions can be very helpful in the classification of the APKs since different categories of applications need different permissions from the OS. For instance, communication applications will require *INTERNET* and *VIBRATE* permissions, which are generally not used by offline and video music players, and so on. To extract Dalvik bytecodes and the permissions required by an app, we used Apktool,[1] which returns the application's manifest file in a readable form as well as the classes.dex files. Android applications must specify the required permissions in the first part of the manifest.
  In our feature vector, we indicated with 1 a permission that was requested, and with 0 a permission that was not requested. We considered the list of 60 permissions that are provided by the official developer guide for Android.[2]
- **Hard-coded Strings as Word Vectors:** Every Android application features a file named *strings.xml* located in the */values/* subfolder of the application project. This file is referred by the application when a particular string value is needed and contains hard-coded strings that can be referred multiple times in the app and changed for localization. Each string is denoted by a unique id, using standard XML encoding. It is evident that we might have an important semantic core in the *strings.xml* file since the strings correspond to the main functionality of the app.
  The AndroClass classifier [7] collects all N words appearing in all the APKs of the training set and uses a binary vector of N elements to represent the presence of a given word in the current app instance. The main drawback of such an approach is that the diversity of the vocabulary and its size depend on the number of applications used in the dataset. We hence adopted the Word2Vec Word Embedding technique, in which each word is identified by a word vector in a vector space of 300 dimensions. We adopted the pre-trained word embedding model by *Google News Word2Vec model* as released by Mikolov *et al.*.[3] The model pairs every word string found in the *strings.xml* file with the corresponding vector of 300 values. Because of the variable number of strings, we next compute the average word vector among all the strings, obtaining a point that locates the app in a 300-dimensional space.

The resulting Feature Vector of 4699 elements is shown in Figure 3. The vector contains 4339 cells with a binary indication of the presence of references to Android API classes; 60 cells with a binary indication of whether one of the Android OS permissions has been requested or not; 300 cells with floating values that represent the average of the word vectors obtained by querying the Google News Word2Vec

[1] https://ibotpeaches.github.io/Apktool/

[2] https://developer.android.com/guide/topics/ permissions/overview #normal.permissions

[3] https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21-pQmM/edit

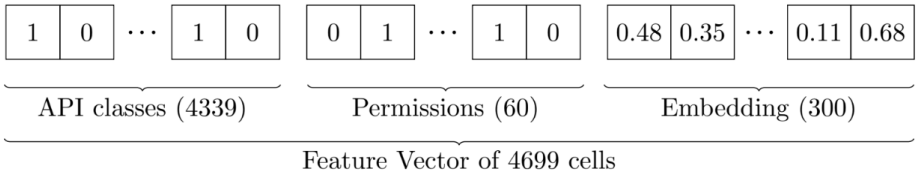


FIGURE 3. Structure of the feature vector used by the APK classifier.

model with all the words contained as hard-coded strings in the APK.

2) BUILDING THE DATASET

While training the classification algorithm, we benefited from the black-box nature of our classification approach that allowed us to select packaged apps available in the usual Android app markets.

For our evaluation, we specifically used the *APK Pure* store,[4] as it provides not only a very wide selection of application categories but also direct app download functionality without requiring a running Android device. In this manner, we built a dataset containing about 3,200 apps.

TABLE 2. Structure of the APK classification data set.

Category	Number of APKs
News and Magazines	293
Communication	342
Shopping	293
Education	295
Social	292
Food and Drink	294
Video Players	307
Medical	330
Weather	409
Music and Audio	341
Total	3,196

We selected ten different categories of applications, choosing the most popular apps in the store. We did not take into account applications belonging to the *games* category since games may feature a very diverse GUI appearance that would make it difficult to generalize a testing approach across the board. An average of more than 300 apps was downloaded for each of the selected categories. Table 2 shows the selected categories and the number of applications for each category.

3) MODEL ARCHITECTURE

In our approach, we considered a variety of statistical learning methods for our classifier. In the end, we chose a fully connected Deep Neural Networks (DNNs) because this method yielded higher accuracy results than the remaining methods that we tried. In Section V we report in detail the empirical results obtained with DNNs. We also give a summary of the results obtained with alternative methods. We have sufficient data to assure it over performs classic classifiers, however a clear understanding of the generality of this network is only possible with sample numbers in the order of millions. Our findings on DNNs are consistent with results obtained by other authors. Reyhani *et al.* [7], who also tried K-Nearest Neighbors (KNN), Naive Bayes (NB), and Support Vector Machines (SVM), also concluded that DNNs outperform the other alternatives for app classification.

TABLE 3. Selected Hyper-parameters for the APK classification architecture.

Hyper-parameter	Selection
Number of layers	2 Dense layers
Number of units per layer	1000 (1st layer), 500 (2nd layer)
Activation function	Sigmoid
Number of Epochs	90
Batch size	128
Optimizer	Adam
Learning Rate	0.00001
Dropout	0.5

We tuned the DNN's hyper-parameters using the Hand Tuning (trial and error) technique. Each parameter is tuned individually and assigned its optimal value in terms of the resulting DNN accuracy. Table 3 lists each hyper-parameter and its assigned value.

- *Number of layers*: The optimal value of this parameter depends on the number of dimensions in the training data. We chose two dense layers. A higher number of layers may lead to over-fitting.
- *Number of Units per Layer*: This parameter defines the number of cells that are used in each layer of the Neural Network.
- *Activation function*: This function defines the output of each DNN cell based on its input. Possible alternatives are ReLu, TanH, and Sigmoid (chosen).
- *Optimizer*: This is the algorithm used by the model to update the weighs of each layer, after every iteration. The most popular algorithms are Adam (chosen) and Stochastic Gradient Descent (SGD).
- *Learning Rate*: This parameter determines the speed of convergence of the DNN.
- *Batch Size*: This parameter defines the number of data points used in each training iteration before updating the weights.

[4]http://apkpure.com

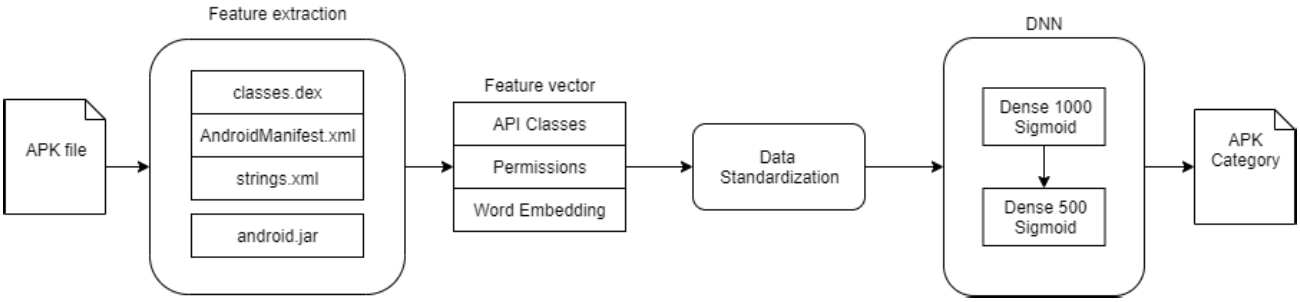


FIGURE 4. Overview of the APK classification pipeline.

```
View[ class=android.widget.Button index=1 selected=false checked=false
clickable=true package=com.example.artur.myapplication text=BUTTON long-
clickable=false enabled=true bounds=((0, 297), (1080, 441)) content-desc=
focusable=true focused=false uniqueId=id/no_id/11 checkable=false resource-
id=com.example.artur.myapplication:id/button password=false class=android.
widget.Button scrollable=false ]   parent=android.widget.RelativeLayout
```

FIGURE 5. Complete set of attributes for the dump of a UI element.

- *Number of epochs*: This parameter defines the number of times the training data is passed through the model.
- *Dropout*: This parameter is a regularization technique that prevents weight to be updated with certain probabilities, and is used to avoid over-fitting.

A final detail of the DNN implementation is related to the *Data Scaling* of the Feature Vector, which contains mixed data (i.e., both binary and continuous features). Data scaling is performed assigning to each sample a normalized value $z = (\ original\_value - mean)/\text{std}$, with *std* being the standard deviation of the distribution of samples. Performing the standardization allows us to reorganize the distribution of values in the data set so that it has a zero mean and unit standard deviation. The complete pipeline of the architecture is shown in Figure 4.

C. ACTIVITY CLASSIFICATION

This section describes the second classification step of the proposed architecture. The activity classifier is responsible for assigning a category to each activity in a given Android app. The main assumption behind this classification is the fact that many activities—even of different applications—share several common features, as they follow very specific patterns in both their structure and design. These design patterns can be exploited for functional testing purposes: for instance, many applications feature a *Settings* activity that is often structured similarly, with a so-called *ListView* element containing a set of clickable elements and toggles. This implies that those activities might require a similar approach to testing. In the remainder of this section, we report the selection of features that we used to classify activities, the classes of activities, and the structure of the classifier that we developed.

1) COMPOSING THE FEATURE VECTOR

An activity can be considered a container and a manager of UI elements, which can be either *visible* (i.e., that can be seen and interacted by the user) or *invisible* (e.g., layouts that are responsible for the arrangement and the behaviour of UI elements contained therein). The type of attributes and their number should be taken into consideration when classifying an Android activity. Also, the attributes of some UI elements have a value that can change over time. Hence the presence of those attributes can be crucial in the classification process (e.g., the *isPassword=true* attribute for an EditText element). Since the attributes of the UI elements can change at run-time, we leveraged the *dump* command of the UI Automator testing framework (called through the *AndroidViewClient* Python library[5]) to obtain the representations of the UI elements present on screen at any moment. (See Figure 5 for a dump example).

In addition to the types of widgets and their attributes, we also take into consideration the different areas where the widgets are located. As stressed by Rosenfeld *et al.* [31] many apps expose the same design patterns with widgets placed in fixed areas of the screen. We adopted their screen partitioning, splitting the device screen into three areas: (1) the top 20% of the screen, where typically the App Bar, Drawer and Options buttons are located; (2) the mid 60% of the screen, where the core of the app content is located; (3) the bottom 20% of the screen, where a Navigation Bar or a Floating Action Button is typically located. While the top and bottom areas tend to be very similar among all classes of activities, the mid-portion might differ a lot for different classes of activities. Figure 6 provides a graphical representation of the three areas on a 1920×1080 screen.

[5]https://github.com/dtmilano/AndroidViewClient

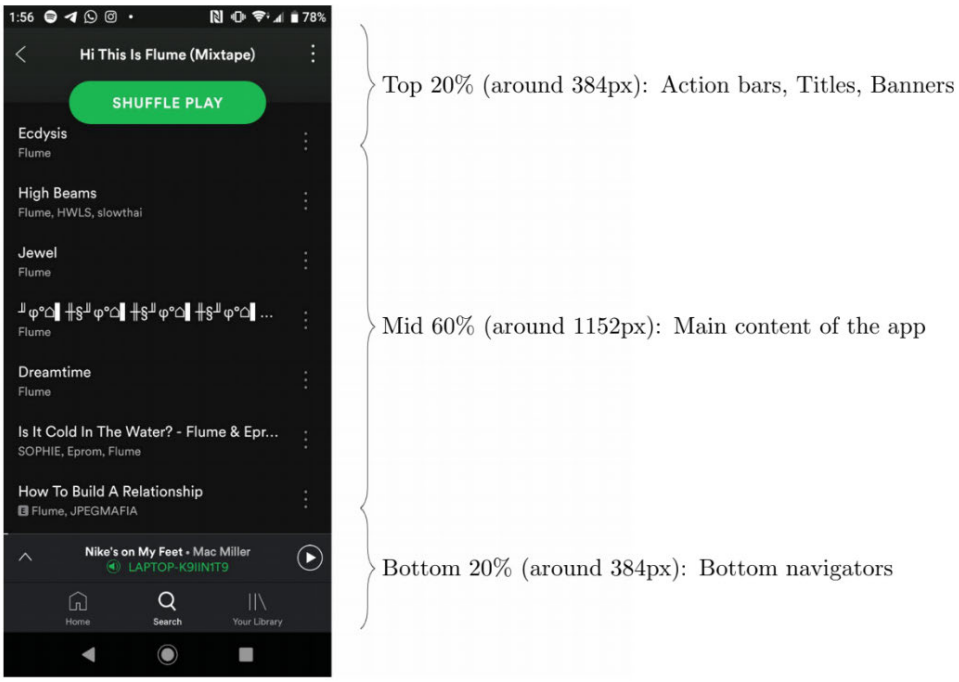


FIGURE 6. Sectioning of a 1920 × 1080 screen into the identified three areas.

TABLE 4. Set of features used for activity classification.

Feature Name	Separation in Three Areas	Number of Features
Number of Clickable elements	Yes	3
Number of Swipable elements	No	1
Number of EditText elements	Yes	3
Number of Long-Clickable elements	No	1
Number of Focusable elements	Yes	3
Number of ImageViews	No	1
Number of Password elements	No	1
Number of Checkable elements	No	1
Presence of a side Drawer	No	1
Number of UI elements on screen	No	1
Total Features		16

As the feature vector of our activity classifier, we selected an extension of the vector by Rosenfeld *et al.* [31], who relied on counters for Clickable, Swipeable, and Text field elements. In addition to those features, we also took into account the placement of the widgets on the different areas of the screen to build our feature vector, consisting of 16 integer numbers. The features that we considered are summarized in Table 4 and described in the following:

- **Number of clickable elements:** the number of elements with which the user can interact. High numbers may indicate, for instance, settings or list screens. We split this value into three counters, one for each partition of the screen.
- **Number of swipeable elements:** the number of elements that can be swiped or scrolled by the user. High numbers may indicate list or message screens.
- **Number of edit text boxes:** the number of elements where the user can input text. High numbers may indicate search bars or login prompts. We split this value into three counters, one for each partition of the screen.
- **Number of long-clickable elements:** the number of elements providing a secondary form of input. High numbers may indicate message activities or pop-up menus.
- **Number of focusable elements:** the number of elements with the *focusable* attribute (i.e., an indicator that the widget is supposed to be interacted by the user) set to true. High values may indicate activities providing the user with high interaction. We split this value into three counters, one for each partition of the screen.
- **Number of *ImageViews*:** the number of elements of the screen hosting image files. High values may suggest the presence of chat or browsing activities.

- **Number of password elements:** the number of elements where the *password* attribute (i.e., an attribute that hides input text) is set to true. A value different from zero of this counter suggests the presence of login activities.
- **Number of checkable elements:** the number of elements that can be checked with a tick-mark, mostly appearing to settings and to-do list screens.
- **Presence of a side drawer:** a binary indicator that indicates the presence of a drawer containing additional menu options.
- **Total number of UI elements on screen:** the total number of widgets in the current layout, even those that are not visible to the users.

2) BUILDING THE DATA SET

We could not find an existing data set with labelled activities that would fit our purposes. Thus, we created a new set of labelled Android activities on our own. Our set of activity categories extends the set adopted by Rosenfeld *et al.* in their related research [31]. The main objective of the definition of activity classes was to find a diverse enough set of screens. We defined the following eight classes:

- **Advertisement:** This class contains full-screen ads that must be closed by the user by selecting a specific widget.
- **Login:** This class contains activities with a login form, typically consisting of two *EditText* elements including one with the *password* attribute set to true.
- **Portal:** This kind of activity consists of an information hub used as the main section of the app, very common in news, music and audio apps.
- **List:** This class contains activities consisting of a dynamically-populated list of interactive elements, typically implemented with the *ListView* widget. If the lists are used for settings screens, they typically incorporate switches or so-called ToggleButton widgets.
- **To-Do:** This class contains activities that allow the user to create and modify a list of tasks, typically characterized by the presence of multiple checkable widgets.
- **Browser:** This class contains activities specifically constructed to perform internet browsing. These typically contain a so-called *WebView* in the central section and a search bar in the upper part.
- **Map:** This class contains activities designed to show the location of a specific point of interest, typically featuring a very complex layout due to a high number of icons, textual contents, and images that have to be shown on screen.
- **Messages:** This class contains activities characterized by a central part of the screen with the set of messages exchanged by the users, and an *EditText* used to input a new message.

Once the categories were decided, we created a handcrafted dataset of labelled samples. To do so, we crawled the ApkPure Store for 70 applications, and we traversed most of their activities. For each activity, we stored a dump of the screen, and we manually assigned the most plausible category among those just described. In this manner, we obtained 100 labelled Activities, equal to 100 labelled vectors of 16 values each, split among the eight defined activity classes. Table 5 shows the complete composition of the dataset.

TABLE 5. Composition of the activity classification data set.

Activity Category	Number of labelled samples
Advertisement	13
Login	12
Portal	13
List	13
To-Do	12
Browser	12
Map	12
Messages	13

3) MODEL ARCHITECTURE

In this work, we evaluated seven types of Machine Learning methods for our classifier. K-Nearest Neighbour, Decision Trees, Random Forest, Support Vector Machines, Naive-Bayes, Logistic Regression, and finally Convolutional Neural Networks (CNNs) have been deeply fine-tuned and their hyper-parameters evaluated with a grid search approach. In the end, we chose the logistic regression approach because of its overall higher performance in accuracy, precision, and recall that we observed empirically. The logistic regression used in multi-class classification is the multinomial one. This approach hypothesizes that all classes are independent of one another. The problem is thus solved considering the eight activity classes listed in Table 5 as eight different binary classification problems. The model outputs a real value in a range from zero to one, indicating the probability of belonging to one of the eight classes. The closer the output probability is to one, the higher the chance the sample belongs to that class.

Suppose that we have just two predictors that are linearly independent as in equation (1) where the β_i's are the parameters of the model.

$$l = log_e \frac{p}{1-p} = \beta_0 + \beta_1 * x_1 + \beta_2 * x_2 \tag{1}$$

From equation (1) we remove the logarithm obtaining the odds in equation (2).

$$\frac{p}{1-p} = e^{\beta_0+\beta_1*x_1+\beta_2*x_2} \tag{2}$$

From equation (2) we obtain the sigmoid function, which also indicates the probability of a data point belonging to a specific class as described in equation (3).

$$p_{Y=1}(X) = \frac{1}{1+e^{\beta_0+\beta_1*x_1+\beta_2*x_2}} \tag{3}$$

The model was trained to find the best set of β_i parameters feeding our feature values and obtaining as output the probability of the unknown data point belonging to that specific class. Once we obtained the whole set of probabilities

for each class, we select the highest among the eight, and we assign the activity to that class.

We also fine-tuned the *C* parameter which is the most important tuning parameter in terms of the logistic regression. This is a penalty term, meant to generalize the data in the training set. Small values mean that the regularization will be strong and the model is simpler with the chance of underfitting the data, while high values are indicative of low regularization, meaning more complex models and possible overfitting.

The effect of *C* on accuracy is depicted in Figure 7.

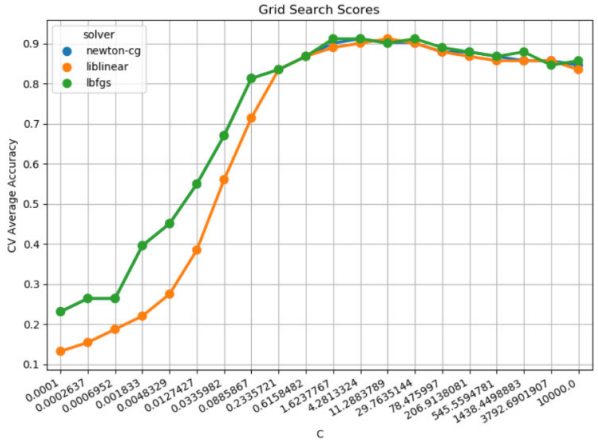


FIGURE 7. Effect of *C* normalization parameter on logistic regression.

Even if the multinomial logistic regression is the best performing in terms of accuracy, precision and recall, we analyzed the output from decision trees and random forest to determine the importance of each input feature in the activity category attribution, as shown in Figure 8.

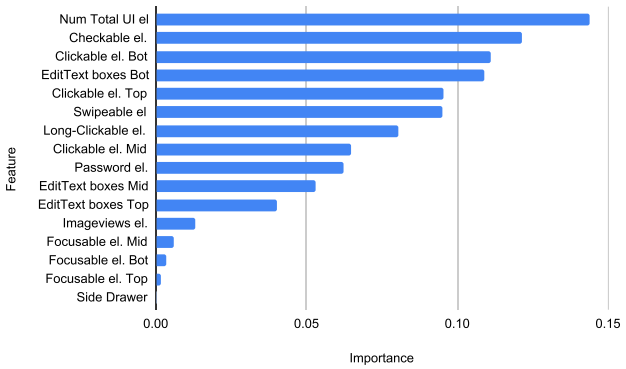


FIGURE 8. Level of importance as produced by decision trees and random forest of the features used in the classification process.

This chart presents the number of total UI elements, checkable elements, and the clickable elements at the bottom of the activity screen as the most representative features for the assignment of an activity's category.

The complete pipeline of the architecture is shown in Figure 9.

D. TEST ADAPTER

This section describes the Test Executor module that was developed based upon the two classifications described in the previous sections.

In the remainder of this section, we report the definition of our scripting language and the implementation details of the modules dedicated to test adaptation and execution.

1) TEST SCRIPTING LANGUAGE

The main drivers for the design of the test scripting language were readability and intuitiveness. Therefore we defined commands that indicate sets of gestures that can be applied to a specific category of APKs and activities.

The execution of a test case is fired when two specific *preconditions* are met:

- The AUT belongs to the specified Application category;
- The current activity belongs to the specified activity category;

The outcome of a test case (pass or fail) depends on three *post-conditions*. The test case is deemed as passing if and only if all these conditions are met:

- All the commands of the test case are executed without triggering crashes in the AUT (implicit verification);
- The condition of the assertion commands (if any) are satisfied (explicit verification);
- The last screen reached after all commands match the activity category specified by the user.

The proposed syntax allows us to declare multiple test cases inside a single test suite. A test suite is declared by using the *"When"* keyword, which indicates that the following list of test cases must be executed only in an app of a given APK category.

After the declaration of the test suite, the user specifies a list of test cases that will be executed in the same order in which they are declared. A test case is defined by a header containing two items: the category of activity in which the test case must be executed (indicated with the *"In"* keyword), and the state in which the test case must end, according to the extracted state graph of the app. If the test case must end in the same state, the *"check for SAME state"* post-condition is used. If the test case must end in a different state, the *"check for DIFFERENT state"* post-condition is used, followed by the category of the screen in which the test case must end.

The commands that compose the test scripts are divided into two families. Activity-specific commands are developed specifically for each of the eight activity categories in which each screen can be classified. The full list of activity-specific commands is detailed in Table 6. Generic commands can be used in any activity category, as they perform actions that can be usually executed in every context. The full list of generic commands is detailed in Table 7.

A sample test suite script is shown in Figure 10. The example defines a test suite for an AUT classified as a communication app. The test suite is composed of two test cases. The first test case starts in a login activity, in which the

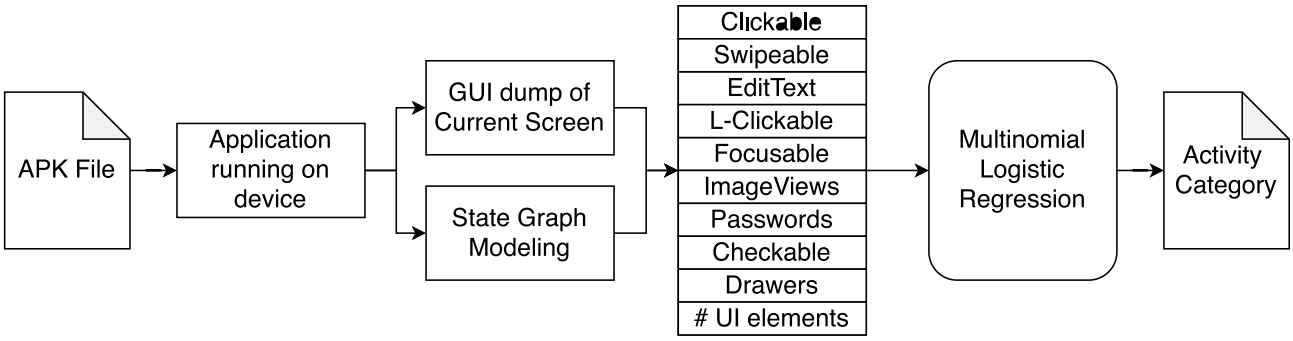


FIGURE 9. Overview of the activity Classification pipeline.

TABLE 6. Description of activity-specific commands.

Command name	Adv.	Login	Portal	List	To-Do	Browser	Map	Mess.	Description
ADD TASK "String"					x				Adds a new entry with the given title
ASSERT LINECOUNT EQUALS number					x				Checks the number of entries in the list
CLEAR FIELDS		x							Delete the text in all editable fields
CLICK AD	x								The advertisement banner is clicked
CLICK CENTER							x		Clicks the center of the activity
CLICK CLOSE	x								The advertisement is closed by the "close" button
CLICK NEXT		x							Submits login credentials
CLICK LINE number				x	x				Clicks one of the entries in a list
INPUT MESSAGE "String"								x	Writes a new message with the given string
INPUT NAME "string"		x							Inputs the string as user-name in the login form
INPUT PASSWORD "string"		x							Inputs the string as the password in the login form
INPUT SEARCH "string"							x		Writes the string as URL in the search bar
INPUT URL "string"						x			Writes the string as URL in the navigation bar
LONG CLICK ALL					x				Long-clicks all lines with spinners in a list
LONG CLICK CENTER							x		Long-clicks the center of the activity
LONG CLICK LINE number					x				Long-clicks a line with a spinner menu in a list
PRESS BACK	x			x		x			The Android "back" button is pressed
PRESS ENTER						x			Presses the enter Action key of the keyboard
SWIPE LEFT			x				x		Performs a default swipe left operation
SWIPE DOWN			x	x	x		x	x	Performs a default swipe down operation
SWIPE RIGHT			x				x		Performs a default swipe right operation
SWIPE UP			x	x	x		x	x	Performs a default swipe up operation
TICK ALL					x				Mark all entries in a to-do list
TICK LINE number					x				Marks a specific entry as completed
TOGGLE LINE number					x				Toggles the switch if present on a line of the list

TABLE 7. Description of Generic commands.

Command name	Description
CUSTOM ASSERT TEXT EQUALS "String"	Checks whether an element on screen contains the given string
CUSTOM CLICK x y	Clicks the screen at the given coordinates
CUSTOM CLICK TEXT "String"	Clicks a UI element containing the given string
CUSTOM DRAG FROM x y TO x y DURATION number	Performs a custom gesture from start to destination
CUSTOM LONG CLICK x y	Long-clicks the screen at the given coordinates
CUSTOM PRESS DEVICE BACK	Presses the Android back button
CUSTOM SLEEP number	stops the execution of the script for the given amount of ms
CUSTOM TYPE "String"	Inputs the given text in the first EditText found in the screen hierarchy

name and password are given as input, and the next button is pressed; the test case passes if the app stays in the same state. The second test case starts in a ToDo activity, where a task is added; the test case passes if the app has moved to a different layout state, in an activity of the PORTAL category.

2) IMPLEMENTATION

To implement a test runner capable of understanding the test scripting language, we have built an interpreter that creates Python code capable of performing the actual testing actions. The interpreter contains the following components.

```
1  When COMMUNICATION app:
2      In LOGIN check for SAME state:
3          INPUT NAME "arturo@gmail.com";
4          INPUT PASSWORD "abcd123";
5          CLICK NEXT;
6
7      In TODO check for DIFFERENT state PORTAL:
8          ADD TASK "title";
```

FIGURE 10. A sample test suite script containing two test cases.

- **Lexer:** Also called tokenizer, the lexer allows the conversion of sequences of characters into tokens that can be used by the following parse tree. In our implementation, we have developed the tokenizer to be case-insensitive, to recognize quoted strings as parameters of the commands, and to ignore any type of whitespace.
- **Parser:** The parser is responsible for analyzing the syntax of the language. We created a Context-Free Grammar (CFG or type-2 grammar), indicating that our production rules are in the form $S- > \gamma$, where S is always a non-terminal symbol, and γ is a string of both Terminals and Non-Terminals in any order. The complete grammar is then given as input to the ANTLR tool,[6] which is capable of producing as output a working parser written in Python, which will automatically build a parse tree. The parser that we produce is a **LL(*) Parser** [34], i.e., a type of top-down parser that parses the input provided from left to right, deriving first the leftmost non-terminal symbol. LL(*) parsers also allow us to retrieve an arbitrary number of look-ahead symbols to preventively solve conflicts in identifying the rules to follow.

[6]www.antlr.org

In Figure 11, we report as an example the Abstract Syntax Tree (AST) produced by the parsers after having encountered the sample script previously provided. (See Figure 10.)

- **Code Generator:** The purpose of the Code Generator module is to translate the logical structure of the test suite into pure Python code. We have modelled a test suite as a sequence of test cases, and each test case as a sequence of operations to be executed along with conditions for the test to succeed.
- **Widget Identification:** The final and most important step for the execution of the generated Adaptive Test Cases, is the identification of the actual UI elements on which to perform the interactions. To do so, the dump of the entire screen GUI is parsed. Four different factors are kept in consideration for the identification of the widget:
  - *Textual hints*: text content, content descriptions, and resource IDs are the primary type of information that can be used to verify the compatibility of a given widget with a desired kind of interactions; e.g., text content containing the ''next,'' ''submit'' or ''confirm'' keywords are likely compatible with the CLICK NEXT operation.
  - *Class type*: each widget class can be compatible with a limited amount of interactions, e.g., the toggle instruction is compatible with spinners but not with buttons.
  - *Attribute values*: the compatibility of a widget with a certain type of interaction can be verified by considering the values of attributes defining the action that can be performed on it; e.g., the *isPassword* attribute is used to identify the widgets were to perform the INPUT PASSWORD command.

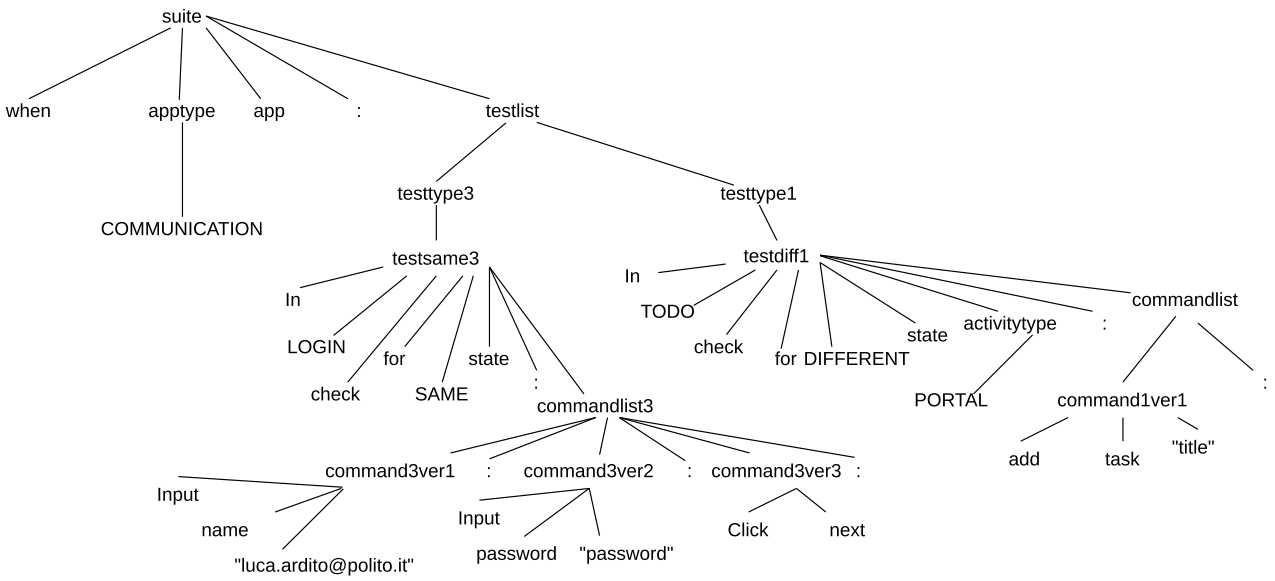


FIGURE 11. Abstract syntax tree resulting from sample input test suite.

TABLE 8. Time to find an injected bug with State Graph Model building procedure, and with the official Android Monkey pseudo-random input generator.

Application Name	Injected Bug	Time to find - SGM	Time to find - Monkey
Omni Notes[7]	Opening side drawer causes a crash	36.21s	61.60s
	Opening the 'Camera' option inside the FAB menu causes a crash	78.01s	191.2s
	Opening FAB, selecting 'New Note' and changing the Reminder Date causes a crash	173.32s	204.64s
MovieGuide[8]	Pressing the 'Like Button' in the details page of a movie causes a crash	64.11s	29.59s
	Changing the movies Sorting Order causes a crash	93.15s	47.93s
Flym[9]	Clicking on the GitHub link of the 'About' page of the app causes a crash	597.96s	747.45s
	Opening the Search Bar causes a crash	49.74s	19.15s
Timber Music Player[10]	Opening the side drawer, then settings, and clicking on a specific line causes a crash	1726.57s	2194.07s
	Selecting the 'Artists' tab causes a crash	65.38s	6.44s

  - *Locational hints*: the screen dump provides the boundaries for each element. It is hence possible to infer if a given widget is located in an area of the screen that is compatible with the desired type of interaction.

  We adopted the heuristic of selecting the widget featuring with a higher number of compatible factors. For each identified widget, we leverage its unique ID to interact with it inside the Python test cases.
- **Test Executor:** The final module of the test adapter performs the actual execution of the tests. The system starts from the State Graph Model of the activity run on the device, and at each node, it runs the activity classifier to understand the category of the current screen. Then, each test case in the test suite written for that activity category is fired.

  At the end of the execution of a test case, its outcome is evaluated, and then the application is brought back to the original state in the SGM; if other test cases are available for the current type of activity they are fired, otherwise the exploration of the app model proceeds to a new state.

V. EVALUATION

To validate our framework, we evaluated it in isolation from other components. The following sections report the results of each evaluation.

A. STATE GRAPH MODELING

As anticipated in Section III, the building process of the State Graph Model can also be deemed a form of automated testing of the correct functioning of the app's User Interface. In this case two forms of *implicit oracles* can be used to verify whether the UI elements of the app's GUI behave correctly:

- An *Unhandled Exception* (or *Crash*) stops the execution of the app;
- A *Freeze* of the app GUI, which does not respond to any input anymore.

To compare the bug-finding capabilities of the State Graph Modeler we adopted, we compared it with Monkey,[11] the official Android Random testing tool, that comes embedded with the Android Studio IDE and that feeds the GUIs of the AUT (application under test) with pseudo-random inputs.

We used the Fault Injection technique, manually modifying the code of four different open source apps by injecting nine different faults; we then (1) verified whether the bugs were spotted; (2) we compared the time to find such faults with our State Graph Modeler and with the Monkey random input generator. Details about the considered application, injected faults, and times to find the bugs are reported in Table 8. For every measurement, we started the execution from the Main Activity of the app, and we instructed the Monkey tool to always stay inside the application package (i.e., not navigating to other applications by sending intents).

We made the following observations:

- Our State Graph Modeler and the Monkey tool were both able to find all 9 injected faults.
- The times to find a fault varied significantly, especially in relation to the place where the faults were injected. When the faults were injected in the Main Activity, the random input generation of the Monkey tool was able to find the fault in less time than our framework; however, when the faults were injected in screens that were reachable after navigation in the app, the SGM building process was able to find them in less time.

We can hence assume that the time required to find a GUI fault is dependant on the size of the state graph and on the number of transitions required in the input sequence to reach the Activity where the fault is required. The ability to find all the injected faults suggests that the SGM creation itself is a reasonable means to perform a basic round of UI testing with the usage of implicit oracles, on all the widgets of the AUT.

B. APK CLASSIFICATION

We performed APK classification using a Deep Neural Network (DNN) choosing the hyper-parameters as stated in Table 9. The output layer of the net has 10 nodes, matching the number of APK categories. Category selection is performed by a Softmax activation function. The loss function is a Kullback-Leibler divergence that predicts how well the predicted data distribution approximates the real data one.

The resulting network structure is shown in Figure 12. We defined this structure by hand-tuning the hyper-parameters listed in Section IV-B3 through extensive empirical studies.

[11] https://developer.android.com/studio/test/monkey

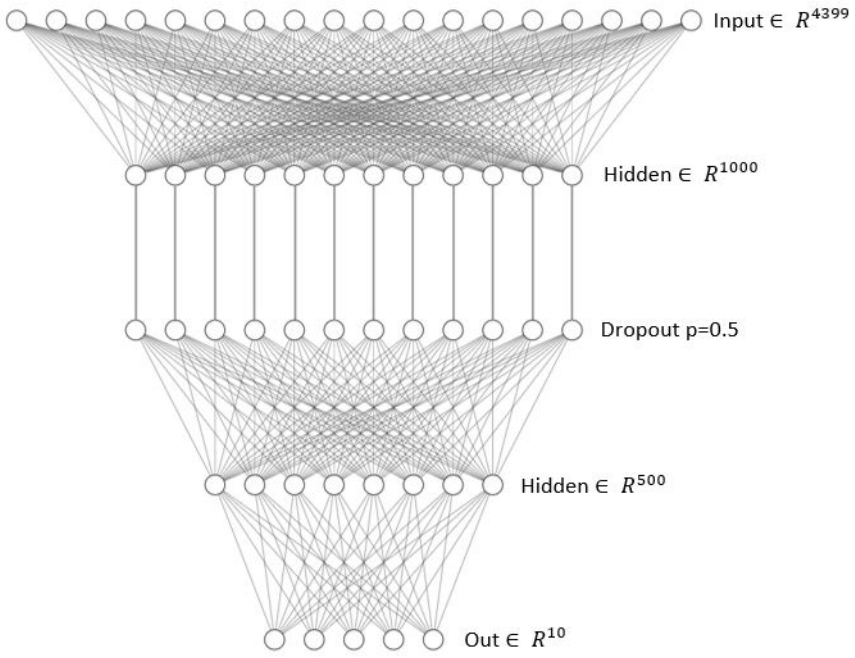


FIGURE 12. Structure of neural network for APK classification.

TABLE 9. Hyper-parameters chosen after an empirical fine tuning phase. They describe the DNN architecture and the choice made in the classification optimization.

Deep Neural Network hyper-parameters	
Parameter	Value
Dropout	0.5
Output Layer Nodes	10
Output Layer Activation Function	Softmax
Loss Function	Kullback-Leibler Divergence
Optimizer	Adam
Learning Rate	0.00001

Evaluation scores are obtained using the hold-out approach. In this work, 10% of the entire dataset is left out to be used as the final test set, while the remaining 90% of the dataset is processed with stratified 10-fold cross-validation. The results on the 90% of the dataset are exploited to fine-tune the DNN hyperparameters, while results on the test set are presented in the following.

Given the DNN above, we can reach 58.46% accuracy with a standard deviation of 2.88%. Precision is 81.36%, with a standard deviation of 9.31%. Finally, recall is 54.05%, with a standard deviation of 7.75%. We obtained these results using only the initial group of 60 permissions. When we use the full set of 158 permissions, the accuracy score is 58.18%. The accuracy so computed indicates that every single requested permission is offset by a wider feature array and this situation makes the classification process more challenging.

We assess the validity of our model comparing its performances with state of the art solutions in application classification. ClassifyDroid [8] tries to classify apps into 10 categories as well using a modified version of Multinomial Naïve Bayes, but it uses a much bigger dataset consisting of 15590 samples coming from the Chinese MM App Market.[12] However, ClassifyDroid also presents the accuracy trend for different percentages of labelled samples used, and for 3118 apps (20% of the whole dataset, perfectly comparable with our dataset of 3196 data points), it reaches an accuracy of 55%, lower than ours.

The Lacta approach [30] classifies apps into 8 categories using a dataset containing only 42 apps. The authors report only precision measures, omitting accuracy. If we use the same number of categories, we reach 88% precision, in line with their precision of 89%. However, using hand-picked apps for constructing the dataset may have artificially biased Lacta's performance. *AndroClass* [7] uses a Neural Network to classify apps retrieved from the same source (APK Pure) as ours, with about 277 apps per category, which is similar to our ratio. The authors report 48% accuracy, 45% precision, and 41% recall. Our approach clearly outperforms those results.

We have also conducted empirical studies with different feature vectors by removing some components to verify that all components help in reaching our results. Table 10 shows that removing the word vector representation of the hard-coded strings reduces performance by about 9% in terms of accuracy, while removing the permission component

[12]http://mm.10086.cn/

TABLE 10. Comparing final APK Classification results with different Feature Vectors.

Feature Vector	Accuracy	Precision	Recall
Standard (API+Perm.+WV)	$58.46\,\% \pm 2.88\,\%$	$81.36\,\% \pm 9.31\,\%$	$54.05\,\% \pm 7.75\,\%$
No WordVector	$49.29\,\% \pm 2.91\,\%$	$61.02\,\% \pm 15.11\,\%$	$45.37\,\% \pm 11.41\,\%$
No Permissions	$45.49\,\% \pm 3.95\,\%$	$54.82\,\% \pm 11.75\,\%$	$45.96\,\% \pm 10.44\,\%$

reduces the performance even further, up to 13%. These results show that none of the elements constituting our feature vector are superfluous; all components have a semantic meaning that is a key to the success of the classification process.

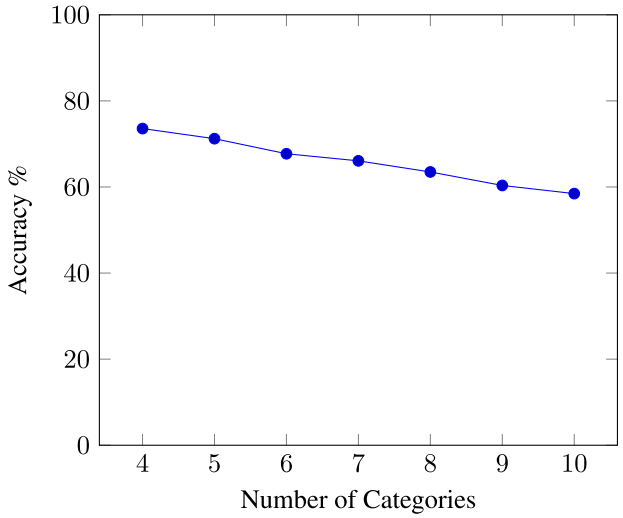


FIGURE 13. Effect of number of APK classes on model accuracy.

Finally, we checked the number of APK categories and their effects on the variation on the accuracy of the model. Figure 13 represents the effect of the number of APK classes on model accuracy. On the left side, the accuracy is 73.57% with 4 categories. Next, the accuracy linearly decreases with the number of categories. If we restrict the number of categories to 5, we obtain an accuracy score of 71.22%. In our experiment, our model classifies APK classes according to 10 categories. We conclude that, even if the accuracy decreases with the number of categories, the incremental loss is modest.

In conclusion, our fully connected Deep Neural Network architecture yields satisfactory results in the APK classification problem. Evidently, the success of classification depends on the chosen dataset for any given classifier structure. For instance, AndroClass [7] reaches an accuracy of 48% using the APK Pure[13] dataset (same source as ours), 57% using the Google dataset[14] and even 85% using a hand-made dataset. These results imply that the choice of training applications and their assignment to APK classes (i.e., the "ground truth") affects the performance of the classifier.

[13]https://apkpure.com/
[14]https://play.google.com/

On the one hand, misclassifying ambiguous applications in the training dataset misleads the entire training process. On the other hand, using APKs that belong to just one class improves the process. Ideally, we would have a manually-labelled dataset that has a reduced yet disjoint set of categories. AndroClass is clear proof of this fact; by "cherry-picking" apps in the training set, their accuracy improved by 37%. All of this highlights how the manual categorization of Android applications performed on some app stores can lead to ambiguities. In our case, due to time and resource constraints, it was possible to just crawl the APK Pure repository, but it would be interesting to see what happens with a dataset consisting of thousands of manually-labelled applications.

C. ACTIVITY CLASSIFICATION

Logistic regression is the best performing machine learning algorithm in activity classification among the algorithms we tested, as shown in Table 12. It has the highest scores in accuracy, precision, and recall. Assuming independence among activity classes, we exploit multinomial logistic regression to perform 8 different binary classifications. Activity selection is performed by choosing the highest positive probability among the 8 classifications. The eight categories are listed in Table 5, while the features exploited to classify them are listed in Table 4. The architecture behind the multinomial logistic regression is depicted in Figure 14.

The evaluation score is obtained with a 10-fold cross-validation. We also compute scores with the "leave one out" approach for a clearer comparison with other research projects in this field. The most notable measurements are the 91.01% accuracy using 10-fold cross validation. It is important to notice the high scores in precision 96.54% and recall 95.25%. The precision score highlights the ability of our model to discriminate among the 8 classes of activities. In addition, the high recall scoring suggests that most of the relevant samples are detected, meaning that the number of false negatives is negligible.

Other than multinomial logistic regression, we tried three Convolutional Neural Networks (CNN) to classify activities and to compare their score with the existing baseline created by previous approaches. The first CNN uses as input 1920×1080 pixel screenshots of activities. Some screenshot pixels are cropped to remove the bottom Android OS Action Bar, and the top OS Status Bar, which are both graphical elements identical in every Activity; therefore, it is no use to consider those areas in the classification process. The screenshot is then converted to grey-scale to reduce the dimension of the image thanks to the removal of the colour channels. After a

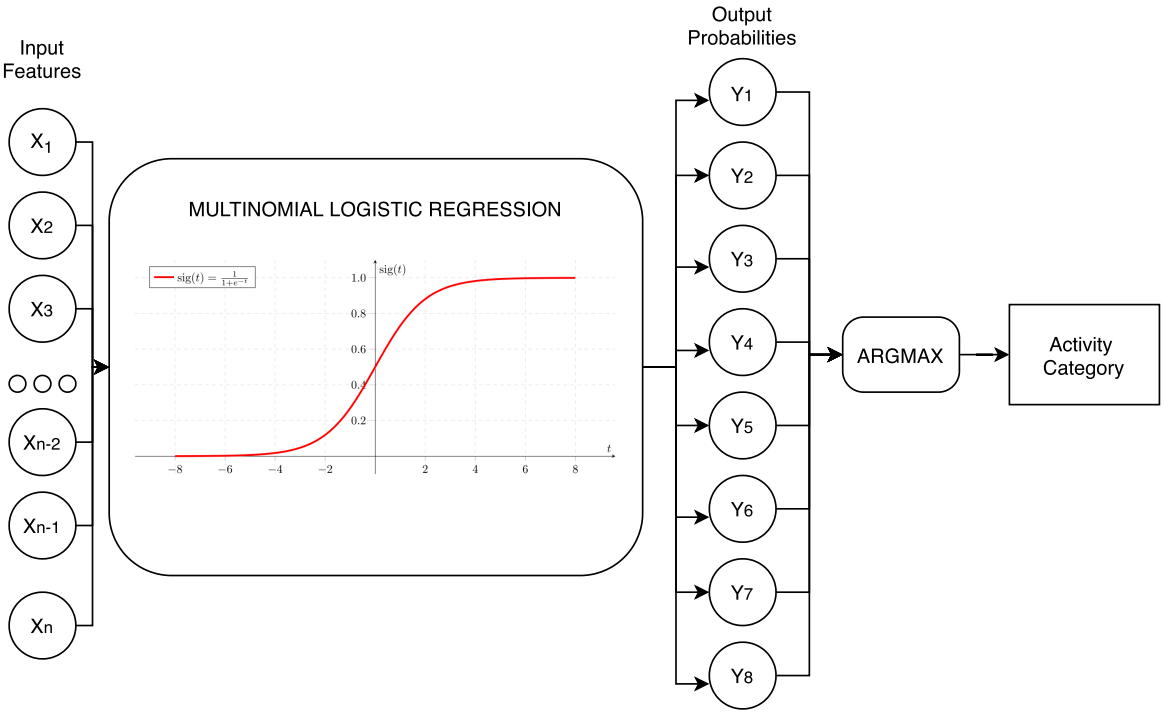


FIGURE 14. Our multinomial logistic regression exploits 16 input features listed in Table 4, then performs 8 different binary logistic regression and finally selects the category with the highest positive probability and assigns the input activity to that category.

first pooling operation, the size of the input is (778, 486, 1). At this stage, the image is fed to the network. The architecture is made of a Convolutional layer (kernel 11 × 11, stride 4), Max Pooling (kernel 3 × 3, stride 2), Convolutional (kernel 5 × 5, stride 1), Max Pooling (kernel 3 × 3, stride 2), 3 Convolutional Layers (kernel 3 × 3, stride 1), Max Pooling (kernel 3 × 3, stride 2), and finally a Dense Relu Layer of 100 units. The output is given as input to the third network concatenation layer.

The second network tested has one Relu (Rectifier Linear Unit) layer that sends the 16 inputs of the Features Vector to the next concatenation layer. The third network joins the first two models: their tensors are concatenated into a single input. One dense layer has been added (Relu Dense of 50 units) at the end. The output layer has a Softmax activation function that outputs the conditional probability of the 8 classes. Categorical cross-entropy has been chosen as the loss function, together with a stochastic gradient descent optimizer with a learning rate of 0.001.

Unfortunately, that model never converges during the training phase, remaining stuck at around 16% accuracy scores after 100 epochs. We have about 988,000 parameters in our classifier, too many variables with few data points (100 labelled samples). Therefore, using Activity's screenshots to aid us in the classification process did not have any beneficial effects.

The reference with other research projects confirms that our DNN classifier outperforms the existing baseline in this field. Rosenfeld *et al.* classify Android activities into 7 categories using a 10-fold validation procedure [31]. They use 80 training activities versus our 100 samples. They report accuracy scores with different approaches, so we compare our work with their best accuracy score of 86.25% using a KStar algorithm. We outperform this model with an increment in accuracy of 4.76%. A second comparison is with the work of Hu *et al.* [32]. In their approach, they classify Android Activities retrieving text with an OCR (Optical Character Recognition) starting from activity screenshots. We compare our results with best result that they obtained with leave-one-out cross-validation. They classify activities in two categories reaching a top accuracy of 87.3%. Even with the leave one out approach and classifying a greater number of categories, our approach reaches an accuracy of 89%, improving their result by 1.7%.

TABLE 11. Per-class evaluation in terms of precision and recall.

Activity Class	Precision	Recall
Advertisement	83.88 %	92.46 %
Login	96.82 %	94.03 %
Portal	93.08 %	80.49 %
List	75.37 %	73.33 %
To-Do	76.97 %	87.20 %
Browser	89.62 %	83.25 %
Map	81.54 %	71.53 %
Messages	95.94 %	95.04 %

A deeper analysis is presented by results in Table 11. These scores are obtained with a 10-fold cross-validation with the logistic regression algorithm, retrieving a confusion matrix at each iteration, and calculating precision and recall

TABLE 12. Comparison of a machine learning algorithm for activity classification. In the first three rows we show our top-performing algorithms, in the last two rows we compare with other research projects results.

	10-Fold			Leave One Out		
	Accuracy	Precision	Recall	Accuracy	Precision	Recall
Logistic Regression	91.01 %	96.54 %	95.25 %	89.00 %	89.00 %	89.00 %
Random Forest	87.98 %	95.67 %	92.26 %	87.00 %	87.00 %	87.00 %
SVM - Linear Kernel	88.08 %	95.40 %	94.57 %	84.00 %	84.00 %	84.00 %
Rosenfeld et al. [31]	82.5 %	-	-	-	-	-
AppFlow [32]	-	-	-	87.3 %	-	-

at each round, then averaging the whole scores at the end of the process. The top scores are yielded by Login, Messages, and Portal activities, showing that they maintain their characteristics switching between different Android applications. This means that they appear similar in terms of widgets and UI elements. Moreover, most widgets are located in the same screen areas across different apps. In conclusion, we outperform existing baselines with an accuracy score of 91.01%, but we also obtained high scores in precision 96.54% and recall 95.25%, thereby setting a new state of the art baseline.

D. TEST ADAPTATION

To evaluate the testing Adaptation phase—and, in general, the testing process with our tool as a whole—we focused on three different research questions:

- **RQ1:** Can widgets with a specific semantic value be found correctly by using textual hints and layout attributes?
- **RQ2:** How often do the generated test cases lead to the expected (pass or fail) result?
- **RQ3:** How much does the approach help developers/testers in saving labour for the creation of test scripts?

The motivation behind the first research question is to understand whether textual hints and layout attribute values are sufficient to identify widgets with the same semantic value but located in different contexts and applications. To answer RQ1, we applied some commands of the scripting framework to a diverse set of applications, to evaluate their versatility. We tested the most complex commands supported by the scripting module, in terms of lines of Python code required for their implementation. In Table 13 we report the results of the application of the (sequences of) commands to different AUTs, the outcome (success or failure of the command execution) and, in case of failure, its cause.

In total, we have defined 4 different sets of commands to be applied to 32 different applications.

The `ADD TASK "NoteTitle"` command led to 6 out of 8 correct executions. The implementation of the command searches for a wide set of textual hints to find the button that can be used to add a new note (e.g., *new, create, write*); for the *Cute ToDo List* AUT, the test failed because no textual hints, content descriptions or ids were added in the layout for the confirmation button. Hence the operation could not be performed. The second failure, for the *ToDo* AUT, was due to the way the addition of a task was implemented. In two cases, the commands proved to be versatile even in unexpected situations: for the *ToDo List* AUT, an unpredictable pop-up of an advertisement could happen after entering the title of the note, forcing the testing engine to press the confirm button twice; for the *Checklist* AUT, the *Add New Note* button was uncommonly placed inside a drawer menu.

The second set of experiments involved a full login sequence, which was executed in 7 out of 8 cases; the only failure happened when the *Polito App* was selected as AUT. However, this failure was due to the missing translation to the English language of one of the applications. Future implementations of the testing framework may overcome this issue by taking into consideration multiple translations of the

TABLE 13. Results of the evaluation of the commands versatility.

Command	Application	Outcome
ADD TASK "NoteTitle";	Todo List Easy	Success
	ToDo List	Success
	Tasks	Success
	Cute ToDo List	Failure
	Reminder	Success
	ToDo	Failure
	inception List	Success
	Checklist	Success
INPUT NAME "testemail@gmail.com"; INPUT PASSWORD "password"; CLICK NEXT;	Turo	Success
	KAYAK	Success
	Polito App	Failure
	TEDx App	Success
	TripAdvisor App	Success
	Just Eat	Success
	Postmates	Success
	GRUBHUB	Success
CLICK LINE 3;	YouTube	Success
	Google Maps	Success
	Google Chrome	Success
	Android Settings	Success
	Expedia App	Failure
	BBC News App	Success
	Booking.com App	Success
	LinkedIn App	Success
INPUT MESSAGE "Test String"	Stock Android Messaging App	Success
	Message Classic	Success
	Pulse Messaging	Success
	QKSMS	Success
	Apple Message	Success
	Messages GO	Success
	Messages	Success
	Messages App	Success

TABLE 14. Quality evaluation of the functional testing procedure.

Starting Activity	Script	App Name	Test Explanation	Expected Outcome	Actual Outcome
ToDo	Script 1	ToDo List (OS)	Adding a single note and then removing it	Fail	Fail
	Script 2	ToDo List (OS)	Same as script 1; a bug preventing the insertion of a new note was injected	Fail	Fail
	Script 3	ToDo List (OS)	In a list with 2 notes, add a new one, and check that 3 notes are displayed	Pass	Pass
Ad	Script 4	Ads Toolbox	Open an advertisement, thenk check it	Pass	Pass
Login	Script 5	Patreon App	Login without correct credentials	Pass	Pass
	Script 6	Patreon App	Login with wrong credentials	Pass	Pass
	Script 7	Patreon App	Login with correct credentials	Pass	Pass
List Selection	Script 8	Android Stock Messaging App	Send a message to the most recent contact	Pass	Pass
	Script 9	Amaze File Manager (OS)	Create a new folder; a bug making all names lowercase was injected	Fail	Fail
Portal	Script 10	Expedia App	Try to visit all the tabs of the application	Pass	Pass
	Script 11	Expedia App	Change app country	Pass	Pass
Browser	Script 12	DuckDuckGo	Check whether the current website is loaded after inputting the URL	Pass	Fail
Map	Script 13	Google Maps	Search for a specific city, and then enter Street View	Pass	Pass
Messages	Script 14	QKSMS	Check if the string content of a message is displayed correctly after the message is sent; a bug changing random characters is injected in the app.	Fail	Fail

keywords used to identify the widgets to execute the desired operations.

Similar results were obtained for the `CLICK LINE 3` command, which was executed correctly in all but one of the List activities; the one failure that happened for the *Expedia App* AUT was due to the use of a custom layout instead of a typical *ListView* structure. In the case of the *Google Chrome* AUT, the test produced a slightly different behaviour than expected since the first element of the *ListView* in the app is a Login button, thereby incrementing the line indexes by 1.

Finally, the command `INPUT MESSAGE "TestString"` led to 8 out of 8 correct executions, in Message activities. The command also worked for the *Messages* AUT, which presented an unusual structure of the Message Composition screen, featuring multiple *EditText* boxes instead of a single one; clearly, only a single one was filled with information by the `INPUT MESSAGE` command.

Overall, 28 out of 32 executions (87.5%) of the commands were correctly translated to actual inputs on the activities of the applications under test, proving very high versatility of the abstract commands to the actual implementation of Android apps. The tests did not require access to the source code or the compiled code of the applications under test and involved applications with very complex user interfaces, thus corroborating the applicability of the approach to real-world Android applications.

To answer RQ2, we verified the capability of the test cases to provide the correct positive or negative results when applied to Android applications (i.e., whether a test fails if it is supposed to fail, and passes if it is supposed to pass).

We selected for that purpose 8 different apps.[15] On top of those 8 apps, we defined 14 testing scenarios that were built to test the core functionalities of the Activity Category to which they belong. To test different possible situations in test executions, we tailored some tests to generate a failure by inserting erroneous explicit assertions; also, we used the Fault Injection technique again, by accessing the source code of some of the apps to inject simple, functional bugs that should lead correct test cases to fail. The full scripts are reported in Appendix A of this manuscript. Table 14 reports the details of all the scripts that we developed, the applications that were used, the explanation of each test case (i.e., its operations), the expected and actual outcomes of the test execution, and the explanation of the test outcome. By comparing the expected outcome with the actual one, it is possible to evaluate the capability of the testing framework to provide the correct test outcomes.

The scripts correctly led to failures when wrong assertions about the final screen were formulated, or when bugs were

[15]The number of AUTs is compatible with the one that emerges in a literature review of Android automated testing approaches by Kong *et al.* [35]

injected in the AUT. Test scripts 2 and 9 were supposed to fail because bugs were injected in the AUTs (*ToDo List* and *Amaze File Manager*); they indeed led to failure, respectively, because of the inability to execute a tap command on a note that was not correctly saved, and because of a mismatch in the text given in input by the user and that shown in the GUI. Test Script 1 was supposed to fail because a wrong assertion was added at the end of the script, checking that a different state of the application is reached while the application is supposed to remain in the same state after the addition of note.

In a single case out of 14 (script 12 for the *DuckDuckGo* AUT), the outcome of the execution of our script did not match the expected one. The behaviour was due to an application that presented a very uncommon behaviour in deleting the user's input after the keyboard is closed.

To summarize, the combination of the Activity Classifier and the script creation framework proved to be a versatile tool, able – in our evaluation phase – to generate correctly functioning test script in more than 90% cases.

As a preliminary answer to RQ3, we report the code saving that can be obtained by adopting our high-level commands instead of utilizing common layout-based scripting languages. Each of the adaptive commands we defined embeds 27.1 lines of Python code using the UIAutomator library (with a maximum value of 89 lines for the ADD TASK command for a *ToDo* activity).

This measure suggests that the proposed approach and the current set of adaptive commands can be considered an excellent starting point to achieve higher simplicity in creating test cases, and a significant reduction of the labour needed by testers in the creation of the test scripts.

VI. DISCUSSION

The evaluation we performed yielded positive results for all the individual models of our proposed framework. All the evaluation steps involved real-world published Android applications (either open-source or not). Hence, it can be considered a preliminary demonstration of the applicability of the framework in existing projects and industrial contexts.

The complete framework showcases bug-finding features already in the exploration phase, where the State Graph Model of the application is constructed (with a fault-finding capacity and a fault-finding time that was comparable to that of state of the random art testers). The high-level commands that we defined in the framework were applicable in near 90% of the cases we tested, and the scripts based upon them led to more than 90% correct test executions (in terms of correct pass or fail outcome).

The main limitations of the proposed approach depend on the design choices of the individual modules, which may affect the generality of the results.

Regarding the model creation phase, we observe that the main limitation of the State Graph Modeler we adopted is the slow-building process, mainly impacted by the time necessary to capture the dumps of every traversed screen. In some cases, we observed that wrong layout structures were yielded by the UI Automator tool in a non-deterministic way. Possible ways to increase the precision and the speed of the approach can be investigated, starting with evaluating other tools and/or libraries to retrieve layout structures and dumps.

With respect to the components of our frameworks performing APK and Activity classification, we have used application and activity datasets already available and validated in the literature. Of course, we cannot guarantee that any app and activity in the universe can be filed under the categories that we used. Another limitation to our generality can be represented by the use of the 10-fold cross-validation in the Activity classification phase. This limitation can be overcome by extending the size of the adopted dataset.

Regarding the test script generation module of the framework, the present study has not investigated the coverage provided by the commands on the use of cases exposed by the tested applications. Extensive empirical experiments are needed to verify the efficiency in finding real bugs in Android apps.

VII. CONCLUSION AND FUTURE WORK

We have described a machine-learning-based framework for creating high-level test scripts for Android applications. Test scripts are defined based on a target application and activity category; scripts can be launched adaptively regardless of the specific implementation details of the individual AUT. We propose a syntax that allows the creation of test suites and the definition of simple assertions based on the current state and type of the Android Activity.

Our framework complements the existing literature by providing a higher accuracy with respect to state-of-the-art APK and activity classifiers and defining a high-level mean of writing implementation-agnostic Android test cases. In that aspect, it allows developing low-weight test scripts that are generalizable to multiple activities and multiple applications of the same type.

We regard the generality mentioned above of test scripts as a significant added value for a testing framework, given the labour-intensive nature of the development of test scripts for mobile apps and the typically high maintenance they need during the normal evolution of the AUTs. High-level activity-related commands, detached from the specific layout properties, can alleviate the tedious and error-prone backtracking of the AUT use cases to the layout properties of individual widgets while ensuring higher readability of test code.

The automated association performed at run-time by the framework between high-level commands and specific layout properties (i.e., content descriptions, ids, and textual content of the widgets) can also move the definition of the test cases to the preliminary steps of the development process. Test cases can be defined during the definition of AUT requirements before the activity layouts are defined, facilitating the application of test-driven development practices in the Android domain.

As our immediate future work, in addition to addressing the limitations discussed in the previous section, we plan to set up additional empirical studies to evaluate the whole framework, measuring its capability to adapt entire test suites over large sets of similar applications.

Regarding the classifiers, we plan to evaluate other feature vectors, datasets of applications and activities, and other architectures to improve the performance of the classifiers.

Finally, we envision the design and execution of empirical experiments to evaluate the Testing Framework's capacity to generate test cases able to provide high coverage of the GUI widgets and high fault-finding capacity. Also, it is worth investigating the labour that testers can save using our Scripting Approach instead of traditional script-based testing.

As a further improvement of our framework, we plan to develop a semantic-aware conversational agent able to interact with the output of this working framework and with human test developers. The agents will look for flaws in the test procedure and will suggest corrections. This kind of approach will require much more data, especially labeled data. Therefore, we will build a dataset following standards in APK and activity description to expand the amount of available data for a community based shared dataset used as a baseline for future developments.

APPENDIX
SCRIPTS FOR TESTING FRAMEWORK EVALUATION

- Script 1:

```
In TODO check for DIFFERENT state TODO:
    ADD TASK "Title";
    CUSTOM SLEEP 2000;
    TICK LINE 1;

In TODO check for DIFFERENT state PORTAL:
    ADD TASK "title";
```

- Script 2:

```
In TODO check for SAME state:
    ADD TASK "Title";
    CUSTOM SLEEP 2000;
    TICK LINE 1;
```

- Script 3:

```
In TODO check for DIFFERENT state TODO:
    ADD TASK "Title";
    CUSTOM SLEEP 2000;
    ASSERT LINECOUNT EQUALS 3;
```

- Script 4:

```
In AD check for DIFFERENT state AD:
    CLICK AD;
    PRESS BACK;
    CUSTOM SLEEP 3000;
    CLICK CLOSE;
```

- Script 5:

```
In LOGIN check for SAME state:
    CLICK NEXT;
```

- Script 6:

```
In LOGIN check for SAME state:
    INPUT PASSWORD "wrongPassword";
    INPUT NAME "correctEmail@gmail.com";
    CLICK NEXT;
    CLEAR FIELDS;
```

- Script 7:

```
In LOGIN check for DIFFERENT state PORTAL:
    INPUT PASSWORD "correctPassword";
    INPUT NAME "correctEmail@gmail.com";
    CLICK NEXT;
```

- Script 8:

```
In LIST check for DIFFERENT state MESSAGES:
    CLICK LINE 0;
    CUSTOM SLEEP 1000;
    CUSTOM CLICK TEXT "Text message";
    CUSTOM SLEEP 1000;
    CUSTOM TYPE "HelloWorld";
    CUSTOM SLEEP 1000;
    CUSTOM CLICK TEXT "send";
    CUSTOM SLEEP 1000;
    CUSTOM ASSERT TEXT EQUALS "HelloWorld";
```

- Script 9:

```
In LIST check for DIFFERENT state LIST:
    CUSTOM CLICK TEXT "Download";
    CUSTOM SLEEP 2000;
    CUSTOM CLICK 940 1640;
    CUSTOM SLEEP 1000;
    CUSTOM CLICK TEXT "Folder";
    CUSTOM SLEEP 1000;
    CUSTOM CLICK TEXT "Enter Name";
    CUSTOM SLEEP 1000;
    CUSTOM TYPE "ThisIsAName";
    CUSTOM SLEEP 2000;
    CUSTOM CLICK TEXT "CREATE";
    CUSTOM SLEEP 1000;
    CUSTOM ASSERT TEXT EQUALS "ThisIsAName";
```

- Script 10:

```
In PORTAL check for SAME state:
    SWIPE RIGHT;
    CUSTOM SLEEP 1000;
    SWIPE RIGHT;
    CUSTOM SLEEP 1000;
    SWIPE LEFT;
    CUSTOM SLEEP 1000;
    SWIPE LEFT;
```

- Script 11:

```
In PORTAL check for DIFFERENT state PORTAL:
    SWIPE RIGHT;
    CUSTOM SLEEP 1000;
    SWIPE RIGHT;
    CUSTOM SLEEP 2000;
    CUSTOM CLICK TEXT "Country";
    CUSTOM SLEEP 3000;
    CUSTOM CLICK TEXT "Canada";
```

- Script 12:

```
In BROWSER check for SAME state:
    INPUT URL "bbc.co.uk";
    PRESS ENTER;
    CUSTOM ASSERT TEXT EQUALS "bbc";
```

- Script 13:

```
In MAP check for DIFFERENT state MAP:
    INPUT SEARCH "San Francisco";
    CUSTOM SLEEP 1000;
    CUSTOM LONG CLICK 226 1220;
    CUSTOM SLEEP 1000;
    SWIPE UP;
    SWIPE DOWN;
    SWIPE LEFT;
    SWIPE RIGHT;
    CUSTOM PRESS DEVICE BACK;
```

- Script 14:

```
In MESSAGES check for DIFFERENT state MESSAGES:
    INPUT MESSAGE "A sentence";
    INPUT MESSAGE "Another sentence";
    PRESS ENTER;
    CUSTOM ASSERT TEXT EQUALS "sentence";
```

ACKNOWLEDGMENT

A. Cardone partially conducted this work in the context of his M.Sc. Thesis [36]. The authors would like to thank Arturo for his work.

REFERENCES

[1] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, and X. Li, "ATOM: Automatic maintenance of GUI test scripts for evolving mobile applications," in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation (ICST)*, Mar. 2017, pp. 161–171.

[2] N. Chang, L. Wang, Y. Pei, S. K. Mondal, and X. Li, "Change-based test script maintenance for Android apps," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Jul. 2018, pp. 215–225.

[3] R. Coppola, M. Morisio, M. Torchiano, and L. Ardito, "Scripted GUI testing of Android open-source apps: Evolution of test code and fragility causes," *Empirical Softw. Eng.*, vol. 24, pp. 3205–3248, May 2019.

[4] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *Proc. IEEE 8th Int. Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2015, pp. 1–10.

[5] M. Linares-Vasquez, K. Moran, and D. Poshyvanyk, "Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2017, pp. 399–410.

[6] D. Rimawi and S. Zein, "A model based approach for Android design patterns detection," in *Proc. 3rd Int. Symp. Multidisciplinary Stud. Innov. Technol. (ISMSIT)*, Oct. 2019, pp. 1–10.

[7] M. R. Hamedani, D. Shin, M. Lee, S.-J. Cho, and C. Hwang, "AndroClass: An effective method to classify Android applications by applying deep neural networks to comprehensive features," *Wireless Commun. Mobile Comput.*, vol. 2018, pp. 1–21, Sep. 2018.

[8] F. Dong, Y. Guo, C. Li, G. Xu, and F. Wei, "ClassifyDroid: Large scale Android applications classification using semi-supervised multinomial Naive Bayes," in *Proc. 4th Int. Conf. Cloud Comput. Intell. Syst. (CCIS)*, Aug. 2016, pp. 77–81.

[9] S. Singh, R. Gadgil, and A. Chudgor, "Automated testing of mobile applications using scripting technique: A study on Appium," *Int. J. Current Eng. Technol.*, vol. 4, no. 5, pp. 3627–3630, 2014.

[10] S. Negara, N. Esfahani, and R. Buse, "Practical Android test recording with espresso test recorder," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., Softw. Eng. Pract. (ICSE-SEIP)*. Piscataway, NJ, USA: IEEE Press, May 2019, pp. 193–202.

[11] *Espresso*. Accessed: Jan. 14, 2020. [Online]. Available: https://developer.android.com/training/testing/espresso

[12] *Uiautomator*. Accessed: Jan. 14, 2020. [Online]. Available: https://developer.android.com/training/testing/ui-automator

[13] M. K. Kulkarni and A. Soumya, "Deployment of calabash automation framework to analyze the performance of an Android application," *J. Res.*, vol. 2, no. 3, pp. 1–6, 2016.

[14] A. Jain, M. Jain, and S. Dhankar, "A comparison of ranorex and qtp automated testing tools and their impact on software testing," *IJEMS*, vol. 1, no. 1, pp. 8–12, 2014.

[15] B. Sadeh, K. Ørbekk, M. M. Eide, N. C. Gjerde, T. A. Tønnesland, and S. Gopalakrishnan, "Towards unit testing of user interface code for Android mobile applications," in *Proc. Int. Conf. Softw. Eng. Comput. Syst.* Berlin, Germany: Springer, 2011, pp. 163–175.

[16] H. Zadgaonkar, *Robotium Automated Testing for Android*. Birmingham, U.K.: Packt Publishing Ltd, 2013.

[17] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "RERAN: Timing- and touch-sensitive record and replay for Android," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*. Piscataway, NJ, USA: IEEE Press, May 2013, pp. 72–81.

[18] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso, "Barista: A technique for recording, encoding, and running platform independent Android tests," in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation (ICST)*, Mar. 2017, pp. 149–160.

[19] *Ui Application Exerciser Monkey*. Accessed: Jan. 14, 2020. [Online]. Available: https://developer.android.com/studio/-test/monkey

[20] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *Proc. 9th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, 2013, pp. 224–234.

[21] H. Zhu, X. Ye, X. Zhang, and K. Shen, "A context-aware approach for dynamic GUI testing of Android applications," in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf.*, vol. 2, Jul. 2015, pp. 248–253.

[22] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated model-based testing of mobile apps," *IEEE Softw.*, vol. 32, no. 5, pp. 53–59, Sep. 2015.

[23] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Lang. Appl. (OOPSLA)*, vol. 48, 2013, pp. 641–660.

[24] W. Choi, G. Necula, and K. Sen, "Guided GUI testing of Android apps with minimal restart and approximate learning," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Lang. Appl. (OOPSLA)*, vol. 48, 2013, pp. 623–640.

[25] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of mobile apps," in *Proc. IEEE 7th Int. Conf. Softw. Test., Verification Validation*, Mar. 2014, pp. 183–192.

[26] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and G. Imparato, "A toolset for GUI testing of Android applications," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance (ICSM)*, Sep. 2012, pp. 650–653.

[27] D. Amalfitano, A. R. Fasolino, P. Tramontana, and N. Amatucci, "Considering context events in event-based testing of mobile applications," in *Proc. IEEE 6th Int. Conf. Softw. Test., Verification Validation Workshops*, Mar. 2013, pp. 126–133.

[28] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk, "CrashScope: A practical tool for automated testing of Android applications," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion (ICSE-C)*, May 2017, pp. 15–18.

[29] R. Mahmood, N. Mirzaei, and S. Malek, "EvoDroid: Segmented evolutionary testing of Android apps," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2014, pp. 599–609.

[30] C.-Z. Yang and M.-H. Tu, "Lacta: An enhanced automatic software categorization on the native code of Android applications," in *Proc. Int. Multiconf. Eng. Comput. Scientists*, in Lecture Notes in Engineering and Computer Science, vol. 2195. Hong Kong: Newswood Limited, Mar. 2012, pp. 769–773.

[31] A. Rosenfeld, O. Kardashov, and O. Zang, "Automation of Android applications testing using machine learning activities classification," 2017, *arXiv:1709.00928*. [Online]. Available: http://arxiv.org/abs/1709.00928

[32] G. Hu, L. Zhu, and J. Yang, "AppFlow: Using machine learning to synthesize robust, reusable UI tests," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*. New York, NY, USA: Association Computing Machinery, 2018, p. 269.

[33] C.-Z. Yang and M.-H. Tu, "Lacta: An enhanced automatic software categorization on the native code of Android applications," in *Proc. Int. Multiconf. Eng. Comput. Scientists (IMECS)*, vol. 1, 2012, pp. 1–5.

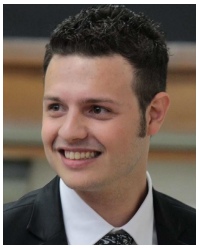
[34] T. Parr and K. Fisher, "LL (*) the foundation of the ANTLR parser generator," *ACM Sigplan Notices*, vol. 46, no. 6, pp. 425–436, 2011.

[35] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyande, and J. Klein, "Automated testing of Android apps: A systematic literature review," *IEEE Trans. Rel.*, vol. 68, no. 1, pp. 45–66, Mar. 2019.

[36] A. Cardone, "Machine learning methods for adaptive test case generation for Android activities," M.S. thesis, Dept. Comput. Sci., Univ. Illinois Chicago, Chicago, IL, USA, 2019.

LUCA ARDITO (Member, IEEE) received the B.Sc., M.Sc., and Ph.D. degrees in computer engineering from the Politecnico di Torino. He is currently an Assistant Professor with the Department of Control and Computer Engineering, Politecnico di Torino, where he is also working with the Software Engineering research group. His current research interests include mobile development and testing, green software, new programming language analysis, and empirical software engineering methodologies.

RICCARDO COPPOLA (Member, IEEE) received the M.Sc. degree in computer engineering and the Ph.D. degree in control and computer engineering from the Politecnico di Torino, Turin, Italy, where he is currently working as a Research Assistant with time contract. His research interests include automated GUI testing for web and mobile applications, and the evaluation of non-functional properties of software projects.



SIMONE LEONARDI received the B.Sc. and M.Sc. degrees in computer engineering from the Poltecnico di Torino, where he is currently pursuing the Ph.D. degree in computer and control engineering. He is also a member with the Software Engineering Group. His current research interests include natural language processing and personality analysis applied at social media analysis and software testing.

MAURIZIO MORISIO received the M.Sc. degree in electronic engineering and the Ph.D. degree in software engineering from the Polytechnic of Turin. In 1998 and 2000, he spent two years at the University of Maryland at College Park, working with the Experimental Software Engineering Group led by V. Basili. From September 1998 to June 2000, he was on the Board of Director of the Software Engineering Laboratory (SEL). He is currently a Full Professor with the Department of Control and Computer Engineering, Polytechnic of Turin, where he leads the software engineering group. He has published more than 130 articles in international journals and conferences, and three books.

UGO BUY is currently an Associate Professor with the University of Illinois at Chicago. His research interests include general area of software engineering with emphasis on modeling and analysis of concurrent and real-time systems, software development for multicore hardware, mobile app development, and sensor networks. In the past, he investigated various methods for automatic verification of these systems using such models as finite automata and Petri nets. In recent years, he has shifted my main research focus to the automatic generation of control supervisors for discrete manufacturing systems. More generally, he is also interested in techniques for supervisory control and dynamic reconfiguration of discrete event systems.