Polytechnic University of Turin
Department of Computer and Information Science

Master's Thesis

PoLiUToDroid: A Non-Invasive Automatic Black-Box UI Testing Technique for Android Mobile Applications based on a Novel Active Learning Approach

by

Vincenzo Junior Forte

2017-11-08

Polytechnic University of Turin
Corso Duca degli Abruzzi, 24 - 10129 Torino

Linköpings universitet
SE-581 83 Linköping, Sweden

Polytechnic University of Turin
Department of Computer and Information Science

Master's Thesis

PoLiUToDroid: A Non-Invasive Automatic Black-Box UI Testing Technique for Android Mobile Applications based on a Novel Active Learning Approach

by

Vincenzo Junior Forte

2017-11-08

Supervisor: Ulf Kargén, Giovanni Malnati
Examinator: Nahid Shahmehri

Abstract

Mobile devices have become an integral part of daily life and their applications are becoming increasingly necessary for people. The rapid growth and the great importance of Android in the world involves a continuous research to ensure that these apps are of high quality and safe for the users. As Android mobile applications (both malicious and benign) become more and more complex, efficient and effective techniques and tools are essential to assure the development, maintenance and vetting of secure and high-quality apps.

The thesis introduces a non-invasive automatic black-box testing technique for Android mobile applications able to interact with an Android app on most devices without modifying the operating system or instrument the app, so that it can be used also in particular scenarios such as malware analysis. Our research focuses on a model-learning technique based on a GUI Ripping approach, able to examine and test the application while inferring a model of the application under test, considering the need to operate as transparently as possible, and taking into account possible interactions across multiple apps.

Acknowledgments

I would first like to thank my thesis supervisor Ulf Kargén of the Division for Database and Information Techniques (ADIT) in the Department of Computer and Information Science at Linköping University. His office door was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right the direction whenever he thought I needed it.

I would also like to acknowledge Professor Nahid Shahmehri, Head of the ADIT, with whom I am gratefully indebted to have accepted me for this thesis.

I would also like to thank the Professor Giovanni Malnati of the Department of Control and Computer Engineering at Polytechnic University of Turin as the second reader of this thesis, and I am grateful to him for his availability.

Finally, I must express my very profound gratitude to my parents, to my brothers, to my relatives and to my friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Thank you

Contents

List of Figures

List of Tables

1 Introduction

The last few years have been revolutionary for mobile devices. The increasing number of mobile users in the global market has led to a huge rise of mobile applications (or simply, "apps") that consumers use on their smartphones. According to the market research report from Statista [25, 26], the Apple App Store boasts close to 2.2 million of these apps while Google Play remained the first-largest app store with 2.8 million available apps.

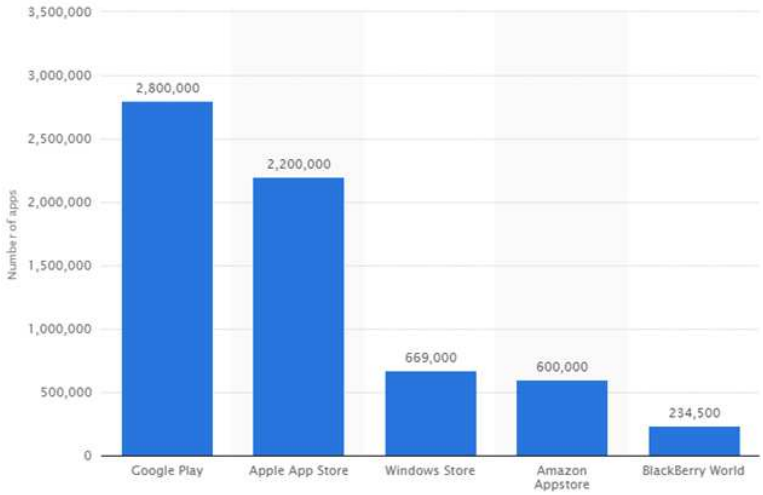


Figure 1.1: Number of apps available for download in leading app stores as of March 2017 [25]

In 2016, consumers downloaded 149.3 billion mobile apps to their connected devices. In 2021, the following figure is projected to grow to 352.9 billion app downloads. However, 2016 data establish that many downloaded apps are not used more than once in the first six months.

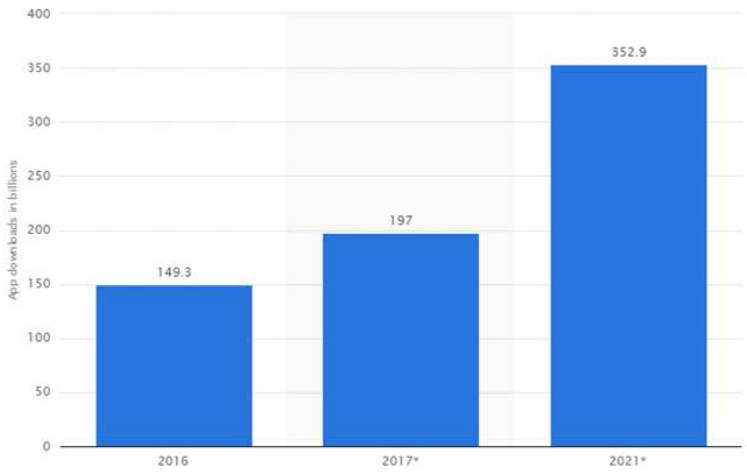


Figure 1.2: Forecast for the number of mobile app downloads worldwide in 2016, 2017 and 2021 [26]

The incredible growth of the mobile business has reinforced the need for greater software quality, in terms of reliability, usability, performance and security. To cope with this growing demand for high quality applications, developers need to pay close attention to the software development processes: the use of well-defined software engineering techniques becomes indispensable and mobile application testing and analysis play a strategic role in ensuring the success of an app.

1.1 Motivation

Software testing is generally one of the most critical and expensive activities in the software lifecycle, but for mobile applications, it may be an even more complex activity due to the specific features and problems that characterize these applications, such as compatibility issues due to platform, operating system (OS) and device fragmentation, network diversity and mobile connectivity troubles, location-dependence and limited processing capability [51, 64]. For this reason, automated testing has become an important topic of research in this area. In particular, a large amount of research has focused on automatic input generation techniques for Android applications [1, 30, 31, 32, 34, 37, 43, 47, 48, 49, 50, 63], for multiple reasons. Firstly, Android has the largest share of the mobile market, which makes this platform extremely appealing to industry practitioners. Secondly, due to fragmentation of devices and operating system releases, Android applications often suffer from cross-platform and cross-version incompatibilities, making manual testing of these apps particularly expensive and, therefore, worthwhile to automate. Thirdly, the open source nature of the Android platform and its technologies makes it a more suitable target for academic researchers who can have full access to the underlying applications and operating system.

On the other hand, the huge amount of sensitive data held by mobile applications scares the world of users as much as the developer world, because malicious apps could thus acquire or use such information without user consent. We have seen how mobile applications have had a widespread adoption in the recent years, but at the same time users of Android devices are also subject to a fast-growing quantity of malware. Based on a report from security specialists F-Secure [41], there are over 19 million malware programs developed especially for Android, of which 4 million new ones only in the year 2016, making Google's mobile operating system the main target for mobile malware. For this reason, it has become essential to analyze mobile applications as fully as possible in order to detect any unwanted behavior.

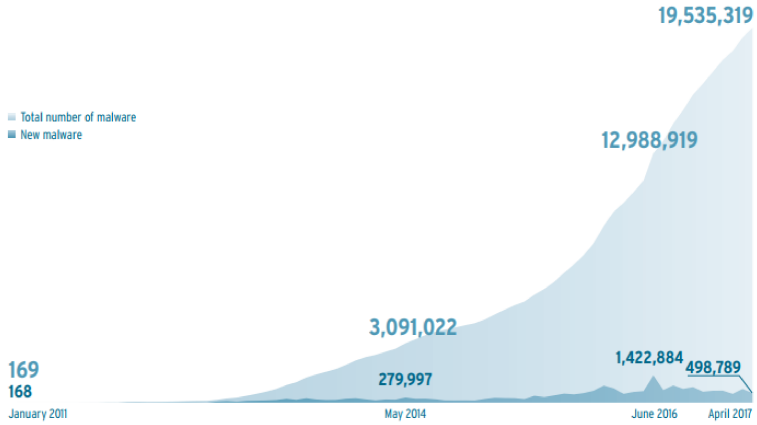


Figure 1.3: Number of Malicious Android Apps in AV-TEST's Database [59]

Table 1.1 provides a brief overview of the features of existing Android testing techniques and tools. A first significant difference between these techniques can be made on the basis of the relative timing between test cases generation and their execution (see also ***Section*** 2.2). These steps can in fact be separated or interlaced. In this regard, we say that test execution can be done either *off-line* or *on-line* from test generation [60]. In off-line testing, test cases are generated strictly before being run, while in on-line testing, test cases generation and execution are performed simultaneously.

Many off-line testing techniques proposed in the context of Android applications exploit an existing model of the application under test (AUT), others utilize static and dynamic analysis approaches to infer the model which is then used to generate test cases.
A static analysis inspects app code (or bytecode) inside the app package (APK) without executing any code. Although it could potentially reveal possible defects, all static methods, however, are vulnerable to obfuscations (e.g., encryption) that remove, or limit, access to the code.
In contrast to static analysis, dynamic analysis executes an application and observes the results. It involves working with the software, giving input values and checking if the output is as expected. Typically, a dynamic test checks for functional behavior, memory/CPU usage, response time, and overall performance of the system.
Another group of off-line testing techniques use search-based approaches to generate test cases, such as EvoDroid [49], which automatically retrieves models from the source code of the target app. Lastly, other techniques presented in the literature do not exploit a model-based approach, but use information obtained from static analysis of the app under test.

On-line testing techniques launch the AUT and then proceed with iteratively sending events to it. There are two main categories in this field, depending on the approach used to define event sequences. A first group includes techniques that implement random event selection from a set of possible ones. A second group of techniques, known as **model-learning** techniques, proceed more systematically, utilizing an *active learning* approach [37] in order to infer a model of the AUT during testing, used to make decisions about future actions to be performed.

Table 1.1: Overview of existing tools and techniques for automated Android app testing [38, 47]

Tool	Instrumentation		Approach	Testing strategy	Emulator	Real device
	Platform	App				
A^{3}E-Depth-first [34]	✗	✓	Model-based	Black-box	✗	✓
A^{3}E-Targeted [34]	✗	✓	Model-based	Grey-box	✗	✓
ACTeve [32]	✓	✓	Program analysis	White-box	✓	✗
AndroidRipper [31]	✗	✓	Model-based	Black-box	✓	✗
DroidBot [47]	✗	✗	Model-based	Black-box	✓	✓
Dynodroid [48]	✓	✗	Random	Black-box	✓	✗
EvoDroid [49]	✗	✓	Search-based	White-box	✓	✗
Monkey [1]	✗	✗	Random	Black-box	✓	✓
ORBIT [63]	?	?	Model-based	Grey-box	✓	✗
PUMA [43]	✗	✓	Model-based	Black-box	✓	✓
Sapienz [50]	✗	✓	Search-based	Grey-box	✓	✓
SwiftHand [37]	✗	✓	Model-based	Black-box	✓	✓

In this work, we focus on fully automatic on-line techniques, able to both automatically generate test cases and to execute them following an active learning strategy. A first noteworthy technique, implemented in the *AndroidRipper* tool, is based on the concept of **GUI Ripping**, a reverse engineering process that aims to build a GUI model of an existing software application by dynamically interacting with its user interface. It fires predefined events [28, 31] allowing exploration of the GUI (Graphical User Interface) according to a systematic and well-defined GUI traversal strategies (such as breadth-first, depth-first) [29]. For this purpose, it rebuilds and maintains a *GUI Tree* model of the graphical user interface. This type of traversal is interrupted when all the distinct GUI states of the app are considered visited. The authors propose two different heuristics to determine when a GUI state can be considered equivalent to an already explored one. The first heuristic considers two equivalent GUI states if they belong to the same *Activity*. Activities are one of the main components of Android apps and can be simplistically seen as screens that an application can present to its users. This is not always true because multiple screens are likely to belong to the same Activity (see ***Section*** 2.1.2). For this reason, the second heuristic is more restrictive: Two GUI states are considered equivalent only if they include the same set of GUI components (*widgets*).
Two other systematic testing techniques have been implemented in the *A^{3}E* tool [34]. One of these requires a preliminary static analysis of the bytecode in order to infer a model of the GUI. The second technique automatically explores the Activities and the GUI components in a depth-first manner. The technique is able to infer a model, consisting of nodes representing the app Activities and edges representing actual transitions between Activities.
A further active learning testing technique, implemented in the *SwiftHand* tool [37], exploits execution traces generated during the testing process to learn an approximate model of the GUI. The model is used to choose inputs that may lead the application to unexplored states; when new inputs are started and new screens are displayed, the model is refined.
Finally, a grey-box active learning approach, implemented in the *Orbit* tool [63], allows to extract the set of user actions supported by each widget in the GUI performing a preliminary static analysis of the AUT source code.

Collectively, these techniques cover several important testing objectives but, even without getting into the details, it is possible to highlight two fundamental aspects that motivate the writing of this thesis:

1. A superficial and nowadays often inadequate comparison criterion is frequently used to determine the equality between two GUI states: An Activity-level GUI model is in fact too abstract to represent dynamically constructed GUIs in recent Android apps (e.g., because they use dynamic Fragments [2]);

2. The exploration of a mobile application often relies on invasive or, in some cases, unreliable techniques: According to a recent survey [38], most testing approaches use app instrumentation or system modification to get enough information to drive the tests. However, while effective in some cases, these techniques are less suited in some scenarios such as compatibility testing or malware analysis. Since many malicious applications are obfuscated, it may be difficult, or sometimes impossible, to use this practice. In addition, some malware apply sandbox detection or use other advanced anti-analysis techniques that could allow them to hide their malicious behavior, or tamper with the environment, if they identify that they are operating under "non-standard" conditions [58].

1.2 Aim

The aim of the thesis is to design a model-learning automatic black-box UI testing technique, able to infer a model and generate input without relying on bytecode instrumentation or other invasive (and therefore easily detectable) techniques, in order to work also on obfuscated or malicious apps. Achieving this goal will thus provide a good starting point for further studies focusing on malware analysis and detection.

1.3 Research Questions

The aim of this work is to find transparent, effective and efficient ways to retrieve structural information about the GUI, for exploration and testing of Android mobile applications. To this end, the following questions are investigated:

RQ1. Which GUI exploration techniques are most suitable for testing/analyzing closed-source, obfuscated and/or malicious apps?

   a) What could be suitable precautions in order to try to avoid that malicious applications detect the activity of a testing/analysis tool?

RQ2. Which could be an effective GUI abstraction of GUI states?

   a) How can GUI-related information be retrieved from closed-source apps?
   b) What GUI information is most useful for characterizing the GUI state?
   c) Taking into account the need to operate on devices with limited resources, how can GUI states be represented in an efficient way?

RQ3. Which could be an effective comparison criterion to distinguish GUI states?

   a) What information needs to be considered to evaluate the equality between two GUI states?
   b) How shall apps that start other apps be handled?
   c) How can GUI state comparisons be implemented in an efficient and scalable way?

RQ1 is the main question to be asked to begin to evaluate options on how it is possible to analyze applications when the source code is unavailable (as is often in the real world) or unreachable through reverse engineering without risking inconsistencies between the tested and the original version. At the same time, during the discussion of the subject, some analysis behaviors which might be detected by malicious applications will be highlighted.

RQ2 concerns model building achieved by using the GUI Ripping technique. In this case, the two main aspects to be considered will be *how* to retrieve information without relying on the internal implementation details of the target app and *what* should be the most useful

information to characterize a GUI state. In this regard, another important aspect is to find an efficient way to represent the GUI state, considering that we want to be able to work directly on real devices to prevent malware from detecting an emulated environment.

Lastly, for *RQ3*, we seek to develop a criterion on which to base the exploration and, therefore, to generate the GUI model. Since test cases are generated based on the underlying model, accurate (but efficient) GUI modeling of an application under test is a crucial factor in order to generate effective test inputs. To address this problem, a GUI model abstraction must be defined, which also takes into account the possible interactions across multiple apps (when the user flow crosses into other apps or into the system UI). In this case, both the need to work on devices with limited resources and the real possibility to operate on applications mapped into models with a large number of GUI states must be considered.

2 Background

2.1 Android

Android is a platform targeted mainly towards mobile devices. Back in 2007, Google 's Andy Rubin [24] described it as follows:

> *"Android is the first truly open and comprehensive platform for mobile devices. It includes an operating system, user-interface and applications – all of the software to run a mobile phone, but without the proprietary obstacles that have hindered mobile innovation."*

It is currently developed by Google Inc. and launched on the market for the first time in 2008. Since then, the success of the Android platform has grown steadily, becoming the world's most popular mobile operating system in 2011. Unlike major competing systems such as iOS (Apple) and Windows Phone (Microsoft), Android rests on an open-source framework, based on the consolidated Linux kernel.

This section is based on the information available on the Android official web pages [3, 4] and describes the main aspects and the structure of Android.

2.1.1 Platform architecture

Android is an open source, Linux-based layered software stack created for an increasingly wide range of hardware and devices. The following figure illustrates the high-level architecture of the Android platform.

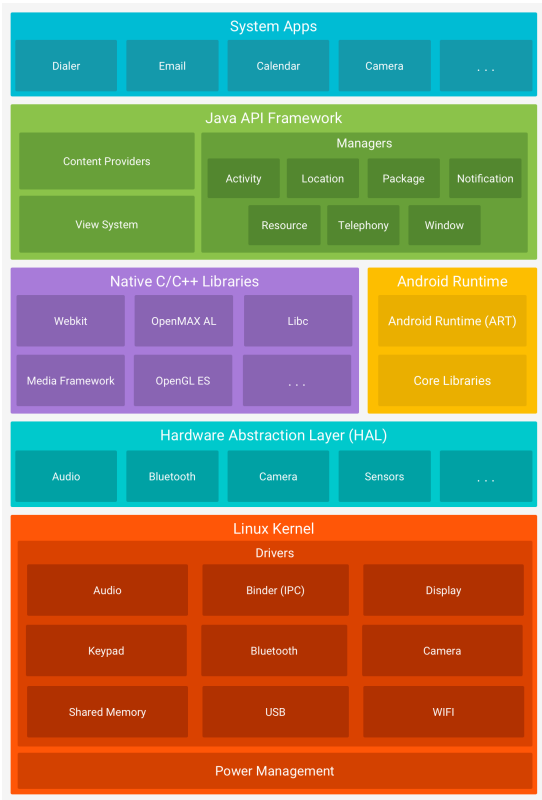


Figure 2.1: The Android software stack [5]

The Linux Kernel

The foundation of the Android is a version of the Linux kernel with a few special and important additions for a mobile embedded platform. The use of a Linux kernel allows Android to inherit the Linux operating system's capabilities, to take advantage of the major security features, and enables device manufactures to develop hardware drivers for a well-known kernel.

Hardware Abstraction Layer

The hardware abstraction layer (HAL) provides standard interfaces that expose device hardware capabilities to the higher-level Java API (Application Programming Interface) framework, thus allowing Android to be agnostic about lower-level driver implementations. The HAL consists of multiple library modules typically developed using native technology (C/C++ and shared libraries), each of which implements an interface for a specific type of hardware component. When a framework API makes a call to access device hardware, the Android system loads the library module for that hardware component.

Android Runtime

Android Runtime or, shortened, ART is the runtime that was introduced experimentally in Android 4.4, and later with Android 5.0 became officially used by the Android applications. Prior to Android 5.0, Android applications ran on the top of the Dalvik Virtual Machine (VM).

The idea behind Dalvik's JIT (just-in-time) execution was to profile the applications while they were being executed and dynamically compile the most used segments of the bytecode into native machine code. Differently, ART re-introduces the concept of AOT (ahead-of-time) compilation. It compiles the whole application code into the native machine code, without

interpreting bytecode at all. This takes some time, but it is done only once during the installation of the application. This enables the application code to later be executed directly by the device's runtime environment.
Other important improvements of ART include an optimized garbage collection (GC) and an improved debugging support, including a dedicated sampling profiler, detailed diagnostic exceptions and crash reporting, and the ability to set watchpoints to monitor specific fields.

Although ART is now the official runtime for the Android platform starting from Android 5.0, Android needs to ensure compatibility with all those apps that are already on the market. All those devices that are running an older version of the Android platform rely on the Dalvik VM. Therefore, for backward compatibility reasons, Android application packages are still prepared based on Dalvik specifications. As it was optimized for mobile environments, the Dalvik VM understands only a special type of bytecode designed specially for Android, known as Dalvik Executable (DEX), which provides lots of advantages compared to standard Java bytecode. The Android SDK (Software Development Kit) comes with tools that can translate standard Java bytecode into DEX bytecode during the packaging of the Android application. ART does an automatic conversion from Dalvik's DEX format into ART's OAT format on-the-fly as soon as an application is installed on the device.

Android also includes a set of core runtime libraries that provide most of the functionality of the Java programming language, including some Java 8 language features, that the Java API framework uses.

Native C/C++ Libraries

Many core Android system components and services, such as ART and HAL, are built from native code that require native libraries written in C and C++. The Android platform provides Java framework APIs to expose the functionality of some of these native libraries to apps. Using Android NDK (Native Development Kit) gives the developers the opportunity to access some of these native platform libraries calling native C, and partially C++, code directly from an Android Java application.

Java API Framework

The entire feature-set of the Android OS is available through APIs written in the Java language. The application framework provides a plethora of managers that allow to Android applications to interact with the Android platform and the device, which include the following:

- A rich and extensible *View System* that allows to build an app's user interface
- A *Resource Manager*, providing access to non-code resources contained in the apps
- A *Notification Manager* that enables all apps to display custom alerts in the status bar
- An *Activity Manager* that provides a common navigation back stack and manages the Android Activity lifecycle
- *Content Providers* that provides a common approach to share data between different apps

System Apps

Android comes with a set of core apps. Apps included with the platform have no special status among the apps the user chooses to install, although software provided by manufacturers typically uses a read-only memory area to be sure that these applications will always be installed. The system apps function both as apps for users and to provide key capabilities that developers can access from their own app.

2.1.2 Application components

The Android framework provides a set of building blocks to enable the development of consistent and interoperable mobile apps. There are four main components of an Android application: *Activities*, *Services*, *Broadcast Receivers*, and *Content Providers*. These components can contact each other using *Intents* which is Android's mechanism for inter-process communication. Application components, the manifest file (see ***Section 2.1.3***), and application resources are packaged in an application package *.apk* file format, which is the only format that Android system recognizes for packages that need to be installed on the Android device.

Activity

Activities are one of the fundamental building blocks of apps on the Android platform. They serve as the entry point for a user's interaction with an app. Almost all activities require interaction with the user, and for that reason, the activity takes care of providing the window in which the app draws its UI. This window is typically full screen, but may be smaller than the screen and float on top of other windows. Each activity can contain a set of *Views* and even *Fragments* presenting information, and allows users to interact with the application. Fragments were introduced in Android 3.0 to address the issue of different screen sizes and represent behaviors or portions of user interface in an activity. Fragments and threads spawned by an activity run in the context of the activity itself. So, if the activity is destroyed, the fragments and threads associated with it will be destroyed as well.

One globally defined Android intent allows an activity to be displayed as an icon on the launcher (the main app list on an Android device). Because the vast majority of apps want to appear on the main app list, they provide at least one activity that is defined as capable of responding to that intent. To be used, the application's activities information must be properly registered in the *App Manifest*.
Usually, the user starts from a particular activity and move through other ones creating a stack of activities all related to the one originally launched; This stack of activities is called *task*. The user can then switch to another task by clicking the HOME button and starting another activity stack from the launcher.
Since there can be only one activity on the screen at a time, when a new one is started, the previous activity pauses and the operating system keeps it in a stack (*back stack*), while the new one comes to the foreground. When operations on the current activity are ended or the BACK button is pressed on the device, the current activity is removed from the stack (and destroyed) and the previous one return to the foreground. The Activity Manager is concerned with stack and activity lifecycle management. Stop of an activity due to the launch of a new one is notified by the callback methods of the activity lifecycle (*Figure 2.2*) that allow to adequately manage the transition state of the activity.

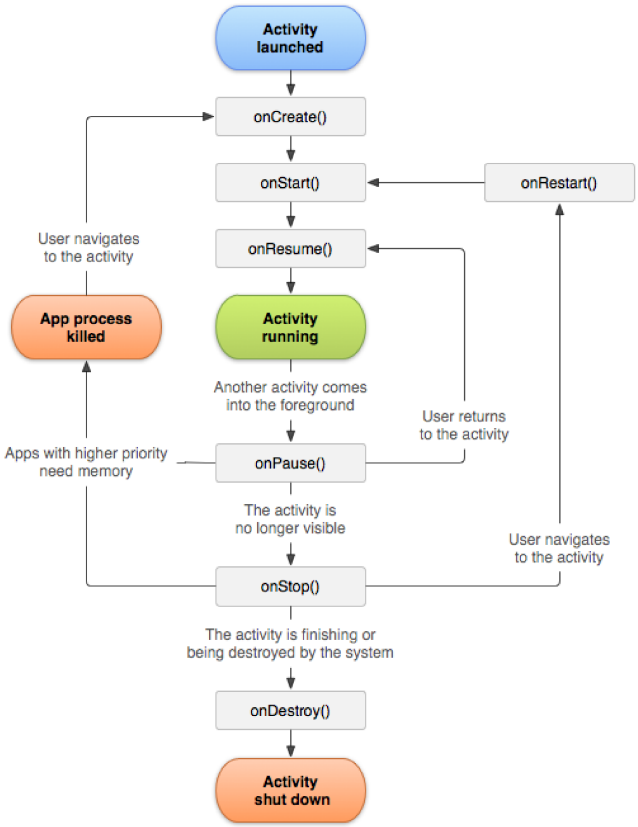


Figure 2.2: Illustration of the Activity lifecycle [6]

The lifecycle of an activity is the set of states that it goes through from the time it is first created until it is destroyed. The Activity class provides a core set of six callbacks, called from the system, that allow the activity to know that a state has changed:

- *onCreate* is invoked when the activity gets first created. All activities must implement this method in order to initialize the Activity object. This method receives a *Bundle* object as parameter that may contain the activity's previously saved state. At this stage, the activity is not yet visible to the user.
- *onStart* is called when the activity becomes visible to the user. Although the activity is now visible, the user still cannot interact with it.
- *onResume* is called every time the activity comes into the foreground. This is the state in which the app interacts with the user. The activity remains in this state until something happens to take focus away.
- *onPause* is called when the activity is no longer in the foreground. This method can be used to release system resources. The activity remains in this state until either the activity resumes or becomes completely invisible to the user.
- *onStop* is called when the activity is no longer visible to the user. In this state, the app should release almost all resources that are not needed and that might leak memory.
- *onDestroy* is called before the platform is destroying the activity.

Service

The Android framework provides an application component, known as Service, to enable applications to perform longer-running operations in the background. A Service component does not provide a user interface. Services can be used within the same application or can also be made available to components outside an app. Another component can start the service and let it run or bind to it in order to interact with it. There are three different types of services:

- *Scheduled*: characterized by jobs and requirements for network and timing. The system gracefully schedules the jobs for execution at the appropriate times.
- *Started*: can run in the background indefinitely, even if the component that started it is no longer executing.
- *Bound*: offers a client-server interface that allows components to interact with the service, send requests, receive results, and even do so across processes with interprocess communication (IPC). A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

All services must be declared in the application's manifest file, just as activities and other components.

Content Provider

The Android platform provides an application component, known as the Content Provider, that manages access to a structured set of data, and provide a way to share data with other apps. The content provider achieves this by providing proper data encapsulation and also security. A content provider supports a variety of data storage sources, including both structured data, such as a SQLite relational database, or unstructured data such as image files, and can also return standard or MIME types. The content provider needs to be declared like other application components in the manifest file.

Broadcast Receiver

The Android platform provides a system-wide message bus facility called broadcast messages , similar to the publish-subscribe design pattern. This facility enables applications and the Android system to propagate events and state change information to the interested parties by broadcasting an Intent as a message. When an event of interest occurs, a Broadcast Receiver is triggered to handle that event on the app's behalf. Even if the application is not running, it still receives intents that can trigger further events. In addition, Broadcast Receiver permissions restrict who can send broadcasts to the associated receiver. An application can declare a receiver in the manifest file.

2.1.3 App Manifest

The Android App Manifest is an indispensable XML (eXtensible Markup Language) file that must reside at the root directory of the app's sources as *AndroidManifest.xml* (with precisely that name). When the application is compiled, the manifest is transformed into a binary format.
The manifest file provides essential information about the app to the Android system, which the system must have before it can run any of the app's code. Basically, it declares the application components, their visibility, the capabilities required to run the app, the minimum level of the API required, the list of permissions required, any hardware requirements, libraries, which icon to use on the Application menu, and a lot of other configurations.

2.1.4 Intents

Intents are Android's asynchronous mechanism for inter-component communication (within the same application or among different applications). They are used by the Android system for starting an Activity or Service, for communicating with a Service, to broadcast events or state changes, for receiving notifications using pending intents, and to query the Content Provider.
Intents themselves are objects containing information on operations to be performed or, in the case of Broadcast Receivers, on details of an event that occurred. Once the intent is created and dispatched through the Android framework, the Android platform resolves the intent to find candidates that can provide the requested action. There are two types of intents that define how they get resolved and dispatched:

- *Explicit intents*: provide the component to start by name (the fully-qualified class name).
- *Implicit intents*: do not specify a component, but instead declare a general action to perform, which allows a component of another app (capable to perform this action) to handle the request.

Implicit intents rely on the system to find an appropriate available component to be started. For this to be possible, each component can provide Intent Filters (structures in the app's manifest file that specifies the type of Intents a component is willing to handle). Likewise, if a component does not have any intent filters, then it can only receive explicit intents. When an implicit intent is created, the system then compares contents of the intent to the intent filters declared in the manifest file of other apps on the device. If the intent matches an intent filter, the system starts that component and delivers it the Intent object. If multiple intent filters are compatible, the system displays a dialog so the user can pick which app to use.

2.2 GUI Testing

The GUI (Graphical User Interface) is the means that allows users to interact with the software; It responds to user-generated events by executing the code behind. One of the commonly used methods to detect defects in a software is to exercise its GUI. Unlike other test approaches where test-suites consist of test cases that invoke software system methods and capture the returned value, the GUI-based approach provides methods for detecting and recognizing GUI components, exercising events on the GUI (e.g., click), providing inputs to GUI components (e.g., fill in text fields), and checking the representations of the GUI to see if they are consistent with the expected ones. This makes the GUI-based testing particularly difficult and its implementation strictly dependent on the technology used. On the other hand, this testing technique is easily executable and automated. GUI testing techniques can be divided into:

- *Model-based testing*: based on an existing model (formal description of the application under test, in particular its GUI) that is sufficiently detailed to allow automatic test cases generation.
- *Random testing*: in the absence of a model, the application is tested randomly, or pseudo-randomly, in search of any failures (caused by unmanaged exceptions).
- *Model-learning testing*: starting without an available model, the application is exercised methodically, following a specific exploration strategy. This approach allows to detect any failures and, in addition, to build a model (usable for test cases generation).

In the GUI testing context, the model plays an essential role; In fact, an appropriate user interface model allows automatic test case generation and makes testing automation feasible.

Realizing a formal model could be an expensive process. The GUI must allow the users to use all software features and must respond properly to all possible user interactions. Usually, the GUI has a proportional complexity (often comparable) to that of the code behind; This means that the model will be just as complex. In addition, it could be updated both when a bug is discovered and when a new feature is added implying that the model needs to be updated continuously. A significant reason of complexity is introduced by the fact that GUI tests must be done by observing the AUT's behaviors following specific input events in order to provide reliable information about the tested software. Adopting a shallow test strategy could lead to a model that does not correspond with the software's behavior and, hence, is invalid. In all this, the GUI model must be sufficiently detailed to distinguish between different types of user inputs and different types of GUI components. As a result, a technique that automatically generates a GUI model is crucial for effective test automation.

2.2.1 GUI Ripping technique

GUI Ripping is a reverse engineering technique that aims to build a GUI model of an existing software application by dynamically interacting with its user interface. It is based on an automatic exploration of the application's graphical interface performed by simulating real user inputs. This exploration allows detecting any failures due to unmanaged exceptions and deriving a GUI model. This model is used for test case generation, which can then be automatically performed for different purposes (e.g., crash or regression tests).
The purpose of the GUI ripping technique is to discover as much structural information about the GUI as possible using automated algorithms and eventually some human input (e.g., password). Thus, the current user interface is analyzed by getting information needed to characterize it: All the widgets and their properties are extracted from the GUI. Properties also include information about events that can be fired on the widgets.

This technique is generally implemented by means of an algorithm which, once the current GUI state is described, involves generating tasks (sequences of inputs on GUI widgets) that are added to a task list. If the task list contains a task to be executed, it is extracted (according to the used traversal strategy) and performed, otherwise the process ends. After the task is executed, the reached GUI state is analyzed, described and compared with the already visited states. If the state has already been visited, the next task is extracted, otherwise the current state is analyzed in order to generate new tasks. The exploration of the application is thus performed methodically and the information obtained is used in subsequent iterations with the aim of reaching previously unexplored states. In general, the exploration terminates when all GUI states are considered covered.

2.3 Testing in Android

Google provides an Android Testing Framework that is part of the Android SDK and is built on top of JUnit (a framework to write repeatable tests) extended with an Instrumentation Framework and Android-specific testing classes. Instrumentation allows to control all the interactions of the application under test with the surrounding environment and also permits the injection of mock components[1].
There are two types of tests to be created in an Android application:

- *Local unit tests*: these tests run locally on the Java Virtual Machine (JVM) without accessing to functional Android framework APIs. This testing approach is efficient because it avoids the overhead of deploying the target app and unit test code onto a physical device or emulator every time the test is run. Consequently, their execution time is greatly reduced. With this approach, it is normally used a mocking framework, like *Mockito* [7], to fulfill any dependency relationships.

[1] Mock objects are fake objects that simulate the behavior of real objects, but are totally controlled by the test.

- *Instrumented tests*: these are tests that must run on an Android hardware device or an Android emulator as they need to exploit Android APIs. Instrumented tests are built into an APK that runs on the device alongside the app under test. The system runs the test APK and the AUT in the same process, so that tests can invoke methods, modify fields, and automate user interaction with the app.

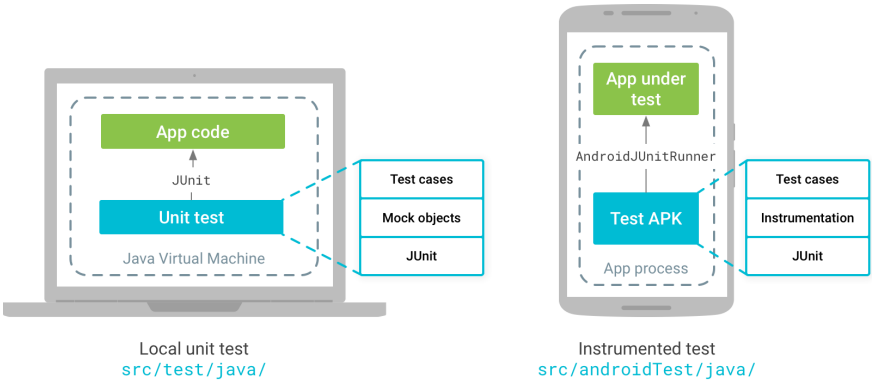


Figure 2.3: Android Test types [8]

However, the *local unit tests* and *instrumented tests* described above are just terms that help distinguish the tests that run on a local JVM from the tests that run on the Android platform (on a hardware device or emulator). A more refined testing categorization is described in the following table.

Table 2.1: Android test types [8]

Type	Subtype	Description
Unit tests	Local Unit Tests	Unit tests that run on the local JVM. It is useful to use this type of test to minimize execution time when tests have no Android framework dependencies or when it is possible to use mock objects for simulating real objects behaviors.
	Instrumented unit tests	Unit tests that run on an Android device or emulator. These tests have access to *Instrumentation* information, such as the *Context* of the app under test. They are needed when mock objects cannot satisfy Android dependencies required by the tests.
Integration Tests	Intra-app Components	They have the objective to verify that the target app behaves as expected when a user performs specific inputs. UI testing frameworks like **Espresso** allow to programmatically simulate user actions and test *intra-app* user interactions.
	Cross-app Components	They have the objective to verify the correct behavior of interactions between different apps. UI testing frameworks that support *cross-app* interactions, such as **UI Automator**, allow to create tests for such scenarios.

The **Android Testing Support Library** provides an extensive framework for testing Android apps by providing APIs that allow to build and run tests for Android applications. The library includes the following instrumentation-based APIs:

- *AndroidJUnitRunner*

  A JUnit 4-compatible test runner for Android that allows to run JUnit 3- or JUnit 4-style test classes on Android devices, including those using the Espresso and UI Automator testing frameworks. The test runner handles loading the test package and the app under test to a device, running tests, and reporting test results.

- *Espresso*

  A UI testing framework suitable for functional UI testing within a single app. The Espresso testing framework provides a set of APIs to build UI tests for testing user flows within an app. These APIs allow to write automated UI tests that are concise and that run reliably. Espresso is well-suited for writing **white box**-style automated tests, where the test code utilizes implementation code details from the app under test. Espresso tests require running on devices provided with Android 2.2 (API level 8) or higher.

- *UI Automator*

  A UI testing framework suitable for cross-app functional UI testing across both system and installed apps. The UI Automator testing framework provides a set of APIs to build UI tests that perform interactions on user and system apps. The UI Automator testing framework is well-suited for writing **black box**-style automated tests, where the test code does not rely on internal implementation details of the target app. UI Automator tests require running on devices provided with Android 4.3 (API level 18) or higher.

2.4 UI Automator 2.0

UI Automator is a black-box testing framework suitable for cross-app functional UI testing. It provides a set of APIs to create UI tests that perform interactions on user apps and system apps, and uses its execution engine to automate and run those tests on Android devices. The UI Automator testing framework includes:

- APIs that support cross-app UI testing;
- An API able to retrieve state information and perform operations on the target device;
- A viewer able to inspect the layout hierarchy of Android apps.

The UI Automator testing framework is an instrumentation-based API and works with the *AndroidJUnitRunner* test runner. It requires Android 4.3 (API level 18) or higher.

2.4.1 UI Automator APIs

The UI Automator APIs allow to write robust tests without needing to know about the implementation details of the app under test. These APIs allow to capture and manipulate UI components also across multiple apps by exploiting ***UiAutomation***[2] [9].
Exploiting UI Automator APIs, tests can look up UI components by using convenient descriptors such as the text displayed in that component or its content description. An element can also be targeted by its location in a layout hierarchy. In fact, UI Automator automatically analyzes the screen of the Android device, and constructs the relative widget hierarchy tree, where widget nodes have parents-children or sibling relationships with each other. These relationships are encoded in an **index**[3] property and, thus, by using index values, each widget can be uniquely identified as a cumulative (from the root node to the target node) index sequence.

[2] UiAutomation is a low-level testing API that allows Instrumentation tests to test across application boundaries. It supports full-screen introspection, taking screenshots, changing device orientation and injection of *raw input*.

[3] Google emphasizes in the official documentation that using the index could be unreliable. Instead, it is advisable to use the **instance** property.

2.4.2 Access to device state

The UI Automator testing framework provides a *UiDevice* class to access and perform operations on the device on which the target app is running. By means of its methods, it is in fact possible to retrieve device properties such as current orientation or display size, and perform device-level actions such as changing the device rotation, pressing the BACK and HOME buttons, or taking screenshots of the current window.

2.4.3 UI Automator Viewer

The *uiautomatorviewer* [10] tool provides a convenient GUI to scan and analyze the UI components currently displayed on an Android device connected to the development machine. This tool is useful to inspect the layout hierarchy and retrieve the properties of UI components that are visible on the foreground of the device. This information allows developers to create more fine-grained tests using UI Automator.

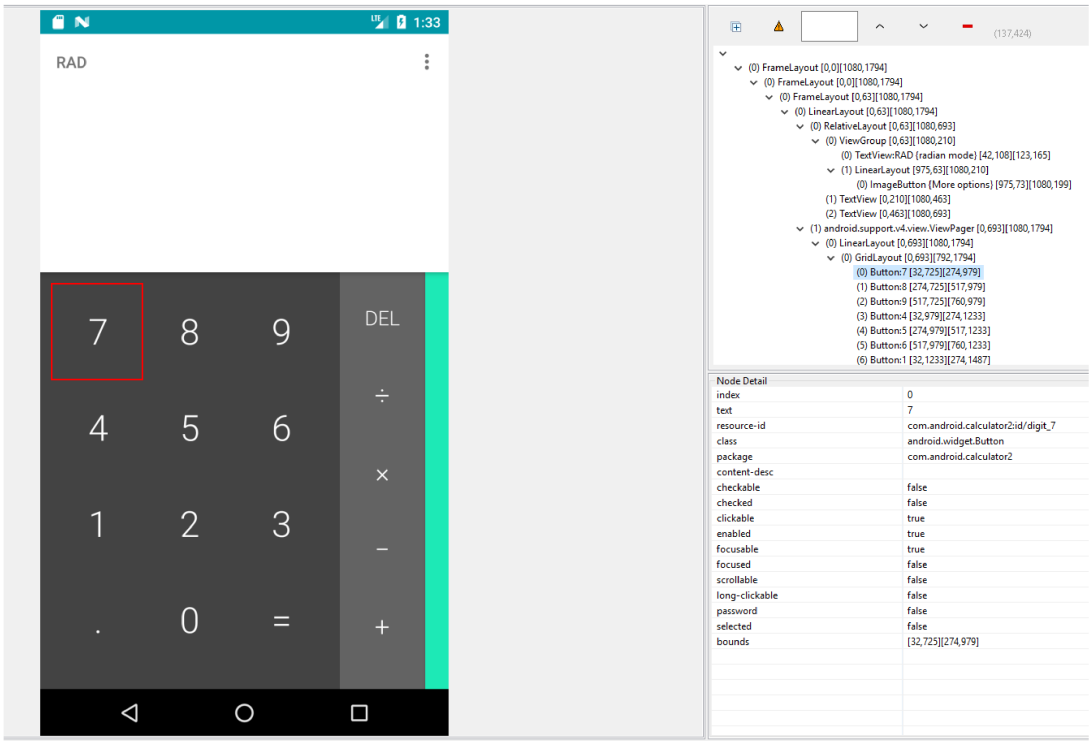


Figure 2.4: UI Automator Viewer

3 Theory

The main purpose of this chapter is to provide information on the state-of-art of mobile GUI testing approaches for Android applications. These techniques will be categorized and described, emphasizing, in the case of *model-based testing*, the importance of an appropriate representation of the GUI and of an effective comparison criterion to determine the equivalence between multiple GUI states. During the discussion of these topics, security-related issues will also be addressed and, in particular, some techniques that might be used by malware to evade automated runtime analysis will be highlighted.

3.1 State-of-the-art approaches for automated GUI testing of Android applications

Regardless of the scope for which they are designed, the primary objective of each testing process is to verify that the applications under test do not exhibit unexpected behaviors. For this purpose, automated testing techniques exercise as much behaviors of the AUT as possible by exploiting inputs generation that can be obtained *randomly* or following a *systematic* or a *model-based* approach. In this latter case, exploration is driven by a *model* of the AUT, which can be built statically or dynamically. Testing tools can generate these events by using either *black-box* testing methods or *white-box* testing methods. In the first case, the automated tests are performed without relying on implementation details of the application. In the second case, the investigation process is based on internal logic and structure of the code; In white-box testing it is necessary a full knowledge of the source code. Finally, a *grey-box* approach is also possible. It represents a hybrid testing based on limited knowledge of the internal details of the AUT. In particular, in Android testing may be possible extract high-level properties of the app, such as the list of Activities. Two essential APK components for Android static analysis used in grey-box tests are the Android manifest, which describes permissions, package name, version, referenced libraries, and app components, and DEX classes, which contains all Android classes compiled into a Dalvik compatible file format.

Previously introduced in ***Introduction Chapter***, the *Table* 1.1 provides an overview of the main existing Android testing tools presented in the literature and in the following sections we will describe these techniques in more detail.

3.1.1 Random-based exploration strategy

A first category of automated testing techniques, also known as *fuzz testing*, use a black-box approach employing a random selection of events from a set of possible ones, such as GUI or system events. This approach is unsuitable for generating highly specific inputs that control the app's functionality, but on the other hand, it is easy to implement robustly. Input generators based on a random exploration strategy can efficiently generate a large number of events and, for this reason, are widely used in stress testing. However, tools that implement a random-based strategy are not aware of how much behavior space of the app has already been covered and might therefore generate redundant events that do not help achieving a satisfactory exploration. Finally, a random input generator does not have a stop criterion for determining the success of the exploration, but usually it relies on a manually specified timeout or threshold value of generated events.

The Android platform includes a fuzz testing tool, called *Monkey* [1], able to generate and perform, in a mobile emulator or device, a pseudo-random stream of user events, which include both UI events such as clicks and gestures, and system-level events such as screenshot capture and volume adjustment, in a random yet repeatable manner. In fact, although the interactions are random, Monkey is based on a seeding system and therefore the use of the same seed involves the generation of the same sequence of actions. It is the most popular tool to perform black-box stress tests on Android apps, in part because it is Google's official testing tool and does not require any additional installation effort, since it is part of the Android developer toolkit. Moreover, the tester can configure the Monkey tool for selecting event types and its frequencies, or for specifying the number of events to attempt or other operational constraints (e.g., restricting the test to a single package).

DynoDroid [48] generates randomized inputs in a mobile device emulator, but it has several features that make its exploration more efficient compared to Monkey. The implementation supports the generation of both UI and system events and allows to manually provide inputs (e.g., for authentication) when the exploration is stalling.
Dynodroid exploits the *Hierarchy Viewer*, a tool packaged with the Android platform able to infer a UI model at runtime, for determining a layout's hierarchy of the current screen. However, it needs to modify Android framework (SKD) for gathering information about broadcast receivers and system services for which the app is currently registered.
Dynodroid is based on an *observe-select-execute* cycle. It first observes which are the relevant events in the current app state. An event is relevant to an app, if the app has registered a listener for that event. In the selection phase, one of the observed events is randomly selected to be executed in the final step. The authors propose three different selection strategies: *Frequency*, *UniformRandom* and *BiasedRandom*. The Frequency strategy has a bias towards least recently used events. The UniformRandom strategy selects an event uniformly at random. The BiasedRandom strategy randomly select an event also by considering the contexts the events belong. The context for an event at a particular instant is the set of all relevant events at that instant: The chance that an event will be selected for execution in a context depends on the number of times that it has been chosen (or not chosen) in the past (or in the current) selection stages.

3.1.2 Systematic exploration strategy

Since some application behaviors can only be revealed after providing specific input sequences, some Android testing tools address this problem by using more sophisticated techniques, such as *symbolic execution* and *evolutionary algorithms*, to guide the exploration towards previously uncovered states. Implementing a systematic strategy involves benefits in exploring behavior that would be hard to reach with random techniques. Compared to random techniques, however, these tools are difficult to scale due to the path explosion problem (exponential growth of possible paths with increasing program size).

The key idea of symbolic execution is to systematically explore feasible paths of the program under analysis by reducing the search space from an infinite number of possible data inputs to a finite number of data domains (represented by symbolic inputs). So, instead of executing a program on a set of sample inputs, the program is symbolically executed for a set of classes of inputs. This implies that each symbolic execution result may be equivalent to a large number of normal test cases. This approach avoids generating redundant inputs and allows to generate highly specific inputs. Moreover, symbolic execution is not black-box and requires instrumenting of, at least, the app under test.

ACTEve [32] is a *concolic-testing* tool that symbolically tracks events from the point where they originate to the point where they are ultimately handled in the app.
Concolic testing is a hybrid software testing technique combining concrete execution of a program (random or given inputs, along specific paths) with symbolic execution (that aims to generate new concrete inputs that forces the program to take unexplored paths).
The key concept (though simplified) of the algorithm implemented by ACTEve lies in the fact that, if an event sequence e_n is subsumed by another event sequence e'_n that ends with an event that have no effect (i.e., the state remains unchanged), then the algorithm prevents extensions of e_n (which will not be considered in any future iteration).
To fulfill this task, ACTEve needs to instrument the app in addition to the Android SDK. It supports the generation of both system and UI events.

Evodroid [49] is the first evolutionary approach for system testing of Android apps. Evolutionary testing is a form of search-based testing that exploits a population-based metaheuristic optimization algorithm, where an *individual* corresponds to a sequences of test inputs, and a *population* composed of many individuals evolves according to certain heuristics, for automatically generating the tests.
Evodroid automatically extracts from the code of the application two models: The Interface Model (based on static analysis of Manifest and XML configuration files) and the Call Graph Model (based on code analysis by using MoDisco [36]). The Call Graph Model extends a typical app's call graph (representation of explicit method call relationships) enriched with information about the implicit call relationships caused by Intents. EvoDroid uses this model to determine the parts of the code that can be searched independently, and evaluate the quality (fitness) of different test cases, based on the paths they cover through the graph, thus guiding the search. EvoDroid executes the test cases in parallel, possibly on the cloud, and the results are then evaluated using a fitness function evaluated by considering code coverage and uniqueness of the covered paths.

A^3E-Targeted [34] has as its main objective to reach a rapid exploration of activities. To achieve this goal, it relies on a component that, by means of a preliminary static bytecode analysis, extracts information about valid Activity transitions and builds a Static Activity Transition Graph of the app, which is then used for systematically exploring the running app. In this way, it can list all the Activities which can be called from other apps or background services directly without user intervention, and generates calls to invoke those Activities directly. The Targeted Exploration strategy is useful in the situations where not all Activities can be invoked through user interaction. In this regard, the so called *Exported Activities* are characterized by Intent Filters, and can be accessed only with special request from within or outside the application. They are marked as such by setting the parameter **exported=true** in the Manifest file.

SAPIENZ [50] is a multi-objective tool for Android testing that seeks to maximize code coverage and fault revelation, while minimizing the length of fault-revealing test sequences. It combines random fuzzing, systematic grey-box and search-based testing techniques. SAPIENZ employs both static analysis of the APK and multi-level instrumentation in order to perform a search-based testing approach. In particular, it uses fine-grained instrumentation at the statement-level (white-box). In case only the binary APK file is available, it uses

repackaging[1] to instrument the app at method-level (grey-box). However, if repackaging is disallowed, it proceeds in a non-invasive manner for calculating an activity-level coverage, also called "skin coverage" by the authors (black-box). Moreover, SAPIENZ also uses evolutionary algorithms to find new interesting combinations of events that may allow reaching parts of the application not previously visited.

3.1.3 Model-based exploration strategy and Model-learning techniques

The idea behind model-based testing is to create a suitable set of user input sequences starting from the model of the target program. This approach uses human and framework knowledge to abstract the input space of a program's GUI, thus reducing redundancy and improving efficiency. In general, a model-based testing approach requires testers to provide a model of the AUT, though automated GUI model inference tools exist (e.g., *GUITAR* [53]) and some static and dynamic analysis techniques could be used for inferring this model. The main limitation of model-based testing tools with an active learning approach (i.e., model-learning techniques) lies in the state representation they use, as most of them represent new states only when some event triggers changes in the GUI. However, some events may change the internal state of the application without affecting the GUI. In such situations, these techniques would miss the change and continue the exploration considering the event as irrelevant. A common scenario in which this problem occurs is in the presence of services, as services do not have any user interface.

AndroidRipper [31], successor of *GUI Ripper* [29], which subsequently became *MobiGUITAR* [30], dynamically analyses the application's GUI with the aim of obtaining sequences of events fireable through the widgets.
Starting from an initial state (the GUI state shown at the beginning, when the ripper starts exercising the AUT) and employing a well-defined GUI traversal strategy, it builds a *GUI Tree*, which is a graph whose nodes represent the states of the GUI and edges describe event-based transitions between consecutive states encountered during the ripping process. For each new state found during the execution, AndroidRipper keeps a list of unfired widgets belonging to the current GUI state, generates related fireable events, and systematically triggers them; The exploration is stopped when all the GUI states are considered explored. The authors propose two different heuristics to determine when a GUI state can be considered as equivalent to an already visited one. The first heuristic considers two GUI states equivalent if they belong to the same Activity class. The second heuristic is more restrictive since two GUI states are considered as equivalent only if they contain the same set of widgets.

ORBIT [63] uses a grey-box model-learning approach based on a combination of a preliminary static analysis of the AUT source code, for extracting a set of UI events triggerable from each Activity, and a dynamic GUI ripping process, for inferring a model of the AUT.
ORBIT implements an optimized depth-first strategy, which tries to exploit the BACK button as much as possible in order to reach a previously seen state without restarting the app, as restarting is a significantly expensive operation. In this regard, it is important to note that this button is context-sensitive and, therefore, it is not a reliable mechanism that leads to exactly the previous state. In fact, the BACK button can allow to reach any ancestor states (or also to the initial screen) and, for that reason, it should be used with caution. The tool is able to model the GUI behavior of an Android app as a finite-state machine (FSM), where nodes are the GUI states (also called *visual observable states* by the authors) and transitions among these states are constituted by the user-actions fired at runtime. A visual observable state is composed of a hierarchy tree of GUI components (classified by the authors in executable components and display components), as well as of a set of attributes characterizing each executable component (which support the user actions detected during the static analysis). The execution is stopped when ORBIT no longer detects new states to be explored.

[1] It is a process through which the application is reverse engineered, some specific payloads are added, and the modified application is rebuilt.

A further model-learning testing technique has been implemented by *SwiftHand* [37]. It aims to achieve code coverage quickly by learning and exploring an abstraction of the model of the GUI of the app. It seeks to optimize the exploration strategy to minimize the restarts of the app trying to extend the current execution path by selecting a user input enabled at the state.
SwiftHand implements traversal strategies of the model based on the Angluin's *L** learning algorithm [33] and uses execution traces generated during the testing process to infer a model of the GUI, represented by what the authors call *Extended Deterministic Labeled Transition System* (structure consisting of states connected by transitions labeled with actions). The model is exploited to choose user inputs that would take the app to previously unexplored states. As SwiftHand triggers the newly generated user inputs and visits new screens, it expands the learned model, and it also refines the model when it detects discrepancies between the model learned so far. During the learning process of the model, a comparison criterion is required to determine the equivalence between user interface states. For this purpose, SwiftHand considers two GUI states equivalent if they have the same set of enabled (available for triggering) user inputs. An enabled user input is only considered according to its type and the bounding box coordinates within the screen where it is enabled, without caring about the content of GUI components such as colors or text content.

PUMA [43] is a programmable framework containing a generic UI automation capability (often called Monkey) that exposes high-level events for which users can define handlers. The novelty of this tool, in fact, is not in its exploration strategy, but rather in its design. Since programmable, PUMA can be extended to implement different dynamic analysis on Android apps. It provides a programmable finite-state machine representation of the AUT, thus allowing the testers to implement different exploration strategies. Additionally, it allows redefining the state representation and the logic to generate events. PUMA exposes a set of configurable hooks that can be programmed with a high-level event-driven scripting language, called *PUMAScript*. This language separates analysis logic from exploration logic, allowing to specify exploration strategies and separately specify the logic for app property analysis. This system relies on the instrumentation of app binaries.

DroidBot [47] is an input test generator that is able to interact with an Android application without instrumentation, making malware analysis a possible field of application. In fact, since many malware encrypts their code or verifies their signature before doing malicious actions, it may be impossible to instrument them or ensure consistency between the instrumented application and the original one. DroidBot models the explored states, as a state transition graph built on-the-fly, exploiting a set of Android-integrated test/debugging utilities. It maintains the information of the current state and monitors the state changes after sending a test input to the device. If changes are detected, the test input and the new state are added to the graph, as a new edge and a new node. Currently, DroidBot uses a content-based comparison, where two states with different UI contents are considered as different nodes. DroidBot can generate the call stack trace for each test input, which contains the methods of the app and the system methods triggered by the test input. It uses the call stack as an approximate metric to quantify the effectiveness of test inputs.

3.2 Runtime analysis detection: techniques and countermeasures

According to Tam et al. [58] and Wei et al. [62], Android malware are increasingly using anti-analysis techniques, such as native code invocation, dynamically-loaded code, Java reflection, and code obfuscation, which often render both human inspections and static analyses ineffective. In this regard, the use of dynamic analyses can overcome such limitations, allowing to detect those app's behaviors that only manifest at runtime.

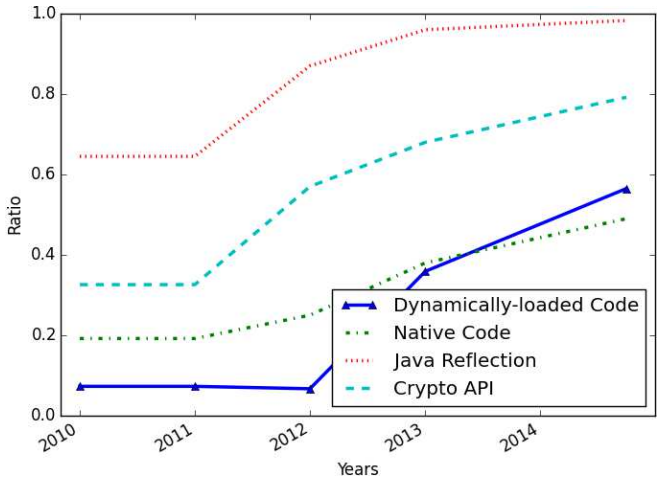


Figure 3.1: Anti-analysis malware techniques trends [57]

Because the malware is running during analysis, the choice of a suitable testing environment is a crucial factor for the success of the analysis itself: If the analysis operates at the same permission level as a malicious software, the malware could detect and bypass it. On the other hand, if the analysis was to work at a lower level (e.g., at kernel level), then it would increase security, but make it more difficult to intercept the AUT's data and communications. To cope with these complications, several methods have been proposed in the literature [55] that use simulated system environments (such as virtual machines or emulators) for facilitating the analysis process and providing greater control over the execution of processes.
Nevertheless, malware frequently refrain from malicious behavior if they detect running in an emulated environment [44, 54, 61]. As countermeasures to anti-emulation techniques, recent research proves the effectiveness of using real devices [40, 52] and tailored emulated environments [42]. Despite this, other aspects should be taken into account during dynamic analyses to counter the detection of an analysis environment.

Since Android malware, as mobile applications, are event-driven systems (EDS), their malicious behaviors could be triggered only after specific user input sequences or system events. For this reason, automated exploration techniques are often used in conjunction with dynamic analysis in order to examine as much of an app's behaviors as possible without user intervention. However, this approach might be identified by a malware: In fact, by monitoring interaction patterns, it may be possible to determine whether the application is used by a human user or by an automated testing tool. Hence, simulating a behavior as much human-like as possible in automated exploration strategies could be a crucial aspect to be considered in the analysis tools in order to not be detected. Furthermore, the effectiveness of dynamic analysis frameworks heavily depends on the input generation strategy adopted and how much it is capable of exploring the runtime behaviors of the application under test. Model-based testing aims to generate events based on specific patterns or on a model of the AUT which could be derived by analyzing the app's code or by exploring its GUI. Test cases generated with this approach are usually more effective and efficient for triggering malicious activities than the ones generated with fuzzing-based techniques. For this purpose, Android black-box testing automation frameworks can be used in order to inspect the layout hierarchy and extract the meaningful UI properties, thereby avoiding generating invalid actions.

However, the use of these frameworks can lead to generate predictable input events, fired at regular and short interval, in order to seek covering all potential UI paths in limited time. This modus operandi significantly differs from a human utilization and therefore provides a feasible criterion for evading dynamic analysis.
On the basis of this principle, Diao et al. [40] design a mechanism for capturing such programmed interactions to distinguish human users from testing platforms. In this way, before

exposing its malicious behaviour, a malware can analyse the harvested events and proceed with the execution of the malicious payload only if the event sequence is determined to be produced by human user.

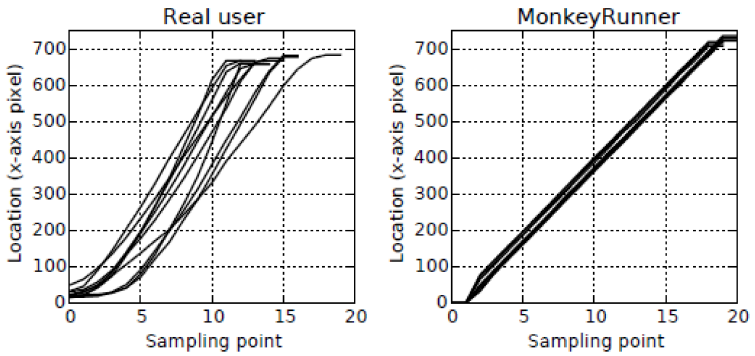


Figure 3.2: Swiping trajectory: real user vs. exploration tool [40]

On Android platform, there are two classes for representing the types of user events: *MotionEvent* [11] for touchscreen movements and *KeyEvent* [12] for key pressures.
The object associated with an event related to a user manual input is initialized directly on the mobile device with the information regarding the hardware onboard. Conversely, events generated by the dynamic analysis tools are represented by objects built from the UI testing framework and some parameters are filled with fixed dummy values.

Table 3.1: MotionEvent: real vs. simulated [40]

Parameter	Real	Simulated
ToolType	1: TOOL_TYPE_FINGER	0: TOOL_TYPE_UNKNOWN
DeviceId	[non-zero value]	0
Device	valid	null

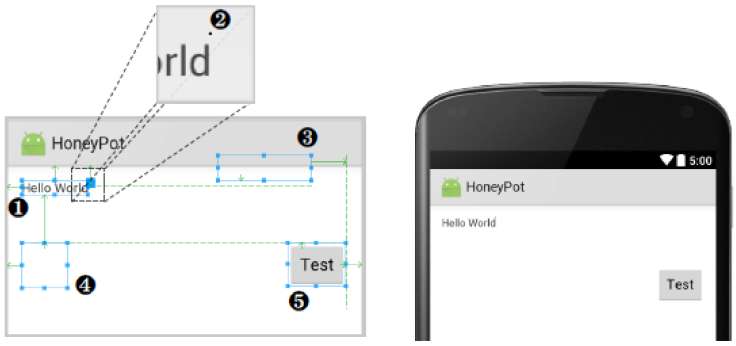
Table 3.2: KeyEvent: real vs. simulated [40]

Parameter	Real	Simulated
ScanCode	[non-fixed value]	0
DeviceId	[non-fixed value]	-1
Device.Name	[non-fixed value]	Virtual
Device.Generation	[non-fixed value]	2
Device.Descriptor	[non-fixed value]	af4d26ea4cdc857cc0f1ed1ed51996db77be1e4d
Device.KeyboardType	1: non-alphabetic	2: alphabetic
Device.Source	[non-fixed value]	0x301: keyboard dpad

Another aspect that malware can exploit to detect dynamic analysis tools lies in the fact that an automated exploration strategy could reach AUT states or exercise UI components inaccessible from a human user.
Some dynamic testing tools exploit static techniques for parsing the Manifest file and, thus, become aware of all the Activities that belong to an application. Although this approach would be beneficial to achieve a high code coverage efficiently, on the other hand, it can

be exploited by malware such as *honeypot*[2] (by creating Activities which can not be reached through user interactions, such as exported Activities) to detect running in a testing environment.

Even the graphical user interface may contain traps that could allow a malware to identify a dynamic analyzer: Not-visible widgets (e.g., as tiny or out of the screen) are normally not tapped from human users, instead they remain perfectly accessible (albeit invisible) from analysis tools.



(a) Real laylout (b) Human's view

```
(0) FrameLayout[0,0][720,1280]
  (0) View[0,50][720,146]
  (1) TextView:Hello World[32,178][178,216]
  (2) ImageButton[176,178][178,180]
  (3) Switch:ON[408,124][614,180]
  (4) ImageButton[32,318][132,418]
  (5) Button:Test[560,318][688,414]
```

Figure 3.3: GUI honeypot [40]

To avoid being detected, automated testing tools should therefore determine the visibility of UI components from a human's-eye point of view.

3.3 Android UI Testing Frameworks

This section provides an overview of most popular Android UI testing frameworks and highlights the peculiarities of each one. This information will be useful to select the appropriate framework to be used for our implementation according to project requirements. Among the various frameworks available for testing the GUI of Android apps, we focused on Espresso, UI Automator, MonkeyRunner, Robotium, Appium, and Calabash frameworks, as they are currently the most active and popular projects in this area [46].

3.3.1 Espresso and UI Automator

The Espresso [13] and UI Automator [14] are two testing frameworks provided by Google and inbuilt with the Android Open Source Project [4]. They use Java as their test-development language and are able to execute tests both on real devices and on emulators, but not in parallel.

Espresso testing framework provides APIs for writing functional UI tests for a single app. It is suitable for a white-box testing and requires an API level of 10 (i.e., Android 2.3.3) or higher. It allows to locate specific UI components according to some criteria (e.g., id, class

[2]Lance Spitzner [56] formally defined a honeypot as "a security resource whose value lies in being probed, attacked, or compromised". In other words, it is a decoy computer system for detecting, deflecting or in some manner counteracting attempts at unauthorized, illicit or unconventional use of information systems.

name, current state, or textual content). Once a View has been selected, Espresso allows the execution of user actions on it: It is possible to fill text fields, perform actions, and even analyze outputs and changes to verify that they are as expected.
The framework uses synchronization mechanisms, and is capable of automatically checking whether the application is in a stable state (i.e., the main thread is idle) before performing operations on widgets.

UI Automator is an Android framework suitable for cross-app functional UI black-box testing. It provides a set of APIs that allow to perform interactions both on user apps and on system applications, but requires an API level of 18 (i.e., Android 4.3) or higher. The ability to build tests that span over multiple applications is a really important feature characterizing this framework, since the Android architecture encourages the switch between different applications (by means of Intents) to handle particular user requests. In addition to the functionalities provided by Espresso (i.e access to the UI), UI Automator enables the access to the device status, in order to retrieve device properties (e.g., orientation, screen resolution or display size), and perform user actions on the device (e.g., pressing the Back or Home buttons, or taking screenshots).
The framework does not wait for the application to be in the steady state like the Espresso framework: the test developer has to be careful that the application is in a stable state before continuing the test run.

UI Automator framework also includes a tool (*uiautomatorviewer* [10]) which allows to inspect the layout hierarchy and retrieve the information about all the visible UI components currently displayed on the Android device connected to the development machine. This information can help developers to create application-specific tests using UI Automator.

3.3.2 Monkeyrunner

Monkeyrunner [15] provides an API for writing scripts that control an Android device or emulator. It is included in the Android SDK, developed by Google, and can be extended by developing Python-based modules. Its scripts (written in Jython, an implementation of Python which is designed to run on the Java™ Platform) can be used for installing Android apps, sending user inputs, and taking and comparing screenshots. Monkeyrunner also allows to apply test suites to multiple devices or emulators.
Unlike other frameworks, it seems to lack high-level methods for retrieving screen items, making code maintenance and its reuse complicated. In particular, UI object selection is based on the object's location (x, y coordinates), which can change when the user interface of the tested application is subject to changes, and then the scripts may require continuous adjustments.

3.3.3 Robotium

Robotium [16] is the most popular 3rd party open-source Android testing framework, suitable for automatic black-box UI testing for Android applications. It offers APIs to directly interact with UI components based on attributes such as index, textual contents, element's name, or resource ID. Unlike UI Automator, it supports both native and hybrid application tests: Native applications are the ones which are developed for a specific platform. On the other hand, hybrid applications are combinations of native and web apps (i.e., they require the HTML to be rendered in the browser).
Robotium tests are written in the Java programming language and can be executed on a single real device or on emulator at a time. Like Espresso, Robotium is able to test only the AUT, meaning that there is no way of testing outside this application.

3.3.4 Appium

Appium [17] is a cross-platform testing framework that supports native, hybrid and mobile Web apps, both for Android and iOS mobile platforms. Appium is suitable for black-box testing, and, in the case of Android, it uses UI Automator or Instrumentation framework for running its tests, which can be executed by real devices or emulators.
Appium provides a client-server architecture where the server handles the communication with the device where the AUT is, and exposes a REST API. The web-server receives connections and test commands from a client, executes those commands on a target device, and responds with an HTTP response to inform the client about the result of the command execution. Furthermore, the client can be written in many languages (e.g., Java, JavaScript, Ruby, PHP, C#, etc.) and Appium client libraries already provide some.

3.3.5 Calabash

Like Appium, Calabash [18] is also a cross-platform framework that performs automated tests. It supports Android native applications, hybrid applications, and iOS native applications. The framework supports a screenshot function, different touch events and assertions[3], and it uses Android's Instrumentation for driving the user interface. Calabash tests can be written with Ruby, but the peculiarity of this framework lies in the possibility to write tests in Cucumber [19], where testing instructions are expressed through a natural language.

3.4 Multi-level GUI Comparison Criteria

In a model-based testing approach, the most important ability to generate effective test inputs lies in the mapping of the behavioral space of the AUT into a model. For inferring the model, model-learning techniques choose a level of behavior abstraction and define a GUI comparison criterion (GUICC) to distinguish different GUI states.
Over the years, these techniques have encountered difficulties in accurate modeling due to the fact that Android application's GUIs increasingly expose tricky dynamic behaviors to the benefit of better user experience. More specifically, a too abstract GUI model tends to represent fewer app's behaviors than those that can actually be manifested. This could lead to have poor testing effectiveness and, in the case of active learning techniques, dynamic behaviors of the GUIs could cause incoherent model generation. Conversely, a too detailed model could lead to state explosion problems.

Based on the fact that Android applications have a hierarchical structure where a package can include many Activities and each Activity can provide a hierarchical layout with multiple UI components (*widget tree*), Baek and Bae [35] designed a layered comparison model that progressively consider this hierarchy.

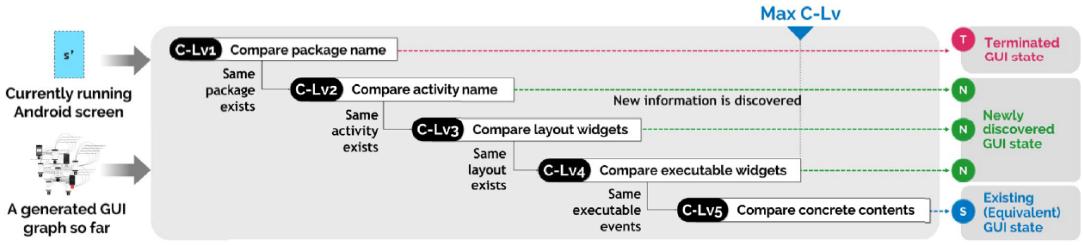


Figure 3.4: Illustration of a *multi-level GUI comparison criteria* model for Android apps [35]

[3] An assertion is a boolean expression at a specific point in the source code that is true if and only if the program is working correctly.

The comparison model is structured on 5 levels of detail (**C-Lv**) and it provides 3 types of outputs according to the comparison result: **T** (Terminated) for an out-of-the-scope GUI state, **S** (Same) for a GUI state already discovered, or **N** (New) for a GUI state previously unexplored. It is also possible to modify the maximum comparison level (**Max C-Lv**) if the tester wants to adjust the abstraction level of the GUI model.

C-Lv1: Compare Package Names

The first level criterion is based on comparing the application package, in order to avoid exploration outside of the application boundary. If the device-focused screen belongs to the AUT's package, the information on the next level (C-Lv2) will be compared, otherwise the terminated status will be assigned to the current GUI state of the application in foreground.

C-Lv2: Compare Activity Names

Since each Activity has a lifecycle that is independent of the other Activities and is implemented separately (i.e., on different source code files) from the other ones, the Activity name is a legitimate criterion (widely used in literature) to certainly distinguish between different GUI states. If the name of the Activity in the foreground does not appear among the Activity names of the already visited GUI states, then the current screen will be considered as a new GUI state. On the contrary, if the current screen belongs to an Activity already explored, next-level comparisons will be performed.

C-Lv3, C-Lv4: Compare Widget Composition

By analyzing the widget tree, extractable for example by using UI Automator, it is possible to distinguish between executable widgets and (non-executable) layout widgets: A widget is considered executable if at least one property that determine whether it can react to certain events (e.g., clickable) has "true" value.
Some UI testing tools (e.g., uiautomatorviewer) encode the relationships between the widgets by means of an **index** property which numbers the tree nodes in an orderly manner. By using index values, each widget can thus be uniquely identified as a cumulative (from the root node to the target node) index sequence (CIS).

At C-Lv3, the *layout CISs* are used as GUICC of GUI state distinction. If the current GUI state has the same set of layout CISs of a visited GUI state, then the *executable CISs* will be compared at C-Lv4 (e.g., by comparing the set of event handlers [37]). Otherwise, the current screen is considered as a new GUI state.
GUI states with different executable widgets should be characterized by a different set of triggerable events and, therefore, may lead to discover new behaviors; For this reason, a screen whit a different set of executable CISs is considered as a new GUI state.

C-Lv5: Compare Contents

A final level of comparison could be used in case it is necessary to differentiate separated contexts on screens featuring the same widget tree. For this purpose, also concrete contents (e.g., textual contents or descriptions) of each screen should be considered.

An additional need for a further comparison level may arise as a result of scroll events performed on lists of widgets (e.g., ListView[4] and GridView[5]). In this case, if the first element of the list is altered after a scroll event, it means that some previously non-visible child items become visible (and then triggerable).

[4] Android ListView is a ViewGroup, that is a special View (basic building block for UI components) that can contain other Views, which contains several items and displays them in a vertically-scrollable list.

[5] Android GridView is a ViewGroup that can contains items and displays them in a two-dimensional scrollable grid.

However, such an accurate comparison criterion rarely works as expected, as, due to the complexity of today's applications, it might run into the state explosion problem during execution.

4 Method

4.1 An Active Learning Approach

In this section, we consider the problem of automatically testing Android apps considered as black boxes and for which we do not have an existing model of the GUI. The goal is to explore as many behaviors of the AUT as possible by learning at runtime a GUI model of the app. For this purpose, we propose a novel active learning algorithm designed for limiting redundant actions and minimizing the number of restarts of the AUT in order to reduce the computational cost of the algorithm (fewer state transitions to cross) and increase the implementation efficiency (as restarting is a significantly expensive operation).

4.1.1 E-LTS Model

An intuitive way to describe event-driven systems is to consider that the system transits from one *state* to another depending on the *action* performed by the environment or by the system itself. A transition system [45] is a conceptual model consisting of states connected by *transitions* that can be represented as a directed graph where vertices are the states of the system and edges represent transitions between the states. In a labeled transition system (LTS), each transition is labeled with the action that triggers the state change.

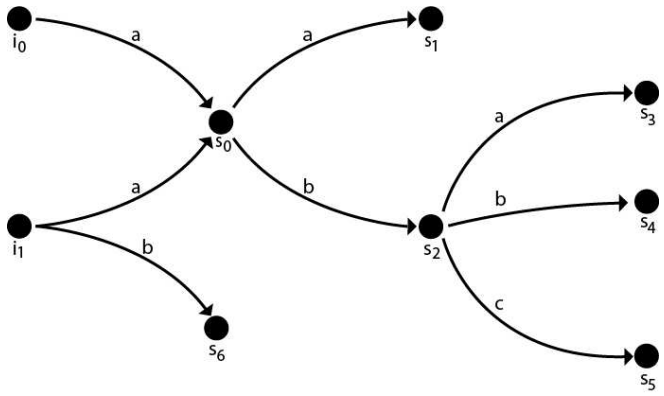


Figure 4.1: Multiroot deterministic labeled transition system representation

Inspired by Choi, Necula, and Sen [37], but by considering also state labeling (which specifies the values of some variables in the state), we defined an extended version of a deterministic[1] labeled transition system (E-LTS) to model an app's GUI. Each E-LTS's state is then labeled with:

1. An immutable set of available inputs (or actions);
2. An editable set of inputs (or actions) considered previously triggered;
3. The first transition that led to it (excluding initial states).

Formally, an E-LTS M is a 7-uple

$$M = (S, A, R, I, \sigma, \delta, \lambda)$$

where

- S is a finite set of states;
- A is a finite set (alphabet) of inputs (or actions);
- $R \subseteq S \times A \times S$ is the transition relation: Given a transition $(s, a, s') \in R$ (denoted by $s \xrightarrow{a} s'$), $s \in S$ is called the *source*, $s' \in S$ the *target* and $a \in A$ the label of the transition;
- $I \subseteq S$ is the set of initial states;
- $\sigma : S \to R$ is a state labeling function: Given a state $s \in S$, $\sigma(s)$ represents the *first transition* that led to the state s. A transition is defined as the first transition if it is learned at the same time that its target state is learned;
- $\delta : S \to \mathcal{P}(A)$ is a state labeling function representing the set of inputs that are available at state s. $\mathcal{P}(A)$ denotes the power set of A;
- $\lambda : S \to \mathcal{P}(A)$ is a state labeling function representing the set of inputs considered previously triggered on state s: for any $s \in S$ and $a \in A$, if there exists a $s' \in S$ such that $(s, a, s') \in R$, then $a \in \lambda(s)$.

  N.B. If exists a $s \in S$ such that $\delta(s) \neq \lambda(s)$, then the model is incomplete.

4.1.2 The Learning Algorithm

Assumptions

Suppose that it is possible to compare the state of the system with states that are already part of the learned model (based on a feasible comparison criterion).
Suppose that it is possible to inspect the state of the system to determine the set of available inputs.
Suppose that it is possible to send inputs to the system and waiting for the system to become stable after performing an action.
Suppose that it is possible to bring the system under learning (SUL) back to an initial state through the $\rho \notin A$ action.
Suppose that, starting from any state $s \in S \backslash I$, it is possible to bring the SUL back to some previous (i.e., an ancestor) state through the $\beta \notin A$ action.
To ensure the algorithm's determinism, suppose that each action $a \in A$ has different priority over each other so that, given a subset of actions $E \subseteq A$, the $Pop(E)$ operation returns the action with the highest priority.

[1] An LTS is deterministic if $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$ imply that $s' = s''$, for all $s \in S$ and for all $a \in A$.

Description of the algorithm

The *Algorithm 4.1* (described in pseudo-code) maintains five local variables:

1. s denotes the state considered in the current iteration, instead s_0 represents the current state at any time
2. p denotes the state considered in the previous iteration
3. a denotes the last action performed
4. r denotes whether or not a restart operation (i.e., the ρ action) has been performed in the previous iteration
5. M denotes the E-LTS model learned so far

Algorithm 4.1: E-LTS learning

```
1  M ← (({s0}, A, ∅, {s0}, {s0 ↦ σ(s0)}, {s0 ↦ δ(s0)}, {s0 ↦ λ(s0)}))   //s0 is the current (fresh) state
2
3  r ← false                                                            //r is a boolean flag
4  s ← null                                                             //s is the current state
5  a ← null                                                             //a is the last action performed
6
7  while STOP_CONDITION do
8   p ← s                                                               //Save previous state
9   s ← s0                                                              //Retrieve current state
10
11   if s ∉ S then
12    add s to S                                                        //Add a new state to M
13   end if
14
15   if r is true then
16    r ← false
17    if s ∉ I then
18     add s to I                                                      //Add a new initial state to M
19    end if
20   else
21    if p is not null then
22     if p --a--> s ∉ R then
23      add p --a--> s to R                                           //Add a transition relation to M
24     end if
25    end if
26   end if
27
28   if δ(s) ≠ ∅ ∧ δ(s) ≠ λ(s) then                                     //Not fully explored
29    a ← Pop(δ(s)\λ(s))
30    EXECUTE(s, a)                                                   //Perform an action a ∈ δ(s)\λ(s) on state s
31   end if
32
33   if δ(s) ≠ ∅ ∧ δ(s) ≠ λ(s) then                                     //Still not fully explored
34    if s ∉ I then
35     UNLOCK(σ(s))                                                  //"Pave the first way" towards s
36    end if
37   else                                                            //Fully explored state
38    if s ∈ I then
39     if s0 ∉ S ∨ δ(s0)\λ(s0) ≠ ∅ then                               //New or not fully explored current state s0
40      remove a from λ(s)                                           //Remove a from the set of inputs triggered on s
41     else
42      return M                                                    //End
43     end if
44    else                                                          //New current state s0
45     if s0 ≠ s then                                               //State unchanged
46      if s0 ∈ S ∧ δ(s0) = λ(s0) then                              //Reached old and fully explored state s0
47       EXECUTE(s0, ρ)                                            //Restart
48       r ← true
49      end if
50     else
51      EXECUTE(s0, β)                                            //Go back
52     end if
53    end if
54   end if
55  end while
56
57  return M
```

```
1  procedure UNLOCK(s_i --a--> s_f)
2    if s_i ∈ I then
3      remove a from λ(s_i)                    //Remove a from the set of inputs triggered on s_i
4      return                                  //An initial state has no incoming transitions
5    end if
6
7    UNLOCK(σ(s_i))                            //Recursion on the first transition towards s_i
8    remove a from λ(s_i)                      //Remove a from the set of inputs triggered on s_i
9  end procedure
```

The E-LTS learning algorithm restarts the AUT sparingly compared to other standard learning algorithms such as *L**: It attempts to extend the current execution path by performing an action that is available but not yet triggered on the current state or by using an appropriate feature (the BACK button in Android devices) to go back to a previous (though not well defined) state.
Starting from an installation from scratch, the first state (as well as the first initial state) is discovered as soon as the system (the AUT in our case) reaches the steady state after the first boot (*line 1*). After the first boot, a state is recognized to be an initial state if and only if the last action performed has been a restart action (*lines 15-18*).
At each state, the algorithm stores (if necessary) information about new states (both initial and non-initial) or new transitions (*lines 12, 18 and 23*).
After updating the model, the algorithm verifies if some action can still be executed on the current state (*line 28*). If there is any action still available to be executed, the algorithm selects and performs the one with the highest priority (*lines 29-30*). It is important that actions are executed respecting a certain order to ensure a proper **depth-first search** strategy.
If no action has been performed on the current state, the algorithm terminates its execution if the current state is identified as being an initial state (i.e., it belongs to *I*) in which it is no longer possible to trigger events (*line 42*), otherwise a restart action is performed (*line 47*).
Conversely, if an action has been executed, the algorithm checks if the state s on which it has been performed still has actions available for execution (*line 33*). In this case, a recursive function ensures that all those actions that have led to the state s from the initial state are made available for next iterations (*line 35*). In particular, for each first transition $s_i \xrightarrow{a} s_{i+1}$, starting from the s state back to the initial state, the algorithm will exclude from the set of actions already performed on each intermediate state the action a that triggered the state change between s_i and s_{i+1}.
Instead, if after the action execution, the state s is fully explored (all available actions have been triggered), it is possible that:

1. the state s is an initial state: In that case, if the previous action caused a state change and the current state s_0 is new or not fully explored then will re-enable the a action just executed on state s (*line 40*) . Otherwise, the algorithm terminates (*line 42*).

2. the state s is not an initial state and the action caused a state change: If the reached (current) state s_0 is fully explored then a restart action is performed (*line 47*). Otherwise, the current state will be considered in the next iteration.

3. the state s is not an initial state and it remains unchanged: In this case the algorithm go back to a previous (ancestor) state (*line 51*).

4.2 Implementation

In this section we introduce a GUI Ripping-based implementation of a novel non-invasive automatic black-box testing technique for Android mobile applications.

4.2.1 Choice of the testing framework

Among several technologies that can be used in black-box testing, we have decided to use UI Automator (see ***Section*** 2.4) for our implementation.
The main features that led us to choose it are the following:

- It is cross-application package;
- It is smoothly integrated with Android Studio [20], which is the official IDE (Integrated Development Environment) for Android;
- It can be used on devices with different display resolution;
- Events are directly linked with Android UI components instead of relying on coordinate locations;
- It can use device-level actions;
- It allows to collect performance metrics.

The major limitation in UI Automator is that it is supported only from Android API level 18. This implies that tests can only be run on devices running Android 4.3 or higher. Despite this being a limitation, Google's official data, updated on 8 August 2017, indicates that the percentage of devices that can be actually considered for our tests is 92.1%, which makes our implementation valid in most cases.

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.7%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.7%
4.1.x	Jelly Bean	16	2.7%
4.2.x		17	3.8%
4.3		18	1.1%
4.4	KitKat	19	16.0%
5.0	Lollipop	21	7.4%
5.1		22	21.8%
6.0	Marshmallow	23	32.3%
7.0	Nougat	24	12.3%
7.1		25	1.2%

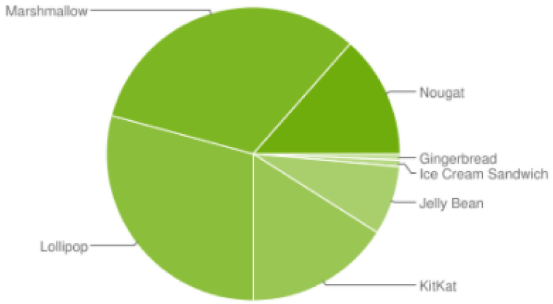


Figure 4.2: Relative number of devices running a given version of the Android platform [27]

4.2.2 Implementation details

We have implemented the E-LTS learning algorithm for Android apps in a tool written in Java called *PoLiUToDroid*. Since developed as an *Instrumented Unit Test* (see ***Section*** 2.3), PoLiUToDroid can test Android apps running both on emulators and on physical Android devices connected to the development/testing machine through ADB (Android Debug Bridge)[2].

The tool is able to inspect all the UI components currently visible on the device screen and interact with them even if they do not belong to the application under test without relying on source code or bytecode instrumentation.
Both the GUI ripping operation and event injection phases are non-invasive because they are based on UiAutomation, UI Automator framework, and existing Android debugging/testing utilities, which are embedded within Android. The information gathered from the connected device can be classified into three sets:

1. **GUI information.** For each UI, PoLiUToDroid takes the screenshot and records the UI hierarchy dumped using an its own module based on UiAutomation;

2. **Process information.** PoLiUToDroid can access the app-level process status by using the *dumpsys*[3] tool in Android.

3. **Logs.** Logs include the actions performed by each test input, the traces produced by the AUT and screen-based CPU/memory performance log. They can be retrieved from the Android profiling tool and *logcat*[4].

PoLiUToDroid implements a model-learning technique, that is, by starting only with the name of the application package to be tested, it is capable of on-the-fly learning a model that in the meantime it exploits to visit the AUT itself. The model is a state transition graph (more precisely an E-LTS), in which each node represents a different GUI state, and each edge between two nodes represents the test input event that triggered the state transition. A state node contains GUI information and state variables (e.g., already executed user inputs), and an event edge contains details about the start (*source*) and the arrival (*target*) GUI state, the widget on which the test input has been injected, and the performed action.
PoLiUToDroid retains the information of all the visited GUI states in a graph data structure (*GUI Graph*) where each vertex is a *GUI State* object, and each edge is *GUI Event* object.
For exploring the application under test PoLiUToDroid follows a depth-first search approach on the dynamically built GUI Graph. In particular, the algorithm implemented by PoLiUToDroid allows to avoid redundant test input choices and to minimize the number of restarts of the AUT.

The principle we use to distinguish between different GUI States is based on the Multi-level GUI Comparison Criteria (see ***Section*** 3.4) and implemented through a GUI State *signature* (made as in ***Code*** 4.2). It allows to achieve an appropriate (user-configurable) abstraction level of the GUI model in order to avoid both to represent fewer app's behaviors than those that can actually be manifested and to incur to state explosion problems.
We extended the multi-level comparison model to handle the cases where the AUT delegates to other apps the task of certain actions (e.g., taking photos by camera), making the implementation capable of acting on apps with different packages than the one of the test's target application. In this regard, relying on a configurable parameter (i.e., the maximum depth of the graph reachable outside the target package), PoLiUToDroid restricts the exploration of apps that do not belong to the AUT's package.

[2] *adb* is a command-line tool, delivered with Android SDK, that provides communication between a development machine and an Android device.
[3] *dumpsys* is a tool that runs on Android devices and provides diagnostic information about system services.
[4] *logcat* is a command-line tool able to dump the internal log of the Android operating system, including application traces.

In the following (simplified) Java code snippet 4.2, it is shown how, depending on the maximum comparison level considered (*line 4*), each GUI State build its signature (*line 32*) by concatenating (from the highest to lowest level) the information related to:

5: textual and descriptive contents related to the widgets that are in the layout[5],

4: the executable features of each executable widget (*lines 5-12*),

3: the layout widget CISs (*lines 13-20*),

2: the name of the Activity (*lines 24-26*),

1: and the package to which the GUI State belongs (*lines 27-29*).

Code 4.2: Multi-level GUI comparison criteria implemented as signature

```
1  private void parseSignature() {
2      StringBuilder stringBuilder = new StringBuilder("");
3
4      switch (Config.sComparisonLevel) {
5
6          case 5:
7              // Contents comparison level
8
9              for (GuiWidget guiWidget : executableWidgets) {
10                 String text = getTextFromWidget(guiWidget);
11                 String desc = getContentDescriptionFromWidget(guiWidget);
12
13                 stringBuilder.append(text).append(desc);
14             }
15
16         case 4:
17             // Executable widget comparison level
18
19             String execSignature = "";
20             // Concat executable property values
21             for (GuiWidget guiWidget : executableWidgets)
22                 execSignature += guiWidget.getExecPropertyValue();
23
24             stringBuilder.insert(0, ":" + execSignature);
25
26         case 3:
27             // Layout widget comparison level
28
29             stringBuilder.insert(0, ":" + layoutCIS.toString());
30
31         case 2:
32             // Activity activityName comparison level
33
34             stringBuilder.insert(0, ":" + activityName);
35
36         case 1:
37             // Package comparison level
38
39             stringBuilder.insert(0, pkg);
40     }
41
42     signature = stringBuilder.toString();
43 }
```

Each GUI State is essentially a detailed widget tree (GUI State-related hierarchical layout representation) where each widget is uniquely distinguished within a GUI State by its CIS. PoLiUToDroid is able to test the widgets by click, long click and check actions in order to explore the app; In future work, we would like to implement scroll, swipe, and screen rotation actions as well.

Each widget is characterized by a set of actions that are available on it (the δ function 4.1.1) and another set that represents the inputs that have not yet been tested on it (the λ function 4.1.1).

[5]For reasons of efficiency, we decided to consider only the contents of the executable widgets.

We decided to implement these features by using a method similar to the mechanism used by Unix systems to determine its file system permissions. In particular, each executable action is represented by a distinct number such that each mixed sum represents, without ambiguity, a specific set of actions. Hence, when added together for defining δ and λ, it is always possible to exactly determine which actions have to be considered on the widget.

Table 4.1: Widget executable values

Action	Value
CLICK	1
LONG CLICK	2
CHECK	4

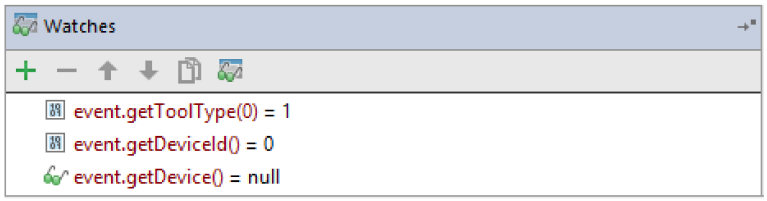
Furthermore, all GUI components currently visible on the device screen that have type *EditText* (i.e., textual input fields) will be randomly filled with predefined strings.
The BACK button will only be used in the cases provided by the algorithm. Instead, the launch of the AUT will be performed in cases where the algorithm expects to perform the restart operation either when the screen in the foreground is the *launcher* (e.g., when the BACK button exits the application).
After each input injection, PoLiUToDroid exploits the UI Automator APIs for waiting until the app reaches a stable state and monitors the state changes and then updates the model. If the GUI state is changed, it adds a GUI Event edge and the new GUI State to the GUI Graph; Otherwise, it updates the state variables of the involved GUI states.

Motivated by the concepts outlined in the ***Section*** 3.2, we implemented the input sending so that the exploration, however automatic, is as much human-like as possible (to avoid being detected by malware). In this regard, we perform tap touch operations on random coordinates within the target widget's surface and wait a random, but reasonable, timeout between one action and the other. Moreover, each widget will be tagged as *NAF* (Not Accessibility Friendly) if it is not "humanly accessible" (e.g., as tiny). In addition to check if the widget presents text or content description (as done by UI Automator framework), PoLiUToDroid checks whether the widget has a higher size than a threshold. In particular, a widget is considered NAF if it has a surface that is less than a certain percentage of the entire screen.

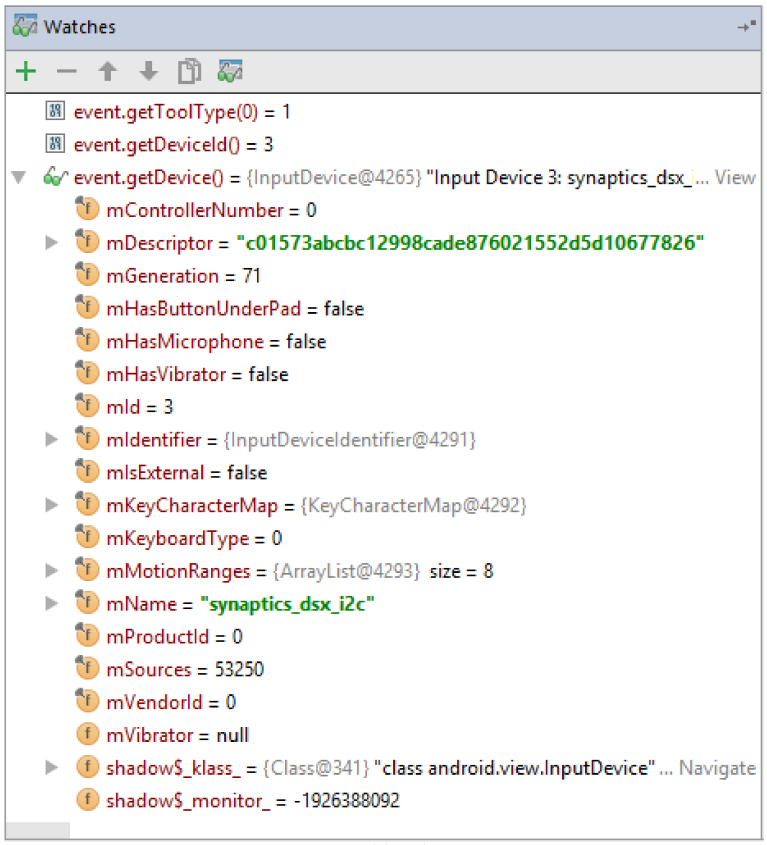
Code 4.3: Function for determining the human visibility of a widget

```
1  private boolean isHumanVisible(GuiWidget guiWidget) {
2
3      // Rectangle characterizing the widget's surface
4      Rect rect = guiWidget.getVisibleBounds();
5
6      // Pixel area of the rectangle (i.e., of the widget) on the screen
7      int pixelWidgetSurface = rect.width() * rect.height();
8
9      // Percentage screen widget's occupancy
10     float widgetOccupancy = pixelWidgetSurface / (float) Config.sDisplayResolution;
11
12     // Check if the widget's surface is greater or equal to a visibility threshold
13     if (widgetOccupancy >= Config.sMinWidgetSurfacePercentage) {
14         // Humanly visible
15         return true;
16     } else {
17         // Not humanly visible
18         return false;
19     }
20 }
```

Finally, PoLiUToDroid is able to generate event objects containing non-dummy values (e.g., the authentic *DeviceId* of the device under test). For this purpose, we developed a module based on UiAutomation responsible for injecting user events such as touch events and text key events into the system.
Figures 4.3 and 4.4 report the values of the variables, considered by Diao et al. [40] to distinguish user manual inputs from the simulated ones, associated with objects created by the standard UI Automator APIs and those generated by PoLiUToDroid. By comparing these values with those in the *Tables* 3.1 and 3.2, we can notice that events generated by using PoLiUToDroid are more properly attributable to a real activity rather than a simulation.

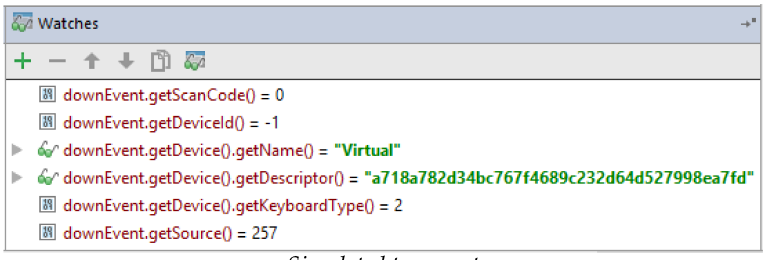


Simulated hardware action

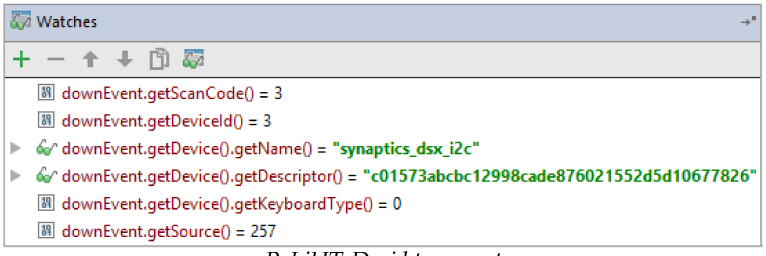


PoLiUToDroid hardware action

Figure 4.3: MotionEvent: standard APIs vs PoLiUToDroid APIs



Simulated tap event



PoLiUToDroid tap event

Figure 4.4: KeyEvent: standard APIs vs PoLiUToDroid APIs

PoLiUToDroid is also extremely configurable. Some of the most important parameters that can be set by the user are:

- **target**: name of the application package to be tested.
- **max-steps**: it limits the number of iterations of the algorithm.
- **max-depth**: it limits the depth that the algorithm will reach when exploring the AUT.
- **max-foreign-depth**: it limits the depth that the algorithm will reach in the exploration of auxiliary applications (i.e., not AUT).
- **max-runtime**: it defines the maximum testing time.
- **comparison-level**: it sets the comparison level to use in the multi-level comparison criteria.
- **min-sleep-action**: it sets the minimum waiting time between two consecutive actions.
- **max-sleep-action**: it sets the maximum waiting time between two consecutive actions.
- **min-widget-percentage**: it sets the percentage occupancy of the screen below which a widget is considered NAF.
- **test-naf**: if true, also widgets tagged as NAF will be tested.

5 Results

In this chapter we present results obtained from tests performed on a randomly selected sample of 29 applications (those which have had fewer problems caused by the instrumentation) among those in the *AndroCoverage Dataset* [21]. These applications show a large variety of different behaviors and are used to compare the performance (in terms of efficiency and efficacy) of different automated testing tools (PoLiUToDroid, Monkey and DroidBot) on the same ground. We decided to consider also DroidBot as it is a recent project with objectives similar to those of this thesis and also because it did not cause too much trouble during test of the instrumented applications.

For an objective comparison, we have instrumented the apps with *Emma* [22] in order to retrieve information about the code coverage of each AUT. Although PoLiUToDroid does not need to instrument the AUT to accomplish its work, this operation has been necessary to obtain code coverage rates; The instrumentation process we follow is similar to that described by Zhauniarovich et al. [65] and depicted in the *Figures* 5.1 and 5.2.

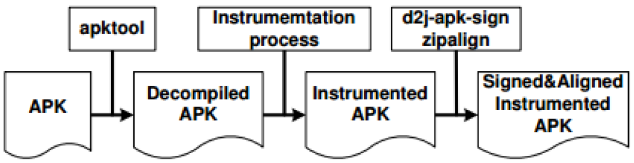


Figure 5.1: Instrumentation workflow [65]

For the analysis of the results, we focus mainly on the coverage of *basic blocks* (i.e., an uninterrupted or continuous section of instructions). The *basic block coverage* for an application is the number of unique basic blocks executed at runtime divided by the number of unique basic blocks present in the source code.

Testing tools could generate exceptions with the instrumentation, making it impossible to retrieve the coverage rate for some applications: We reported these cases as *N/A* in our results. However, a global quality index can still be given by the *average coverage of basic blocks*, that is the sum of the basic block coverage of all applications divided by the number of applications.

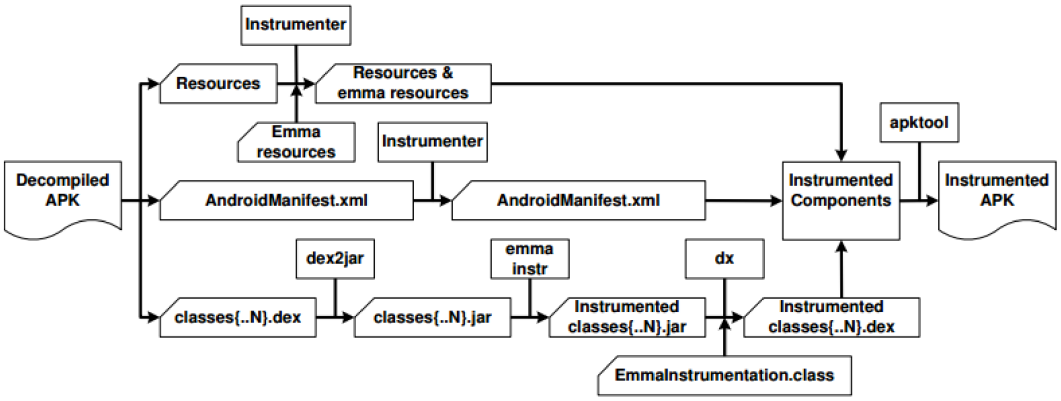


Figure 5.2: Instrumentation process [65]

The following sections group the results obtained so that the reader can have a better awareness of the context of the results and is able to refer directly to them when these will be discussed in the next chapter.

All tests were conducted under the configurations set out in ***Appendix A.1***.

5.1 Overall code coverage statistics

In this section has been reported all code coverage data grouped by tool and testing constraints used for each experiment. For each Android application, the tables report "Coverage" fields (respectively class, method and block coverage) which give information about the coverage achieved by the considered tool, "Time" that represents the test execution time or "Event count" which represents the number of actions triggered in the benchmark.

For comparing the coverage of different tools, we have considered different comparison metrics and different scenarios.
In the first testing sessions (set of tests characterized by the same constraints executed on different apps by the same tool), we limit the execution time of the tools by using the time that PoLiUToDroid spent to test the same app. In this situation, we are able to calculate the coverage/time ratio, which gives information about the effectiveness of the testing tool.
In another experiment, we limit the execution time by using the maximum time spent by PoLiUToDroid among all its tests. In this situation we have given to each tool the chance to achieve maximum results.
In the last testing session, only for Monkey, we limited the event count by using the number of events generated by PoLiUToDroid to test the same app, so to emphasize the effectiveness of a systematic technique compared to a random one.

Table 5.1: PoLiUToDroid: code coverage

Package AUT	Time (s)	Class coverage (%)	Class coverage	Method coverage (%)	Method coverage	Block coverage (%)	Block coverage
ch.hgdev.toposuite	2197	18%	101/573	12%	419/3609	10%	6952/72032
com.achep.acdisplay	176	39%	252/639	28%	848/3038	25%	20472/82898
com.gladis.tictactoe	7	26%	36/141	18%	157/896	9%	3136/36434
com.glanznig.beepme	187	29%	44/152	24%	227/929	21%	6374/30203
com.khuttun.notificationnotes	96	41%	12/29	65%	43/66	35%	1037/2926
com.luk.timetable2	129	7%	41/583	3%	129/5006	2%	2473/116559
com.pindroid	59	17%	35/204	11%	107/970	8%	2012/24161
com.secuso.privacyFriendlyCodeScanner	89	30%	123/409	22%	522/2353	12%	17202/145921
com.secuso.torchlight2	31	15%	3/20	37%	16/43	17%	264/1547
com.yasfa.views	1589	22%	36/165	17%	104/628	18%	6199/34283
cx.hell.android.pdfview	297	16%	11/67	12%	49/395	16%	2061/12765
de.markusfisch.android.shadereditor	781	72%	72/100	67%	438/649	60%	7705/12908
fr.mobdev.goblim	1088	39%	22/57	35%	56/160	19%	935/4903
fr.xtof54.scrabble	579	39%	13/33	42%	58/139	76%	10535/13904
jp.takke.cpustats	119	81%	17/21	79%	89/112	30%	2142/7131
me.anuraag.grader	10	19%	10/52	11%	19/174	10%	381/3762
net.olejon.spotcommander	626	29%	65/225	23%	265/1171	24%	7267/29708
net.sourceforge.opencamera	858	48%	80/166	41%	555/1342	37%	15922/42936
org.androidsoft.games.memory.tux	139	48%	21/44	75%	161/214	76%	2388/3157
org.mumod.android	237	15%	45/306	9%	164/1761	6%	2914/48276
org.openintents.notepad	253	48%	50/104	46%	213/465	40%	5215/13041
org.petero.droidfish	421	48%	181/377	47%	985/2118	57%	56715/100020
org.pyneo.maps	197	31%	113/363	26%	544/2131	24%	15987/66423
org.scoutant.blokish	809	41%	24/58	39%	93/241	40%	3571/8986
org.tmurakam.presentationtimer	93	100%	11/11	88%	66/75	86%	1262/1463
org.zamedev.gloomydungeons1hardcore.opensource	45	82%	90/110	67%	402/596	63%	18476/29199
ru.valle.btc	1126	14%	135/992	7%	507/6853	7%	20675/285754
se.tube42.drum.android	8	69%	191/276	48%	824/1730	38%	19819/52636
se.tube42.kidsmem.android	7	67%	186/277	46%	795/1732	36%	18999/53340

Table 5.2: DroidBot: code coverage (with equal testing time of *Table 5.1*)

Package AUT	Time (s)	Class coverage (%)	Class coverage	Method coverage (%)	Method coverage	Block coverage (%)	Block coverage
ch.hgdev.toposuite	2197	13%	75/573	8%	297/3609	10%	7530/72032
com.achep.acdisplay	176	31%	196/639	20%	599/3038	18%	15165/82898
com.gladis.tictactoe	7	26%	36/141	18%	158/896	9%	3152/36434
com.glanznig.beepme	187	29%	44/152	23%	216/929	20%	6058/30203
com.khuttun.notificationnotes	96	31%	9/29	48%	32/66	28%	815/2926
com.luk.timetable2	129	7%	41/583	2%	109/5006	2%	2144/116559
com.pindroid	59	15%	30/204	9%	92/970	7%	1785/24161
com.secuso.privacyFriendlyCodeScanner	89	30%	123/409	23%	537/2353	13%	18753/145921
com.secuso.torchlight2	31	15%	3/20	37%	16/43	17%	260/1547
com.yasfa.views	1589	44%	73/165	34%	214/628	29%	9921/34283
cx.hell.android.pdfview	297	16%	11/67	10%	41/395	15%	1913/12765
de.markusfisch.android.shadereditor	781	49%	49/100	46%	298/649	41%	5336/12908
fr.mobdev.goblim	1088	39%	22/57	38%	60/160	21%	1031/4903
fr.xtof54.scrabble	579	N/A	N/A	N/A	N/A	N/A	N/A
jp.takke.cpustats	119	71%	15/21	64%	72/112	23%	1651/7131
me.anuraag.grader	10	0%	0/52	0%	0/174	0%	0/3762
net.olejon.spotcommander	626	29%	66/225	23%	273/1171	27%	7985/29708
net.sourceforge.opencamera	858	34%	57/166	36%	489/1342	31%	13186/42936
org.androidsoft.games.memory.tux	139	48%	21/44	75%	161/214	76%	2387/3157
org.mumod.android	237	11%	34/306	6%	114/1761	4%	1847/48276
org.openintents.notepad	253	50%	52/104	46%	213/465	42%	5530/13041
org.petero.droidfish	421	39%	147/377	35%	740/2118	27%	27205/100020
org.pyneo.maps	197	23%	85/363	18%	384/2131	17%	11331/66423
org.scoutant.blokish	809	41%	24/58	37%	88/241	37%	3341/8986
org.tmurakam.presentationtimer	93	82%	9/11	61%	46/75	71%	1039/1463
org.zamedev.gloomydungeons1hardcore.opensource	45	79%	87/110	66%	392/596	63%	18332/29199
ru.valle.btc	1126	N/A	N/A	N/A	N/A	N/A	N/A
se.tube42.drum.android	8	70%	192/276	50%	865/1730	39%	20619/52636
se.tube42.kidsmem.android	7	0%	0/277	0%	0/1732	0%	0/53340

Table 5.3: DroidBot: code coverage (with max testing time of *Table 5.1*)

Package AUT	Time (s)	Class coverage (%)	Class coverage	Method coverage (%)	Method coverage	Block coverage (%)	Block coverage
ch.hgdev.toposuite	2197	16%	90/573	10%	345/3609	11%	7948/72032
com.achep.acdisplay	2197	41%	263/639	28%	865/3038	25%	20690/82898
com.gladis.tictactoe	2197	N/A	N/A	N/A	N/A	N/A	N/A
com.glanznig.beepme	2197	N/A	N/A	N/A	N/A	N/A	N/A
com.khuttun.notificationnotes	2197	41%	12/29	70%	46/66	37%	1088/2926
com.luk.timetable2	2197	7%	41/583	2%	113/5006	2%	2262/116559
com.pindroid	2197	N/A	N/A	N/A	N/A	N/A	N/A
com.secuso.privacyFriendlyCodeScanner	2197	30%	124/409	23%	541/2353	13%	18589/145921
com.secuso.torchlight2	2197	15%	43891	40%	17/43	18%	275/1547
com.yasfa.views	2197	44%	72/165	34%	213/628	29%	9899/34283
cx.hell.android.pdfview	2197	N/A	N/A	N/A	N/A	N/A	N/A
de.markusfisch.android.shadereditor	2197	N/A	N/A	N/A	N/A	N/A	N/A
fr.mobdev.goblim	2197	39%	22/57	36%	58/160	19%	921/4903
fr.xtof54.scrabble	2197	N/A	N/A	N/A	N/A	N/A	N/A
jp.takke.cpustats	2197	81%	17/21	79%	89/112	31%	2231/7131
me.anuraag.grader	2197	31%	16/52	21%	37/174	17%	651/3762
net.olejon.spotcommander	2197	29%	66/225	23%	273/1171	27%	7985/29708
net.sourceforge.opencamera	2197	41%	68/166	40%	541/1342	33%	14303/42936
org.androidsoft.games.memory.tux	2197	45%	20/44	72%	155/214	73%	2301/3157
org.mumod.android	2197	N/A	N/A	N/A	N/A	N/A	N/A
org.openintents.notepad	2197	N/A	N/A	N/A	N/A	N/A	N/A
org.petero.droidfish	2197	38%	143/377	34%	713/2118	26%	26233/100020
org.pyneo.maps	2197	N/A	N/A	N/A	N/A	N/A	N/A
org.scoutant.blokish	2197	48%	28/58	39%	94/241	38%	3417/8986
org.tmurakam.presentationtimer	2197	82%	9/11	63%	47/75	74%	1088/1463
org.zamedev.gloomydungeons1hardcore.opensource	2197	81%	89/110	68%	404/596	64%	18617/29199
ru.valle.btc	2197	N/A	N/A	N/A	N/A	N/A	N/A
se.tube42.drum.android	2197	N/A	N/A	N/A	N/A	N/A	N/A
se.tube42.kidsmem.android	2197	N/A	N/A	N/A	N/A	N/A	N/A

Table 5.4: Monkey: code coverage (with equal event count of *Table 5.1*)

Package AUT	Event count	Class coverage (%)	Class coverage	Method coverage (%)	Method coverage	Block coverage (%)	Block coverage
ch.hgdev.toposuite	999	3%	20/573	2%	59/3609	2%	1392/72032
com.achep.acdisplay	141	24%	152/639	14%	422/3038	12%	9709/82898
com.gladis.tictactoe	N/A	N/A	N/A	N/A	N/A	N/A	N/A
com.glanznig.beepme	101	12%	19/152	9%	82/929	5%	1658/30203
com.khuttun.notificationnotes	57	17%	5/29	24%	16/66	8%	232/2926
com.luk.timetable2	92	2%	11/583	1%	40/5006	1%	656/116559
com.pindroid	23	15%	30/204	9%	90/970	7%	1757/24161
com.secuso.privacyFriendlyCodeScanner	51	1%	3/409	0%	10/2353	0%	160/145921
com.secuso.torchlight2	14	10%	2/20	14%	6/43	9%	141/1547
com.yasfa.views	392	26%	43/165	15%	92/628	14%	4955/34283
cx.hell.android.pdfview	216	13%	9/67	9%	37/395	14%	1793/12765
de.markusfisch.android.shadereditor	415	29%	29/100	30%	196/649	28%	3649/12908
fr.mobdev.goblim	999	39%	22/57	31%	49/160	18%	865/4903
fr.xtof54.scrabble	263	36%	12/33	35%	49/139	75%	10461/13904
jp.takke.cpustats	65	62%	13/21	43%	48/112	12%	873/7131
me.anuraag.grader	N/A	N/A	N/A	N/A	N/A	N/A	N/A
net.olejon.spotcommander	222	15%	33/225	9%	101/1171	5%	1464/29708
net.sourceforge.opencamera	686	28%	47/166	29%	386/1342	25%	10634/42936
org.androidsoft.games.memory.tux	66	23%	10/44	23%	50/214	26%	814/3157
org.mumod.android	114	10%	32/306	6%	98/1761	4%	1723/48276
org.openintents.notepad	172	16%	17/104	7%	31/465	4%	535/13041
org.petero.droidfish	140	37%	140/377	32%	671/2118	24%	23979/100020
org.pyneo.maps	142	23%	84/363	17%	357/2131	15%	10212/66423
org.scoutant.blokish	289	29%	17/58	28%	67/241	35%	3187/8986
org.tmurakam.presentationtimer	50	45%	5/11	32%	24/75	41%	600/1463
org.zamedev.gloomydungeons1hardcore.opensource	26	27%	30/110	11%	66/596	5%	1485/29199
ru.valle.btc	426	0%	0/992	0%	0/6853	0%	0/285754
se.tube42.drum.android	N/A	N/A	N/A	N/A	N/A	N/A	N/A
se.tube42.kidsmem.android	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 5.5: Monkey: code coverage (with equal testing time of *Table 5.1*)

Package AUT	Event count	Class coverage (%)	Class coverage	Method coverage (%)	Method coverage	Block coverage (%)	Block coverage
ch.hgdev.toposuite	43960	12%	70/573	7%	262/3609	9%	6766/72032
com.achep.acdisplay	1920	39%	250/639	28%	838/3038	25%	20644/82898
com.gladis.tictactoe	2580	0%	0/141	0%	0/896	0%	0/36434
com.glanznig.beepme	15620	19%	29/152	14%	128/929	8%	2275/30203
com.khuttun.notificationnotes	21760	17%	5/29	26%	17/66	9%	251/2926
com.luk.timetable2	12540	7%	40/583	2%	99/5006	2%	2050/116559
com.pindroid	8440	17%	35/204	11%	106/970	8%	1974/24161
com.secuso.privacyFriendlyCodeScanner	16200	30%	123/409	23%	543/2353	13%	18892/145921
com.secuso.torchlight2	22520	15%	3/20	40%	17/43	18%	274/1547
com.yasfa.views	3540	N/A	N/A	N/A	N/A	N/A	N/A
cx.hell.android.pdfview	160	13%	9/67	11%	43/395	15%	1917/12765
de.markusfisch.android.shadereditor	3740	48%	48/100	44%	287/649	42%	5366/12908
fr.mobdev.goblim	1180	39%	22/57	38%	60/160	20%	982/4903
fr.xtof54.scrabble	1800	N/A	N/A	N/A	N/A	N/A	N/A
jp.takke.cpustats	640	81%	17/21	75%	84/112	29%	2096/7131
me.anuraag.grader	31800	21%	11/52	12%	21/174	12%	464/3762
net.olejon.spotcommander	5940	29%	65/225	20%	238/1171	22%	6634/29708
net.sourceforge.opencamera	11600	30%	50/166	31%	414/1342	29%	12272/42936
org.androidsoft.games.memory.tux	2380	45%	20/44	73%	156/214	73%	2298/3157
org.mumod.android	220	12%	38/306	7%	129/1761	5%	2182/48276
org.openintents.notepad	17180	38%	39/104	32%	149/465	31%	4022/13041
org.petero.droidfish	2780	44%	164/377	38%	813/2118	29%	29224/100020
org.pyneo.maps	4740	27%	97/363	22%	473/2131	20%	12982/66423
org.scoutant.blokish	5060	41%	24/58	43%	103/241	45%	4069/8986
org.tmurakam.presentationtimer	3940	91%	10/11	77%	58/75	78%	1146/1463
org.zamedev.gloomydungeons1hardcore.opensource	1860	79%	87/110	66%	395/596	65%	19036/29199
ru.valle.btc	920	N/A	N/A	N/A	N/A	N/A	N/A
se.tube42.drum.android	160	70%	193/276	49%	855/1730	40%	21074/52636
se.tube42.kidsmem.android	160	N/A	N/A	N/A	N/A	N/A	N/A

Table 5.6: Monkey: code coverage (with max testing time of *Table 5.1*)

Package AUT	Event count	Class coverage (%)	Class coverage	Method coverage (%)	Method coverage	Block coverage (%)	Block coverage
ch.hgdev.toposuite	43960	12%	70/573	7%	262/3609	9%	6766/72032
com.achep.acdisplay	43960	39%	250/639	28%	838/3038	25%	20644/82898
com.gladis.tictactoe	43960	0%	0/141	0%	0/896	0%	0/36434
com.glanznig.beepme	43960	19%	29/152	14%	128/929	8%	2275/30203
com.khuttun.notificationnotes	43960	17%	5/29	26%	17/66	9%	251/2926
com.luk.timetable2	43960	7%	40/583	2%	99/5006	2%	2050/116559
com.pindroid	43960	17%	35/204	11%	106/970	8%	1974/24161
com.secuso.privacyFriendlyCodeScanner	43960	30%	123/409	23%	543/2353	13%	18892/145921
com.secuso.torchlight2	43960	15%	3/20	40%	17/43	18%	274/1547
com.yasfa.views	43960	N/A	N/A	N/A	N/A	N/A	N/A
cx.hell.android.pdfview	43960	13%	9/67	11%	43/395	15%	1917/12765
de.markusfisch.android.shadereditor	43960	48%	48/100	44%	287/649	42%	5366/12908
fr.mobdev.goblim	43960	39%	22/57	38%	60/160	20%	982/4903
fr.xtof54.scrabble	43960	N/A	N/A	N/A	N/A	N/A	N/A
jp.takke.cpustats	43960	81%	17/21	75%	84/112	29%	2096/7131
me.anuraag.grader	43960	21%	11/52	12%	21/174	12%	464/3762
net.olejon.spotcommander	43960	29%	65/225	20%	238/1171	22%	6634/29708
net.sourceforge.opencamera	43960	30%	50/166	31%	414/1342	29%	12272/42936
org.androidsoft.games.memory.tux	43960	45%	20/44	73%	156/214	73%	2298/3157
org.mumod.android	43960	12%	38/306	7%	129/1761	5%	2182/48276
org.openintents.notepad	43960	38%	39/104	32%	149/465	31%	4022/13041
org.petero.droidfish	43960	44%	164/377	38%	813/2118	29%	29224/100020
org.pyneo.maps	43960	27%	97/363	22%	473/2131	20%	12982/66423
org.scoutant.blokish	43960	41%	24/58	43%	103/241	45%	4069/8986
org.tmurakam.presentationtimer	43960	91%	10/11	77%	58/75	78%	1146/1463
org.zamedev.gloomydungeons1hardcore.opensource	43960	79%	87/110	66%	395/596	65%	19036/29199
ru.valle.btc	43960	N/A	N/A	N/A	N/A	N/A	N/A
se.tube42.drum.android	43960	70%	193/276	49%	855/1730	40%	21074/52636
se.tube42.kidsmem.android	43960	N/A	N/A	N/A	N/A	N/A	N/A

5.1.1 Trend relationship between test execution time and block coverage

In this section we summarize the results obtained in the first test sessions by graphing the data. The graphs also show a polynomial approximation (dotted lines) of time (equal in all graphs) and code coverage; The greater the area below the code coverage trend line, the better the effectiveness of the testing tool. In particular, this information can be summarized in r (coverage/time ratio):

$$r = \frac{\overline{BC}}{\overline{T}}$$

where $\overline{BC}$ is the average block coverage percentage and $\overline{T}$ is the average execution time. It represents an indicative index of the effectiveness of the testing tool: the higher the value, the better the effectiveness.

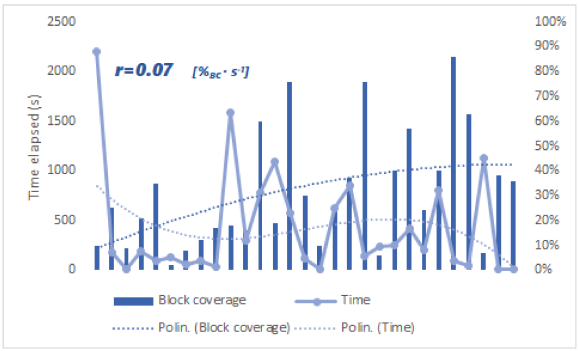


Figure 5.3: PoLiUToDroid: block coverage in relation to testing time

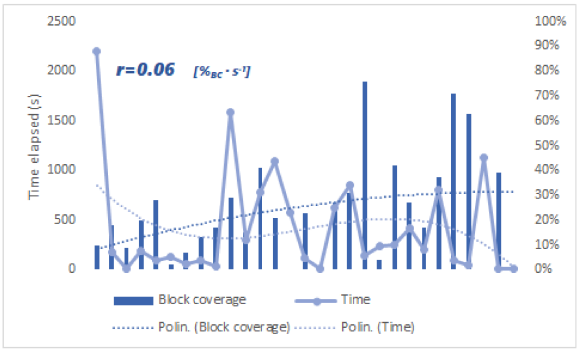


Figure 5.4: DroidBot: block coverage in relation to testing time

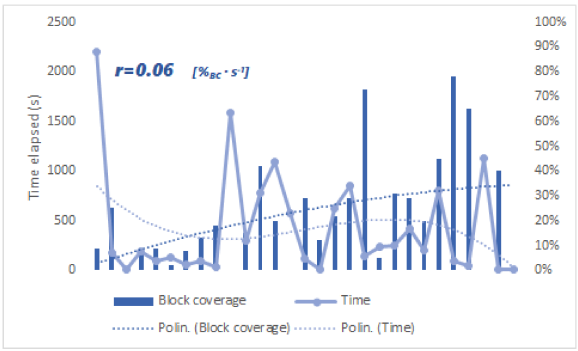


Figure 5.5: Monkey: block coverage in relation to testing time

5.2 PoLiUToDroid vs DroidBot

This section compares the best results achieved by PoLiUToDroid with the best results obtained by DroidBot; The first data is always related to the tool to which the best results are considered in that context.

Table 5.7: Block coverage PoLiUToDroid top 10: comparison with DroidBot

Package AUT	Time (s)	PoLiUToDroid		DroidBot	
		Block coverage (%)	Block coverage	Block coverage (%)	Block coverage
org.tmurakam.presentationtimer	93	86%	1262/1463	71%	1039/1463
org.androidsoft.games.memory.tux	139	76%	2388/3157	76%	2387/3157
fr.xtof54.scrabble	579	76%	10535/13904	N/A	N/A
org.zamedev.gloomydungeons1hardcore.opensource	45	63%	18476/29199	63%	18332/29199
de.markusfisch.android.shadereditor	781	60%	7705/12908	41%	5336/12908
org.petero.droidfish	421	57%	56715/100020	27%	27205/100020
org.openintents.notepad	253	40%	5215/13041	42%	5530/13041
org.scoutant.blokish	809	40%	3571/8986	37%	3341/8986
se.tube42.drum.android	8	38%	19819/52636	39%	20619/52636
net.sourceforge.opencamera	858	37%	15922/42936	31%	13186/42936

Table 5.8: Block coverage DroidBot top 10: comparison with PoLiUToDroid

Package AUT	Time (s)	DroidBot		PoLiUToDroid	
		Block coverage (%)	Block coverage	Block coverage (%)	Block coverage
org.androidsoft.games.memory.tux	139	76%	2387/3157	76%	2388/3157
org.tmurakam.presentationtimer	93	71%	1039/1463	86%	1262/1463
org.zamedev.gloomydungeons1hardcore.opensource	45	63%	18332/29199	63%	18476/29199
org.openintents.notepad	253	42%	5530/13041	40%	5215/13041
de.markusfisch.android.shadereditor	781	41%	5336/12908	60%	7705/12908
se.tube42.drum.android	8	39%	20619/52636	38%	19819/52636
org.scoutant.blokish	809	37%	3341/8986	40%	3571/8986
net.sourceforge.opencamera	858	31%	13186/42936	37%	15922/42936
com.yasfa.views	1589	29%	9921/34283	18%	6199/34283
com.khuttun.notificationnotes	96	28%	815/2926	35%	1037/2926

Table 5.9: Block coverage DroidBot top 10 (with max testing time): comparison with PoLiUToDroid

Package AUT	Time (s)	DroidBot		PoLiUToDroid	
		Block coverage (%)	Block coverage	Block coverage (%)	Block coverage
org.tmurakam.presentationtimer	2197	74%	1088/1463	86%	1262/1463
org.androidsoft.games.memory.tux	2197	73%	2301/3157	76%	2388/3157
org.zamedev.gloomydungeons1hardcore.opensource	2197	64%	18617/29199	63%	18476/29199
org.scoutant.blokish	2197	38%	3417/8986	40%	3571/8986
com.khuttun.notificationnotes	2197	37%	1088/2926	35%	1037/2926
net.sourceforge.opencamera	2197	33%	14303/42936	37%	15922/42936
jp.takke.cpustats	2197	31%	2231/7131	30%	2142/7131
com.yasfa.views	2197	29%	9899/34283	18%	6199/34283
net.olejon.spotcommander	2197	27%	7985/29708	24%	7267/29708
org.petero.droidfish	2197	26%	26233/100020	57%	56715/100020

5.3 PoLiUToDroid vs Monkey

This section compares the best results achieved by PoLiUToDroid with the best results obtained by Monkey; The first data is always related to the tool to which the best results are considered in that context.

Table 5.10: Block coverage PoLiUToDroid top 10: comparison with Monkey (based on event count)

Package AUT	Event count	PoLiUToDroid		Monkey	
		Block coverage (%)	Block coverage	Block coverage (%)	Block coverage
org.tmurakam.presentationtimer	50	86%	1262/1463	41%	600/1463
org.androidsoft.games.memory.tux	66	76%	2388/3157	26%	814/3157
fr.xtof54.scrabble	263	76%	10535/13904	75%	10461/13904
org.zamedev.gloomydungeons1hardcore.opensource	26	63%	18476/29199	5%	1485/29199
de.markusfisch.android.shadereditor	415	60%	7705/12908	28%	3649/12908
org.petero.droidfish	289	57%	56715/100020	35%	3187/8986
org.openintents.notepad	172	40%	5215/13041	4%	535/13041
org.scoutant.blokish	289	40%	3571/8986	35%	3187/8986
se.tube42.drum.android	N/A	38%	19819/52636	N/A	N/A
net.sourceforge.opencamera	686	37%	15922/42936	25%	10634/42936

Table 5.11: Block coverage PoLiUToDroid top 10: comparison with Monkey (based on testing time)

Package AUT	Time (s)	PoLiUToDroid		Monkey	
		Block coverage (%)	Block coverage	Block coverage (%)	Block coverage
org.tmurakam.presentationtimer	93	86%	1262/1463	78%	1146/1463
org.androidsoft.games.memory.tux	139	76%	2388/3157	73%	2298/3157
fr.xtof54.scrabble	579	76%	10535/13904	N/A	N/A
org.zamedev.gloomydungeons1hardcore.opensource	45	63%	18476/29199	65%	19036/29199
de.markusfisch.android.shadereditor	781	60%	7705/12908	42%	5366/12908
org.petero.droidfish	421	57%	56715/100020	29%	29224/100020
org.openintents.notepad	253	40%	5215/13041	31%	4022/13041
org.scoutant.blokish	809	40%	3571/8986	45%	4069/8986
se.tube42.drum.android	8	38%	19819/52636	40%	21074/52636
net.sourceforge.opencamera	858	37%	15922/42936	29%	12272/42936

Table 5.12: Block coverage Monkey top 10 (with max testing time): comparison with PoLiUToDroid

Package AUT	Time (s)	Monkey		PoLiUToDroid	
		Block coverage (%)	Block coverage	Block coverage (%)	Block coverage
org.tmurakam.presentationtimer	2197	78%	1146/1463	86%	1262/1463
org.androidsoft.games.memory.tux	2197	73%	2298/3157	76%	2388/3157
org.zamedev.gloomydungeons1hardcore.opensource	2197	65%	19036/29199	63%	18476/29199
org.scoutant.blokish	2197	45%	4069/8986	40%	3571/8986
de.markusfisch.android.shadereditor	2197	42%	5366/12908	60%	7705/12908
se.tube42.drum.android	2197	40%	21074/52636	38%	19819/52636
org.openintents.notepad	2197	31%	4022/13041	40%	5215/13041
jp.takke.cpustats	2197	29%	2096/7131	30%	2142/7131
net.sourceforge.opencamera	2197	29%	12272/42936	37%	15922/42936
org.petero.droidfish	2197	29%	29224/100020	57%	56715/100020

5.4 Average overall coverage rates

In this section, we show average results (of all apps with a valid coverage) achieved by each tool, grouped by testing session.

Table 5.13: PoLiUToDroid: average overall coverage rates

Number of apps with valid coverage	Class coverage	Method coverage	Block coverage
29	39,67%	36,02%	31,09%

Table 5.14: DroidBot: average overall coverage rates (with equal testing time of *Table 5.1*)

Number of apps with valid coverage	Class coverage	Method coverage	Block coverage
25	36,90%	33,40%	27,50%

Table 5.15: DroidBot: average overall coverage rates (with max testing time of *Table 5.1*)

Number of apps with valid coverage	Class coverage	Method coverage	Block coverage
17	41,72%	40,15%	31,61%

Table 5.16: Monkey: average overall coverage rates (with equal event count of *Table 5.1*)

Number of apps with valid coverage	Class coverage	Method coverage	Block coverage
28	24,34%	19,31%	19,04%

Table 5.17: Monkey: average overall coverage rates (with equal testing time of *Table 5.1*)

Number of apps with valid coverage	Class coverage	Method coverage	Block coverage
24	36,03%	32,89%	26,92%

Table 5.18: Monkey: average overall coverage rates (with max testing time of *Table 5.1*)

Number of apps with valid coverage	Class coverage	Method coverage	Block coverage
21	35,21%	34,59%	27,72%

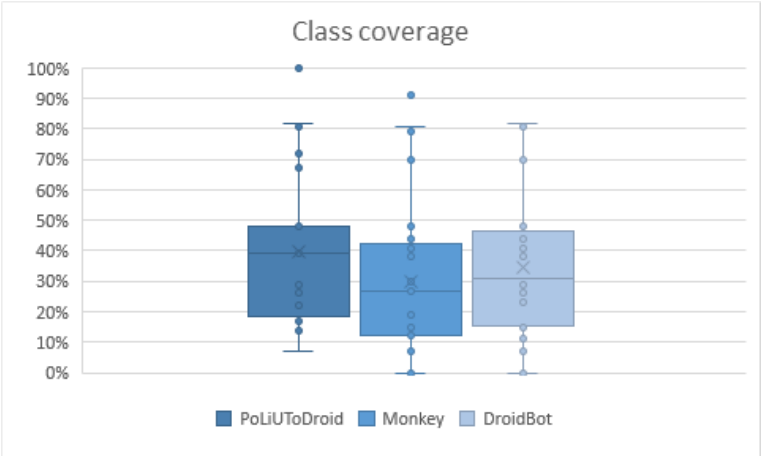


Figure 5.6: Class coverage of the best available results

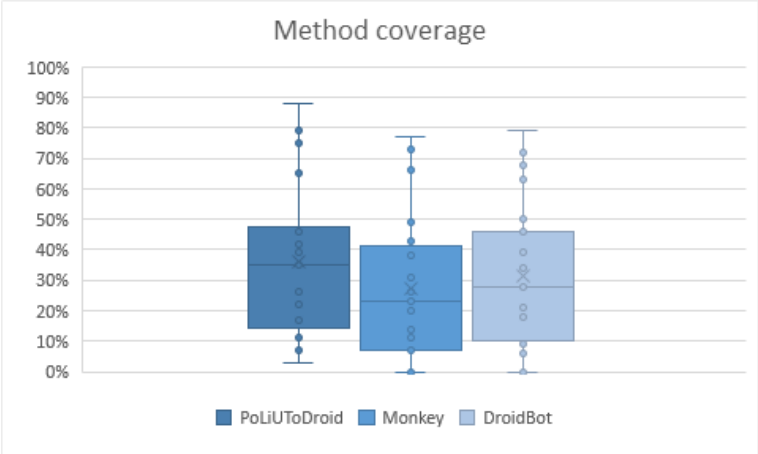


Figure 5.7: Method coverage of the best available results

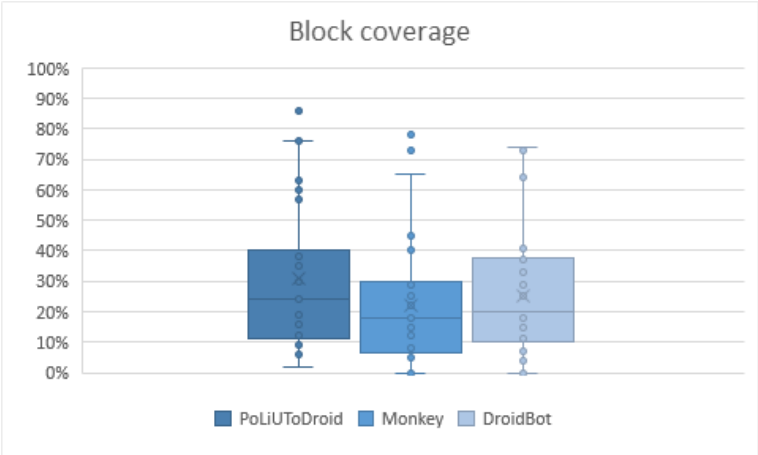


Figure 5.8: Block coverage of the best available results

5.5 PoLiUToDroid: CPU/Memory performance

In order to address **RQ2.c** and **RQ3.c** and, hence, demonstrate the efficiency (i.e., low resource use) of our implementation, we reported in this section the performance of PoLiUToDroid measured for the longest test run (that is, the test of TopoSuite).

Table 5.19: TopoSuite screen-based performance: top 10 CPU

Activity name	Time elapsed (hh:mm)	CPU usage	Memory usage (KB)
MainActivity	00:00	11,2%	10868
MainActivity	00:01	12,1%	12417
CheminementOrthoActivity	00:08	9,8%	19027
OrthogonalImplantationActivity	00:10	9,9%	20731
CheminementOrthoActivity	00:23	20,5%	26935
PolarImplantationActivity	00:23	9,4%	27907
MainActivity	00:28	11,2%	29883
MainActivity	00:30	13,6%	30912
MainActivity	00:30	9,8%	31977
CheminementOrthoActivity	00:35	9,7%	29374

Table 5.20: TopoSuite screen-based performance: top 10 memory

Activity name	Time elapsed (hh:mm)	CPU%	Memory (KB)
AxisImplantationActivity	00:34	3,7%	36131
LeveOrthoActivity	00:35	3,8%	36659
LeveOrthoActivity	00:35	2,4%	36683
CheminementOrthoActivity	00:35	3,7%	36406
CheminementOrthoActivity	00:35	3,6%	36418
LineCircleIntersectionActivity	00:35	6,3%	36574
PolarImplantationActivity	00:36	3,3%	36850
LineCircleIntersectionActivity	00:36	6,0%	36098
LineCircleIntersectionActivity	00:36	3,4%	36914
MainActivity	00:36	3,9%	36182

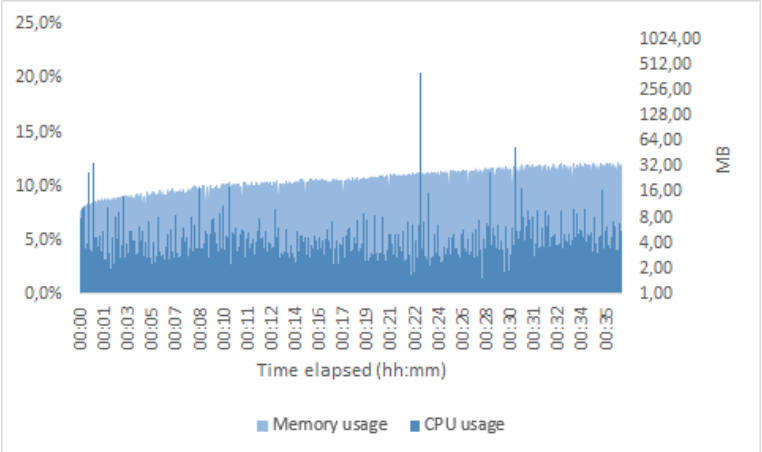


Figure 5.9: TopoSuite overall performance

6 Discussion

In this chapter we discuss about our achievement. For an objective interpretation of these results, we have also taken into account the criteria suggested by Choudhary, Gorla, and Orso [38] for a general evaluation of an automatic input generation tools.

Main evaluation criteria

1. **Ease of use.** Usability is certainly a key factor for all tools, especially if they are just research prototypes, as it highly affects their reuse.
2. **Android framework compatibility.** One of the major problems that Android developers have to constantly deal with is the fragmentation. Test input generation tools for Android should therefore ideally run on devices that have different hardware characteristics and use different releases of the Android framework, so that developers could assess how their apps behave in different environments.
3. **Code coverage achieved.** The aim of test input generators should be to cover as much behavior as possible of the app under test. Since code coverage is a commonly used criterion for evaluating behavior coverage, we measured the statement (i.e., basic block) coverage that each tool achieved on each benchmark and then compared the results of the different tools.
4. **Fault detection ability.** The primary goal of testing tools is to expose existing faults and, for this reason, it is convenient that they implement failure detection features.

6.1 Results

In general, code coverage is unable to give normalized information about the effectiveness of test cases, but it can provide meaningful statistics when comparing different test cases on the same app. Since the most critical resource for an input generator is the time, these tools should be evaluated on the basis of how much coverage they can achieve within a certain time limit. For this reason, in the ***Sections*** 5.2 and 5.3, we compared PoLiUToDroid's results with the best 10 benchmarks (in terms of block coverage percentage) obtained by running DroidBot and Monkey with different time and event count (for Monkey) constraints.

In the first case, we limit the execution time by using the time that PoLiUToDroid spent to test the same app. In this situation, we are able to extract information (reported in the ***Section 5.1.1***) about the coverage/time ratio r. More precisely, it aims to represent the percentage of code coverage per second that a testing tool is able to achieve. Even though slightly, PoLiUToDroid obtains the best achievement.
In another case, we limit the execution time by using the maximum time spent by PoLiUToDroid among all its tests, enabling each tool to achieve maximum results. Despite this, it is rare that other tools get better results than PoLiUToDroid.
In the last case, only for Monkey, we limited the event count by using the number of events generated by PoLiUToDroid to test the same app. As expected, we obtained results that highlight the effectiveness of a systematic technique compared to a random one.

For each testing session, we calculated, and reported in the ***Section 5.4***, the average of all apps with a valid coverage. This information shows that PoLiUToDroid achieves on average higher code coverage than other tools. We would like to point out that, albeit higher, the average code coverage rates in *Table 5.15* are strongly influenced by the fact that it was impossible to retrieve the code coverage information from most of the tested applications.

The graphs reported in *Figures 5.3*, *5.4* and *5.5* show the trend relationship between test execution time and block coverage achieved. These data prove that our novel approach for inferring the model and generating events can lead to better results in less time, particularly if PoLiUToDroid will be enriched with other missing user inputs.

To further investigate these results, *Figures 5.6*, *5.7* and *5.8* present the boxplots (for which an X indicates the mean) of the coverage results for apps. This analysis reveals that in most applications there was no meaningful difference in code coverage between tools, but PoLiUToDroid achieved the highest mean coverage across all available results, despite not implementing some important features like scroll, swipe, and screen rotation.
We conclude that there is evidence that PoLiUToDroid can efficiently attain superior coverage and, thus, discover a higher number of AUT's behaviors.

Finally, in order to answer **RQ2.c** and **RQ3.c**, we analyze the performance of PoLiUToDroid (and then of the active learning and exploration techniques) measured for the longest test run (that is, the test of TopoSuite).
As can be seen from *Table 5.20*, the information retained for identifying a GUI state is largely within the reach of any mobile device. Most information is expressed in the form of Strings and tree structures or graphs, which are definitely efficient in both search and comparison.
Monitoring data presented in the *Figure 5.9* shows a very low use of resources. In fact, by deciding to map all the properties of a GUI state into a *signature*, the comparison method has been reduced to the simple comparison between Strings, making the comparison operation itself and the information retention simple and efficient. This mechanism allows to PoLiUToDroid to be used in applications with a large number of screens.

6.2 Method

PoLiUToDroid implements a non-invasive automatic black-box UI testing technique for Android mobile applications using a novel active learning approach. It is able to retrieve system and application information and to inject user inputs to the device connected through ADB. Both the monitoring and input phases are non-invasive because they are based on UiAutomation, UI Automator framework, and existing Android debugging/testing utilities, which are embedded within Android. Nevertheless, the fact that PoLiUToDroid must be started via ADB could provide malware a policy to decide not to manifest its malicious behavior and, for this reason, this aspect should be considered for future work.

While exploring the application, PoLiUToDroid infers a model of the AUT based on the information retrieved at runtime. The model leads the input generation to avoid redundant test input choices thus aiming to optimize the strategy with which exploring the AUT itself.

The E-LTS learning algorithm allows to PoLiUToDroid to minimize the number of restarts of the AUT in order to achieve greater exploration efficiency than other testing tools. In fact, it attempts to extend the current execution path by performing an action that is available but not yet triggered on the current state or by using, in an appropriate manner, the BACK button of Android devices to return to a previous state.

The model learning process relies on the underlying multi-level comparison criteria. It aims to achieve an appropriate (user-configurable) abstraction level of the GUI model in order to avoid both to represent fewer app's behaviors than those that can actually be manifested and to incur to state explosion problems.

PoLiUToDroid can be used in several fields of application: One of the possible scope is compatibility testing. It aims to determine whether the hardware and software of different devices display and allow the application to work properly.
Since compatibility testing should be performed on many different devices, system instrumentation is unrealistic. Meanwhile, app instrumentation might also be unwanted because an instrumented application may behave differently from the original app. With PoLiUToDroid, a developer is able to test apps on different devices without instrumentation, reaching more GUI states in optimized time.

For this reason as well, malware analysis is also a useful scenario of PoLiUToDroid. As many malware encrypt their code or check their signature before doing malicious things, it might be impossible to instrument them or guarantee the consistency between the instrumented app and the original application.
Even Monkey, for instance, can test malware without instrumentation, but its random strategy might not be able to discover the malicious behaviors. In fact, as experiments conducted by Desnos and Lantz [39] show, the amount of sensitive behaviors triggered by using a systematic exploration strategy is much higher than randomized inputs generated by Monkey.

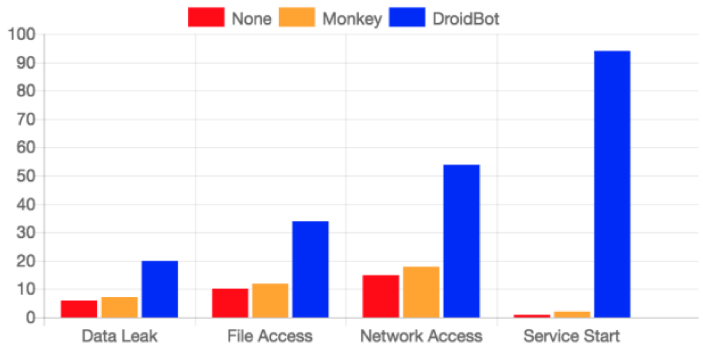


Figure 6.1: Total number of sensitive behaviors in four categories [39]

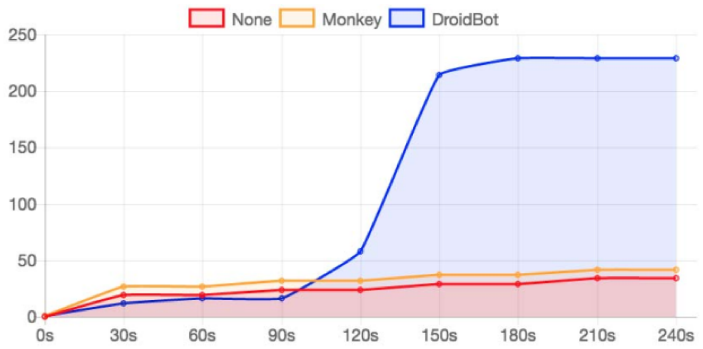


Figure 6.2: Speed of triggering sensitive behaviors [39]

The reason for this ineffectiveness may be that malware could require users to touch a specific sequence of actions before getting into a malicious state.
In addition, PoLiUToDroid, as well as being easy to use like Monkey, is also more effective in exploring applications as it uses a model-based strategy. PoLiUToDroid can also simulate a human-like usage as well as generating events that refer to real device information (i.e., non-dummy) so that it can not be distinguished from a human user.

Finally, a feature present in PoLiUToDroid, but that should be improved, is that of error detection and reporting. Currently, it is able to detect and manage "Application Not Responding" (ANR) dialog and crash and provides to the user human reproducible steps log. According to studies conducted by Choudhary, Gorla, and Orso [38], "none of the Android tools can identify failures other than runtime exceptions"; For this reason, we believe that this may be one of the testing topics to be addressed in the future, with particular attention to the generation of test cases for easily reproducing the identified failures.

6.3 The work in a wider context

Testing & Security Awareness

Cybersecurity is very important in a world such as today, where the network and mobile devices dominate most of the social, business, economic, and political relationships.
With more than 2 billion monthly active users [23] and 2.8 million applications available on Google's digital market [25], the Android mobile world is faced with ever-growing volumes of ever-evolving threats.
Nowadays, cybersecurity is critical to protect everyday life. Since people rely on technological devices every day, their protection is really crucial to avoid that sensitive information (such as bank, financial, personal and confidential information) could be acquired by third parties without user consent.

Cybercrime is widespread, and is generally aimed at attacking software or devices in order to trap sensitive information. Every day, companies and individuals are unknowingly subject to various cyber attacks. More and more sophisticated threats emerge, while cybercriminals develop new techniques to bypass traditional security technologies. Traditional security solutions such as anti-virus, firewall and intrusion prevention systems are no longer able to provide complete protection, as well as often not being applicable to devices with limited capabilities such as smartphones. To overcome these gaps in security, new detection and protection techniques are needed for new malware, and security has become a core concern for the mobile business.

Vulnerabilities might occur due to various reasons, and more and more recent attacks are associated with the way the mobile applications are used. Moreover, there is no doubt that the software industry is experiencing a historic moment in which the quality is vital. In fact, consumer and business users are constantly within an application-driven digital universe, enabling them to benefit from services capable to determine the success of a company and service.
In this scenario, automatic testing, combined with malware analysis techniques, could prove to be both an effective threat detection technique, able to highlight malicious behaviors hidden in the apps, and, at the same time, a time-optimized method to ensure software quality.

7 Conclusion

In this thesis we described a non-invasive automatic black-box UI testing technique for Android mobile applications based on a novel active learning approach and presented PoLiUToDroid as a GUI Ripping-based implementation.

Our work started with the evaluation of the state-of-the-art approaches for automated GUI testing of Android applications. This study has highlighted some shortcomings in the testing techniques presented in the literature, both in terms of adequacy and versatility. To cope with these limitations, we focused our studies on achieving a triple objective:

1. Investigate on a model-learning approach able to properly distinguish the different Android apps GUI states;
2. Design a black-box testing technique that is not limited to analyze a single application, but is therefore capable of interacting with, ideally, any app started by the AUT;
3. Realize a testing tool capable of operating as transparently as possible on limited capacity real devices, so that malicious applications can be analyzed avoiding that they detect that they are operating in a testing environment.

Since a superficial and nowadays often inadequate comparison criterion (i.e., Activity-based) is frequently used to determine the equality between two GUI states, we decided to found our comparison mechanism on a multi-level GUI abstraction model.

We defined an extended version of a deterministic labeled transition system (E-LTS) to describe event-driven systems such as Android applications, and a novel model learning algorithm that aims to avoid redundant test input choices and minimize the number of restarts of the system (e.g., a mobile app) under test.

Finally, since PoLiUToDroid was also intended for use in software security analysis, we considered the evolution of Android malware in order to find appropriate countermeasures to allow our tool to work without being detected.

In the rest of the chapter, we recap the research questions and give them a clear and concise answer based on the studies carried out and the results obtained. Finally, we will announce some possible future prospects.

7.1 Answers to Research Questions

***RQ1*: Which GUI exploration techniques are most suitable for testing/analyzing closed source, obfuscated and malicious apps?**

After extensive research, we decided to base our Android mobile apps GUI exploration approach exclusively on dynamic techniques, both to avoid invasive (and therefore easily detectable) operations on tested applications and to ensure bypassing certain types of honeypots (e.g., feasible through exported Activities).

In particular, a black-box approach, usable in most cases, is based on the GUI ripping technique; In this regard, we have relied on the UI Automator framework, which provides simple APIs to detect and interact with UI graphical components, and on UiAutomation for injecting ad hoc (human-like) user events on the connected device.
The use of these frameworks is in fact well-suited for developing automated tests without relying on internal implementation details of the target app, so as to be able to test applications regardless of whether their source code is unavailable or obfuscated.

By using UiAutomation, we can also generate event objects containing non-dummy values, so that malicious apps are not able to classify PoLiUToDroid as a testing/analysis tool.

***RQ1.a*: What could be the precautions to try to avoid that malicious applications detect the activity of a testing/analysis tool?**

Some precautions, implemented by PoLiUToDroid, that can be taken to avoid being identified as an automatic testing tool are:

- Introduction of a reasonable random delay between performing consecutive inputs;
- Identification of NAF (Not Accessibility Friendly) components so that, if requested, they will not be triggered by actions so as to avoid GUI honeypots;
- Injection of events characterized by real information (unlike those that are obtained by using standard UI Automator APIs);
- Not relying on static techniques (e.g., parsing the Android Manifest or bytecode analysis) in order to avoid honeypots.

***RQ2*: Which could be an effective GUI abstraction of GUI States?**

The development of increasingly complex applications has led to the fact that the *Activity↔GUI state* mapping being no longer enough to differentiate the various screens of a GUI; In today's applications, an Activity can in fact be characterized by multiple screens by using dynamic Fragments.

For this reason, we are convinced that a multi-level GUI abstraction is necessary and suitable to characterize the different mobile app GUI states. With this approach it is possible to take into account various AUT's components, in order to avoid to represent fewer app's behaviors than those that can actually be manifested. At the same time, a configurable multi-level GUI abstraction model makes the tester capable to adjust the level of abstraction with which to analyze the app so as to avoid state explosion problems.

***RQ2.a*: How can GUI-related information be retrieved from closed-source apps?**

Like many testing frameworks, UI Automator allows to retrieve information about any graphic component on the screen of the connected device. Its APIs simply allow to identify and interact with the GUI of the AUT without relying on invasive techniques. UI Automator framework is in fact well-suited for writing black box-style automated tests and also provides useful tools and features (e.g., cross-app UI testing) which make it appealing.

***RQ2.b*: What GUI information is most useful for characterizing the GUI state?**

For an event generation tool, it is crucial to distinguish all the graphical components that make up a GUI, but also to identify the actions that can be triggered on each widget: The UI Automator framework allows to access this information. However, we also want to be able to operate on devices with limited resources and that is why we need to pay attention to the amount of information we hold.

A GUI state can be identified by the information described in the Multi-level GUI comparison criteria section and implemented as described in the Method chapter. In particular, we believe that a good trade-off of GUI abstraction level lies in the C-Lv4 of that model, where the AUT's package, Activity name, layout CISs and executable widget's properties are considered to characterize a GUI state.
To avoid redundant test inputs in the exploration strategy, it is also important to enrich each GUI state with appropriate state variables. We decided to introduce the E-LTS model to describe an event-driven system like an Android mobile application, where such state variables are the set of available actions and the set of inputs considered previously triggered on the state; They play a key role in the proposed learning algorithm.

***RQ2.c*: Taking into account the need to operate on devices with limited resources, how can GUI States be represented in an efficient way?**

A suitable solution to minimize the use of embedded system resources and at the same time maintain good efficiency in both search and comparison, is to represent much of the information as strings and manage them in tree structures or graphs.
As we expected, the information kept by PoLiUToDroid is largely within the memory capacity of any mobile device and also the CPU usage is minimal.

***RQ3*: Which could be an effective comparison criterion to distinguish the GUI States?**

Based on the fact that Android applications have a hierarchical structure where a package can include many Activities and each Activity can provide a hierarchical layout with multiple UI components, we decided to consider multi-level comparison criteria to distinguish the GUI states. In fact, a layered comparison model meets the needs to identify different screens that belong to the same Activity, but also give the user the possibility to determine which abstraction level is best suited to test the target application.

***RQ3.a*: What information needs to be considered to evaluate the equality between two GUI States?**

PoLiUToDroid identifies each GUI state by means of a signature, built on the basis of the multi-level GUI abstraction model, which can then consider, depending on the user's needs, graphical contents and properties of executable widgets, as well as package and Activity to which the screen belongs.

***RQ3.b*: How shall apps that start other apps be handled?**

UI Automator testing framework allows to capture and manipulate UI components across multiple apps. PoLiUToDroid, as it exploits UI Automator APIs, offers the ability to interact with any application started by the target app. Therefore, this feature enable to manage all those cases where the AUT delegates to external applications for accomplishing tasks (e.g., take a picture). Despite this being a common behavior in Android, we are not aware of other testing tools able to handle it.
Moreover, to avoid state explosion problems, PoLiUToDroid can be configured by the user in order to limit the depth reachable outside of the AUT package.

***RQ3.c*: How can GUI state comparisons be implemented in an efficient and scalable way?**

By deciding to map all the properties of a GUI state into a signature, the comparison method is reduced to the simple verification of the equality of strings, making the comparison operation itself and the maintenance of the information simple and efficient. For these reasons, the proposed mechanism allows to scale even on applications made up of a large number of screens without compromising performance.

7.2 Future work

In future work we plan to extend PoLiUToDroid both to enrich it with other features and to better integrate it into a security environment and malware analysis.

Firstly, we intend to consider other types of events (e.g., scroll, swipe and screen rotation) in the exploration process, in order to trigger even more behaviors in Android applications. For our own purposes, all actions performed by PoLiUToDroid must be characterized by input objects containing real information (i.e., information regarding the hardware onboard), and for this reason, it is not enough to implement them by using standard UI Automator APIs.

Useful features that could be implemented are the generation of the learned model (usable by other model-based testing tools) and of test cases (in order to easily reproduce the identified failures); For this purpose, the error detection capability should also be improved.

We also contemplate to improve the capability to identify not accessibility friendly UI components by integrating a mechanism (e.g., image-based) to certainly determine the widgets visibility from a human's-eye point of view (e.g., if a widget has the same color as the nearby pixels of background, it would be unrecognizable by a user).

Last but not least, the fact that PoLiUToDroid must be started via ADB provides malware a policy to decide not to manifest its malicious behavior. For this reason, a security topic to be discussed in the future should include masking this behavior (e.g., via system hooks) or finding an alternative way to transparently run the tool.

A Appendix

A.1 Workstation details and test settings

A.1.1 PoLiUToDroid configuration

- max-steps: 999
- max-depth: 30
- max-foreign-depth: 0
- max-runtime: 3600
- comparison-level: 4
- min-sleep-action: 0
- max-sleep-action: 0
- min-widget-percentage: 1
- test-naf: true

A.1.2 DroidBot configuration

```
droidbot -a APK_PATH -o OUTPUT_DIR
         -keep_app -keep_env
         -is_emulator -grant_perm -timeout TIMEOUT
```

A.1.3 Monkey configuration

```
adb shell monkey -p PACKAGE --throttle 50 -v
                 --ignore-crashes --ignore-timeouts
                 --ignore-security-exceptions COUNTS
```

A.1.4 Testing environment information

All tests has been performed in the following environment:

- Linux distribution: Ubuntu 17.04
- CPU details:
  - Architecture: x86_64
  - CPU op-mode(s): 32-bit, 64-bit
  - Byte Order: Little Endian
  - CPU(s): 4
  - On-line CPU(s) list: 0-3
  - Thread(s) per core: 2
  - Core(s) per socket: 2
  - Socket(s): 1
  - NUMA node(s):1
  - Vendor ID: GenuineIntel
  - CPU family: 6
  - Model: 142
  - Model name: Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz
  - Stepping: 9
  - CPU MHz:819.165
  - CPU max MHz: 3500,0000
  - CPU min MHz: 400,0000
  - BogoMIPS: 5808.00
  - Virtualization: VT-x
  - L1d cache: 32K
  - L1i cache: 32K
  - L2 cache: 256K
  - L3 cache: 4096K
  - NUMA node0 CPU(s):0-3
- Android Emulator:
  - Device: Nexus 5X (Google)
  - Target: Google APIs (Google Inc.)
  - Based on: Android 7.0 (Nougat) Tag/ABI: google_apis/x86
  - Skin: nexus_5x
  - Sdcard: 100M

Bibliography

[1] Web Page. URL: https://developer.android.com/studio/test/monkey.html.

[2] Web Page. URL: https://developer.android.com/training/basics/fragments/index.html.

[3] Web Page. URL: https://developer.android.com/index.html.

[4] Web Page. URL: https://source.android.com/.

[5] Figure. URL: https://developer.android.com/guide/platform/index.html.

[6] Figure. URL: https://developer.android.com/guide/components/activities/activity-lifecycle.html.

[7] Web Page. URL: https://github.com/mockito/mockito.

[8] Figure. URL: https://developer.android.com/training/testing/start/index.html#test-types.

[9] Web Page. URL: https://developer.android.com/reference/android/app/UiAutomation.html.

[10] Web Page. URL: https://developer.android.com/training/testing/ui-automator.html#ui-automator-viewer.

[11] Web Page. URL: https://developer.android.com/reference/android/view/MotionEvent.html.

[12] Web Page. URL: https://developer.android.com/reference/android/view/KeyEvent.html.

[13] Web Page. URL: https://developer.android.com/training/testing/espresso/index.html.

[14] Web Page. URL: https://developer.android.com/training/testing/ui-automator.html.

[15] Web Page. URL: https://developer.android.com/studio/test/monkeyrunner/index.html.

[16] Web Page. URL: https://github.com/RobotiumTech/robotium.

[17] Web Page. URL: https://github.com/appium/appium.

[18] Web Page. URL: https://github.com/calabash/calabash-android.

[19] Web Page. URL: https://cucumber.io/.

[20] Web Page. URL: https://developer.android.com/studio/index.html.

[21] Web Page. URL: https://github.com/paul-irolla/androcoverage.

[22] Web Page. URL: http://emma.sourceforge.net/.

[23] Blog. URL: https://twitter.com/Google/status/864890655906070529.

[24] Blog. 2007. URL: https://googleblog.blogspot.se/2007/11/wheres-my-gphone.html.

[25] Figure. 2017. URL: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[26] Figure. 2017. URL: https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/.

[27] Figure. 2017. URL: https://developer.android.com/about/dashboards/index.html#Platform.

[28] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. "A gui crawling-based technique for android mobile application testing". In: *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. IEEE, pp. 252–261. ISBN: 0769543456.

[29] Domenico Amalfitano et al. "A toolset for GUI testing of Android applications". In: *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, pp. 650–653. ISBN: 1467323128.

[30] Domenico Amalfitano et al. "MobiGUITAR: Automated model-based testing of mobile apps". In: *IEEE Software* 32.5 (2015), pp. 53–59. ISSN: 0740-7459.

[31] Domenico Amalfitano et al. "Using GUI ripping for automated testing of Android applications". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, pp. 258–261. ISBN: 1450312047.

[32] Saswat Anand et al. "Automated concolic testing of smartphone apps". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, p. 59. ISBN: 145031614X.

[33] Dana Angluin. "Learning regular sets from queries and counterexamples". In: *Information and computation* 75.2 (1987), pp. 87–106. ISSN: 0890-5401.

[34] Tanzirul Azim and Iulian Neamtiu. "Targeted and depth-first exploration for systematic testing of android apps". In: *Acm Sigplan Notices*. Vol. 48. ACM, pp. 641–660. ISBN: 145032374X.

[35] Young-Min Baek and Doo-Hwan Bae. "Automated model-based android gui testing using multi-level gui comparison criteria". In: *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, pp. 238–249. ISBN: 1450338453.

[36] Hugo Bruneliere et al. "MoDisco: a generic and extensible framework for model driven reverse engineering". In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, pp. 173–174. ISBN: 1450301169.

[37] Wontae Choi, George Necula, and Koushik Sen. "Guided gui testing of android apps with minimal restart and approximate learning". In: *Acm Sigplan Notices*. Vol. 48. ACM, pp. 623–640. ISBN: 145032374X.

[38] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. "Automated test input generation for android: Are we there yet?(e)". In: *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, pp. 429–440. ISBN: 1509000259.

[39] Anthony Desnos and Patrik Lantz. "Droidbox: An android application sandbox for dynamic analysis". In: (2011). URL: https://code.google.com/p/droidbox.

[40] Wenrui Diao et al. "Evading android runtime analysis through detecting programmed interactions". In: *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM, pp. 159–164. ISBN: 1450342701.

[41] F-Secure. *The State of Cyber Security 2017*. Report. 2017. URL: https://business.f-secure.com/the-state-of-cyber-security-2017.

[42] Jyoti Gajrani et al. "A robust dynamic analysis system preventing SandBox detection by Android malware". In: *Proceedings of the 8th International Conference on Security of Information and Networks*. ACM, pp. 290–295. ISBN: 1450334539.

[43] Shuai Hao et al. "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps". In: *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, pp. 204–217. ISBN: 1450327931.

[44] Yiming Jing et al. "Morpheus: automatically generating heuristics to detect Android emulators". In: *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, pp. 216–225. ISBN: 1450330053.

[45] Robert M Keller. "Formal verification of parallel programs". In: *Communications of the ACM* 19.7 (1976), pp. 371–384. ISSN: 0001-0782.

[46] Tomi Lämsä. "Comparison of GUI testing tools for Android applications". In: (2017).

[47] Yuanchun Li et al. "DroidBot: a lightweight UI-guided test input generator for Android". In: *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, pp. 23–26. ISBN: 1538615894.

[48] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. "Dynodroid: An input generation system for android apps". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, pp. 224–234. ISBN: 1450322379.

[49] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. "Evodroid: Segmented evolutionary testing of android apps". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, pp. 599–609. ISBN: 1450330568.

[50] Ke Mao, Mark Harman, and Yue Jia. "Sapienz: Multi-objective automated testing for android applications". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, pp. 94–105. ISBN: 1450343902.

[51] Henry Muccini, Antonio Di Francesco, and Patrizio Esposito. "Software testing of mobile applications: Challenges and future research directions". In: *Proceedings of the 7th International Workshop on Automation of Software Test*. IEEE Press, pp. 29–35. ISBN: 1467318221.

[52] Simone Mutti et al. "BareDroid: Large-scale analysis of Android apps on real devices". In: *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, pp. 71–80. ISBN: 1450336825.

[53] Bao N Nguyen et al. "GUITAR: an innovative tool for automated testing of GUI-driven software". In: *Automated Software Engineering* 21.1 (2014), pp. 65–105. ISSN: 0928-8910.

[54] Thanasis Petsas et al. "Rage against the virtual machine: hindering dynamic analysis of android malware". In: *Proceedings of the Seventh European Workshop on System Security*. ACM, p. 5. ISBN: 145032715X.

[55] Alireza Sadeghi et al. "A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software". In: *IEEE Transactions on Software Engineering* 43.6 (2017), pp. 492–530. ISSN: 0098-5589.

[56] Lance Spitzner. *Honeypots: tracking hackers*. Vol. 1. Addison-Wesley Reading, 2003.

[57] Kimberly Kim-Chi Tam. "Analysis and Classification of Android Malware". In: (2016).

[58] Kimberly Tam et al. "The evolution of android malware and android analysis techniques". In: *ACM Computing Surveys (CSUR)* 49.4 (2017), p. 76. ISSN: 0360-0300.

[59] AV-TEST. *Security Report 2016/17*. Report. 2017. URL: https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2016-2017.pdf.

[60] Mark Utting, Alexander Pretschner, and Bruno Legeard. "A taxonomy of model-based testing approaches". In: *Software Testing, Verification and Reliability* 22.5 (2012), pp. 297–312. ISSN: 1099-1689.

[61] Timothy Vidas and Nicolas Christin. "Evading android runtime analysis via sandbox detection". In: *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, pp. 447–458. ISBN: 1450328008.

[62] Fengguo Wei et al. "Deep Ground Truth Analysis of Current Android Malware". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, pp. 252–276.

[63] Wei Yang, Mukul R Prasad, and Tao Xie. "A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications". In: *FASE*. Vol. 13. Springer, pp. 250–265.

[64] Dongsong Zhang and Boonlit Adipat. "Challenges, methodologies, and issues in the usability testing of mobile applications". In: *International journal of human-computer interaction* 18.3 (2005), pp. 293–308. ISSN: 1044-7318.

[65] Yury Zhauniarovich et al. "Towards black box testing of Android apps". In: *Availability, Reliability and Security (ARES), 2015 10th International Conference on*. IEEE, pp. 501–510. ISBN: 1467365904.