Modelling Android applications through static analysis and systematic exploratory testing

Jordan Doyle[1,*], Thomas Laurent[2], and Anthony Ventresque[3]
[1]SFI Lero & School of Computer Science, University College Dublin, Dublin, Ireland
[2]JSPS International Research Fellow, National Institute of Informatics, Tokyo, Japan
[3]SFI Lero & School of Computer Science and Statistics, Trinity College Dublin, Dublin, Ireland
jordan.doyle@ucdconnect.ie, thomas-laurent@nii.ac.jp, anthony.ventresque@tcd.ie
*corresponding author

Abstract—Mobile application development is a fast paced industry with frequent releases. While the development pace increases, so too does the need for automated test generation. Model-based test generation is one of the most common and successful approaches to support this need. Understanding and modelling the application under test is integral to producing comprehensive, dependable and effective tests. Unfortunately, mobile platforms such as Android, introduce a host of difficulties. Static analysis struggles with Android's event-based nature and the growing variety of mechanisms available for developers to implement different features. Additionally, dynamic analysis, implemented by popular random test generators, is slow, inefficient, and limited by a lack of application knowledge.

This paper introduces DroidGraph, a framework to generate a comprehensive control flow model of Android applications using traditional static analysis and efficient systematic exploratory tests. DroidGraph provides a detailed model of an Android application, from low level method statements to high level user interface structures. This model can be used to support automated test generation. We apply DroidGraph to 19 diverse apps and show that our efficient exploratory tests, on average, interact with 18% more of the app than commonly used random exploration in 345 less interactions. Integrating the dynamic analysis results provided by these tests complements our static analysis, and uncovers on average 51 more components and 49% more interface callback links in the application code.

Keywords–Android, Static analysis, Exploratory testing, Automated testing

1. Introduction

Testing is the established technique to ensure stability and reliability in any software-based system. However, with the fast pace commercialisation and increasing marketability of mobile platforms it has become increasingly important to efficiently create strong test suites for complex applications.

Mobile testing remains largely manual, despite the considerable amount of research around test automation [1]–[5]. Unit testing, while essential to ensure basic functionality during development, is time consuming and becomes more demanding as the application is developed. Even testing methods designed to reduce a developers workload, such as record and replay, involve consequent manual effort and do not cope well with changing devices and updated code. The majority of research toward Android testing has been devoted to generating user inputs automatically by means of random [6], [7], systematic [8]–[10], or model-based [11]–[13] input generation. One of the most well-known automated mobile testing tools is Exerciser Monkey [6], commonly known as Monkey, which generates pseudo-random streams of user events such as clicks, touches, or gestures in order to test an application. While Monkey's testing method is reliable and requires little maintenance, its random nature introduces significant waste and cannot provide guarantees on the degree of coverage achieved for a certain number of interactions.

Model-based test generation is the most actively used technique in past research [14]. Although modelling software is not a new area of research, Android introduces new challenges compared to classic sequential execution programs. This makes traditional modelling methods unusable, particularly for modelling callback methods. Mobile applications are highly integrated with the underlying frameworks supported by the mobile operating system, making the execution path and state of an application largely unknown until runtime. Android components such as activity lifecycle methods are called by the Android framework rather than within the application. The framework calls these methods based on the current system state as well as user interactions [15], [16]. Thus, the majority of models generated in past research are obtained from dynamic analysis.

This paper introduces DroidGraph, a framework to generate a comprehensive control flow model of Android applications using traditional static analysis and efficient systematic exploratory tests. Building on our previous work [17], DroidGraph provides a complete model of an Android application, from back-end instruction level code to front-end user interface structures without the need for manual instrumentation. Our previous work using static analysis, manual instrumentation, and control flow model traversal in [17] suggested that DroidGraph can be used to support automated test generation and provide more efficient tests and better coverage of the application under test (AUT). We now augment our static analysis with efficient systematic exploratory tests on the AUT to enhance the model with runtime data and explore the following questions:

RQ1 How much does the inclusion of dynamic analysis enhance the model of the application compared to pure static analysis?

RQ2 Can systematically generated exploratory tests reach more of the application than currently used random inputs?

RQ3 What is the difference in efficiency between systematically and randomly generated exploratory tests?

To study these questions, we apply DroidGraph to 19 diverse apps and show that, on average, our exploratory tests interact with 18% more of the app than the widely used Exerciser Monkey in 345 less interactions. Integrating the dynamic analysis results provided by these tests complements our static analysis, and uncovers on average 51 more components and 49% more interface callback links in the application code. The DroidGraph framework as well as the experimental setup and results underlying these conclusions are made available, and can be reproduced using the code and Docker image provided. [18].

The remainder of the paper is structured as follows: Section 2 provides the background of Android testing, Section 3 introduces DroidGraph, our control flow model for automated testing of Android apps. Section 4 details the experiments supporting the exploration of the above mentioned research questions. Section 5 describes the results of our experiments and discuss our conclusions from these results. Finally, Section 7 provides a summary of related work in the area of model-based Android testing, and Section 8 concludes the paper.

2. Background

Testing is the most established technique for maintaining dependable software in any system. The fast pace and ever expanding market of applications available to users makes it especially important for developers to provide a stable and dependable experience. Since Android applications are built with Java, they benefit from being compatible with Java's unit testing frameworks. Unfortunately, this is not the case for all testing related tools. For example, static analysers built for Java will not work with Android because it is compiled into an Android PacKage (APK) using Dalvik bytecode, instead of Java bytecode. An additional conversion from Dalvik to Java bytecode, or some other intermediary representation, is required to make them compatible [19].

Android introduces many challenges to testing, the most significant being the event based nature of the platform and the high integration that individual apps have with the Android framework [15], [16]. A standard application operates with explicit entry and exit points along with defined paths through the code and interface. Conversely, Android is comprised of standalone units (Activities, Fragments, Services, Broadcasts and Content Receivers) that can be instantiated and referenced in unpredictable ways. This structure lends itself to the interactivity and event based priority of the Android platform. The path taken during execution is largely determined by the Android framework, for example, the order of execution of activity lifecycle methods. Android uses the activity lifecycle to enable developers to ensure that each component functions at every stage of its creation and usage. While the structure of the lifecycle, shown in Figure 1, is clear and concise, its execution is determined by runtime decisions. The activity lifecycle can be entered at any stage; the point at which the application enters is determined by the Android framework [15], [16]. In a static analysis context, one of the challenges of analysing Android applications is that these lifecycle and listener methods appear disconnected from the rest of the application code. Determining where or when they are called is impossible as it is determined at runtime.

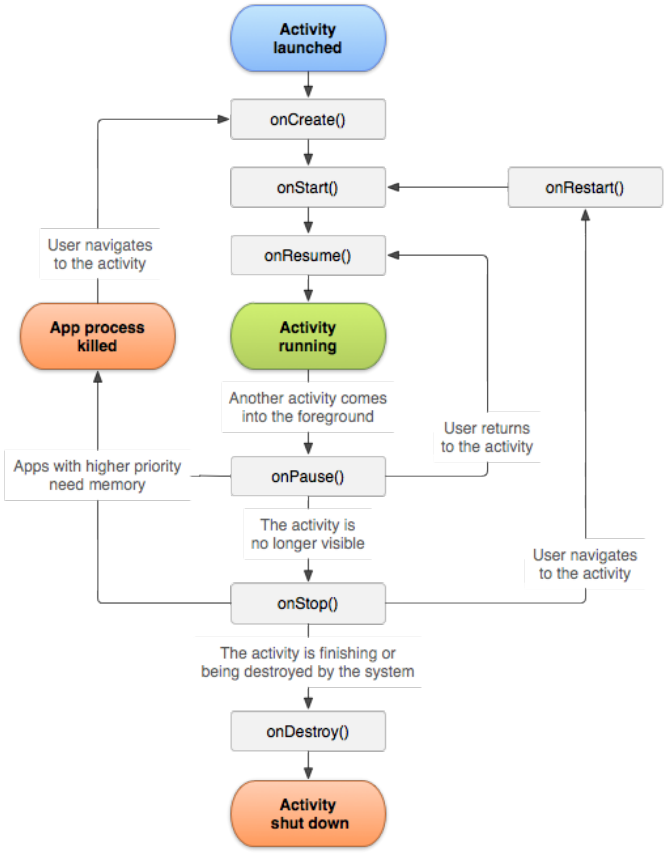

Figure 1: A simplified illustration of the activity lifecycle [15].

Given the challenges that Android poses, a considerable amount of research has been conducted around test automation, particularly in the area of GUI testing. Many of the challenges associated with Android prevent traditional techniques being applied to Android GUI testing. The event-based priority and GUI-central user experience make interface testing particularly important for mobile applications. Despite considerable research already, mobile testing remains largely manual [1]–[5], making it time consuming and more demanding as the application develops. Frameworks and APIs such as UI Automator [20] as well as many others [21]–[25], allow developers to create test scripts for an applications interface. However, these tools require hours of manual scripting to maintain the test suite as the AUT develops, and scripts are

not guaranteed to work on all devices, due to Android device fragmentation [5]. Record and replay tools [26]–[29] such as Espresso Recorder provide the same features as an automation API but they eliminate the need for scripting. This method provides a quicker way of creating tests but does not solve the core problems; the recordings are still time consuming and reliant on the device used to create them.

The majority of research toward Android testing has been devoted to generating user inputs automatically by means of random [6], [7], systematic [8]–[10], or model-based [11]–[13] input generation. Exerciser Monkey [6], commonly known as Monkey, is the most well known random input generator. It creates pseudo-random streams of interactive events, such as clicks or gestures, and executes them on the AUT. Monkey is reliable and requires little maintenance but it's random strategy introduces waste, and the degree of application coverage cannot be guaranteed. Another well-known random input generator, Dynadroid, applies a Frequency Strategy and Biased-Random Strategy, to increase the efficiency of the selected inputs [7]. Dynodroid also supports keyboard input and system notifications but these features contribute to its downfall. For instance, in order to provide system events, Dynodroid needs to instrument the Android framework [7] and in doing so becomes difficult to update and maintain.

Systematic input generation involves dynamically analysing the interface of an application and generating appropriate test inputs as they are found. This method provides effective testing of an application without prior knowledge of the interface structure or the underlying code. Systematic approaches refine random based generators by using sampling strategies and intelligent inputs. Model-based test generation is the most actively used technique in past research [14] and is arguably the most affected by the runtime challenges Android poses as it makes traditional modelling methods unusable, particularly for modelling callback methods. Model-based testing requires a formal model of the application, such as an event flow graph, control flow graph, finite state machine, etc. Once the model has been made, it is used to generate a test suite of device inputs. Due to runtime execution path decisions the majority of models generated in past research are obtained from dynamic analysis.

3. Control Flow Model

This section first defines the model used by DroidGraph to represent Android applications. It then describes how DroidGraph generates this model from an application using both static and dynamic analysis.

3.1 Model definition

Definition 1 (Extended Control Flow Graph)**.** An *extended Control Flow Graph* of a program P with a user interface U is a directed graph $G = (V, E)$ where $V = \{v_1, \ldots, v_n\}$, $n \in \mathbb{N}$, is a set of vertices where v_i represents a program point $p_i \in P$ or an interaction $u_i \in U$ and $E = \{e_1, \ldots, e_m\}$, $m \in \mathbb{N}$, is a set of arcs (directed edges), where $e_i = (v_j, v_k)$ with $v_j, v_k \in V$ and e_i representing the flow of control from either, p_j to p_k under some execution of the program P or u_j to p_k after some user interaction.

DroidGraph's extended control flow graph G is composed of three types of vertices: $V = V_s \cup V_m \cup V_{ui}$, that represent three constituent elements of a mobile application:

- V_s is the set of statement vertices. While testing, we want to focus on the code written for the application rather than the system or library code. Therefore we exclude such statements from V_s. Statements play an important role by revealing application logic, internal method calls, and the detailed control flow of the application.
- V_m is the set of method vertices. Similarly, it only comprises methods developed for the application. The method vertices represent the entry points to a method and lead to the statement vertices within. These vertices can be further split into three types: Android lifecycle methods, user input callback methods, and standard Java methods.
- V_{ui} is the set of interface vertices, representing a GUI element a user can interact with and providing input to the application.

Not all control flow edges can be included in our graph as they originate from outside of the application code. For example, Activity lifecycle methods are called by the Android framework based on the current state of the system or application, meaning they have no explicit call within the application code. This causes these lifecycle methods to appear as disconnected clusters in the graph. The control flow to and from these clusters is determined at runtime.

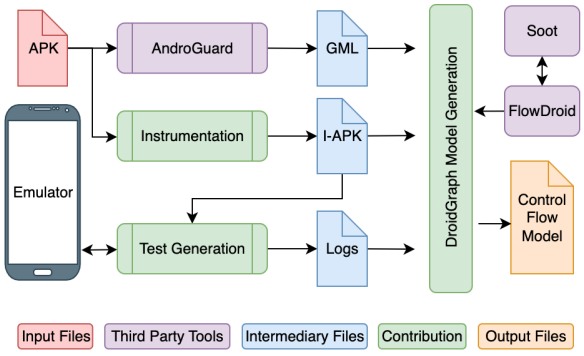

Figure 2: DroidGraph component structure.

DroidGraph is composed of several tools and components that work together to create the extended control flow model of an Android application. Figure 2 provides an overview of all the processes involved and how they interact. The AUT is provided as input, in the form of an APK file, to Droidgraph's instrumentation module (Section 3-B) and AndroGuard (Section 3-C2). They produce an instrumented APK file and a GML call graph file, respectively. The instrumented APK is then run with some tests (e.g., exploratory tests generated by DroidGraph, see Section 3-D) which outputs log files containing the actions performed by the tests and any

application components found. Using the GML call graph, the instrumented APK and the exploratory test logs, DroidGraph generates the extended control flow model (Section 3-C).

3.2 Instrumentation

In order to discover links between interface elements and callback methods at runtime, and to calculate both interface- and method-coverage in our experiments, we instrument the AUT using Soot [30]. Soot began as a Java optimisation framework, but by now, researchers and practitioners use Soot to analyse, instrument, optimise and visualise Java and Android applications. Soot's primary functionality is inter-procedural and intra-procedural analysis. It takes as input Java source- and byte-code, or more recently Android byte-code, and transforms it into an intermediary representation known as Jimple—a typed three address code, for easier analysis [30]. Our instrumentation involves adding a log statement to each method of the application (excluding system and library methods), containing the following data:

- Log tag: The `<CONTROL>` tag for any method with a parameter of type `android.view.View`. The `<METHOD>` tag for all other methods. Tags are used for filtering logs during and after execution.
- Method name: The name of the method is used for 1) linking a callback method to an interface view ID, and 2) calculating method coverage achieved by tests.
- View ID: Callback methods take a parameter of type `android.view.View` as input. This is the view object that the user interacts with, and subsequently calls a callback method. If this view object is present, we log the call `view.getId()`. This enables us to identify which UI element interaction triggered the callback method during runtime.

3.3 Static Model Generation

3.3.1 **FlowDroid**

Although Soot has the ability to analyse Android byte-code, it's ability to fully analyse an Android application is limited. It has no knowledge of the Android lifecycle or the Android framework, and therefore cannot analyse or categorise Android-specific callback methods. Unlike Java, Android applications have multiple entry points in the form of callback methods which prevent Soot from creating a call graph. To solve these issues we use FlowDroid [31], a fully context-, field-, object- and flow-sensitive taint analysis tool which considers the Android application lifecycle, while also featuring a novel, particularly precise variant of an on-demand alias analysis. It was designed to analyse applications, and alert users of malicious data flows, or as a malware detection tool which could determine if a leak was a policy violation. Despite its main purpose, FlowDroid extends Soot, adding knowledge of the activity lifecycle, and the ability to categorise callback methods; it can thus be used as a powerful static analysis framework.

Our extended control flow graph is composed of three types of vertices, that represent three constituent elements of a mobile application, as defined above. Each type of vertex is collected in a specific way from the initial static analysis:

- **Statement** vertices are gathered from FlowDroid `UnitGraph` objects created for each method in the AUT. Each `UnitGraph` contains a unit chain detailing all the Jimple statements and their execution order within the method. We also check for inter-procedural calls within each statement of the method.
- **Method** vertices are created by retrieving all the methods, for each class, identified by FlowDroid in the AUT. We limit our control flow model to internal components by filtering external library methods such as AndroidX. Before adding methods to the model, we categorise them as activity lifecycle, input listener, or standard Java methods.
- **Interface** controls are identified in FlowDroid by parsing the Android XML layout files included in the compiled APK. FlowDroid is capable of identifying controls and their XML attributes when declared in XML layouts but cannot identify controls that have been declared in Java. We attempt to link controls found to their associated listener methods by searching the application code for instances of a `setListener` method call on a variable containing the resource ID of an interface control. However, due to the number of possible set listener methods, limitations in tracking variable values, and the ambiguity of a developers coding style, preferences and techniques, these links are rarely populated from the static analysis only.

3.3.2 **AndroGuard**

Soot is built for analysing Java applications and FlowDroid's main objective is taint analysis, neither is focused on the static analysis of Android applications. This has led to some features becoming outdated and unreliable, for example, the identification of Android fragments. Since FlowDroid was originally released, Android fragments have seen the release of AndroidX, with a newly built API for fragment development. New versions of FlowDroid have yet to add compatibility for fragments built using AndroidX. Consequently, the call graph generated by FlowDroid is consistently missing fragment classes.

AndroGuard, is a python-based tool used for reverse engineering Android apps [32], [33], that can produce an acceptable call graph of an Android application. Using AndroGuard, we generate a call graph for the instrumented AUT using the Graph Modelling Language (GML) format. We import the GML graph, filter external methods already classified by AndroGuard, and augment DroidGraph's model with the missing edges between method vertices.

3.4 Model Enhancement through Dynamic Analysis

There are several control flow components and links missing in the static analysis of Android applications, for example, controls declared within the Java code. While some of these components and links could be retrieved through static analysis, it would require far more development and future

maintenance to overcome difficulties such as evolving APIs and developer coding methods. The result would be a system that is not dependable and always behind new trends. Relying on dynamic analysis, through execution of the instrumented application is simple, dependable, and more likely to work with future generations of Android with little to no maintenance.

We explore the AUT using Appium, an open-source project and ecosystem of related software, designed to facilitate UI automation of many app platforms, including Android [24], [34]. Our exploration mimics an enhanced depth-first search (DFS) on the application GUI. Appium is highly integrated with Android's UI-Automator and provides detailed knowledge of interface elements currently available to interact with, and their locations on the screen. This allows us to avoid interactions that lack corresponding interface element and therefore have no triggered behaviour. Our systematic traversal avoids unnecessary repetitive interactions by tracking interface controls and the number of times they have been used. We also flag interactions that did not lead to further Activities or Fragments (leaf nodes) so that they are not repeated.

Throughout our exploratory tests we monitor the application logs for instances of our instrumentation, discussed in section 3-B. When a component that is missing from our model is found it is added. When a log with the tag `<CONTROL>` is found, we augment our model to include the link between the vertices representing the interface control and listener method found in the log. Droidgraph's exploration of the AUT provides reliable (not subject to randomness) tests to sustain the dynamic analysis phase of model building and complement the static analysis. It does not aim at generating the state of the art, efficient test suites, simply at providing a more reliable alternative to often used random tests such as those generated by Exerciser Monkey.

4. Experiments

This section describes the experiments we conducted. First, it describes the research questions we explored. Then it introduces the applications we used as benchmarks. Finally, it describes the protocols and metrics we used to investigate the research questions. The experimental setup and results are made available online [18].

4.1 Research Questions

In order to understand the contribution of DroidGraph we explore the following 3 research questions:

RQ1 How much does the inclusion of dynamic analysis enhance the model of the application compared to pure static analysis?
This research question explores the contribution of the exploratory tests used by DroidGraph to the final model the framework produced. It highlights the elements of the graph that would have been missed if DroidGraph only relied on static analysis.

RQ2 Can systematically generated exploratory tests reach more of the application than currently used random inputs?
This research question compares the exploratory power of the inputs generated by DroidGraph and of those generated by Exerciser Monkey. It explores how much of the applications' interface the tests interact with, and how much of their code they execute, reflecting how complete of a model they can lead to.

RQ3 What is the difference in efficiency between systematically and randomly generated exploratory tests?
This research question studies how quickly the tests generated by DroidGraph and Monkey explore the apps. Even if similar coverage is achieved by both tools, achieving this coverage faster, i.e., using fewer interactions, would lead to faster modelling of the application.

4.2 Subject applications

In order to study the above research questions, we need to apply both DroidGraph and Exerciser Monkey to a diverse set of Android applications. We randomly selected applications from the F-Droid market place [35], which contains over 4000 apps. We select 1 application per category found in F-Droid in order to represent the different types of applications developed in practice. Before choosing the apps we filtered unsuitable apps based on 3 criteria:

1) apps in the games category, as they often include game development frameworks such as unity, which we do not support.
2) apps with an Android SDK less than 16 (too old) or greater than 29 (not yet supported by FlowDroid).
3) apps that have not been maintained, i.e., not updated within the last 10 years.

We also include 3 apps used in previous research [17] to show that previous results traversing the control flow model are consistent with our new approach. Table I shows the list of apps used in our experiments. All corresponding APK files are made available in [18].

4.3 Experimental procedure

In order to investigate RQ1, we first applied DroidGraph to all the applications in Table I. During the execution, we recorded which elements in the graph (vertices and edges) were collected from the dynamic analysis results obtained from exploratory tests. We also record the total size of the final model of each application in order to understand the respective contribution of the static analysis and of the dynamic analysis to the model.

In order to investigate RQ2 and RQ3, we also generated exploratory tests for all applications using Exerciser Monkey. As DroidGraph only performs click interactions when exploring the application, we executed Monkey under two settings: limiting it only to the click interaction (Monkey Click), and allowing it to use all interactions it supports (Monkey All). In order to account for the random nature of Monkey, we ran both settings of the tool 10 times for each application. DroidGraph, however, is deterministic in its test generation, and was thus only run once per application. In order to achieve

TABLE I: Applications used in the experiment.

Category	App name	Version code
Connectivity	Webradio	5
Development	Git Quick Reference	7
Graphics	Pixel Filter	24
Internet	Ad-Free	41
Money	Loyalty Card Keychain	39
Multimedia	Camera Roll	36
Navigation	GPSTest	18093
Phone & SMS	Contact Book	1
Reading	Drinks	32
Science & Education	Activity Lifecycle	1
Science & Education	Timetable	17
Security	PIN Mnemonic	6
Sports & Health	Wine Cellar	4
System	Volume Control	32
System	Simple Explorer	67
Theming	Battery Live	13
Time	MoClock	4
Time	Simple Todo	5
Writing	Taskkeeper	6

a fair comparison, we ran all tools with the same budget of 500 interactions.

We then measured the overall coverage achieved by each of the generated tests, as well as the coverage achieved after each individual interaction in each test. This was achieved by parsing the logs produced by the instrumentation described in Section 3-B. Individual interaction coverage is only calculated for the Monkey click only interaction tests due to a limitation in Monkey's output logs. Comparison of the overall coverage achieved for each application by exploratory tests generated by DroidGraph and Monkey serves to answer RQ1, while comparing the rate at which coverage increases during the tests, as well as the number of interactions required to achieve the final coverage answers RQ3.

All experiments and results can be reproduced using the code and Docker image provided [18].

5. Results

This section describes the results we obtained from our experiments and how they relate to our research questions.

5.1 RQ1 How much does the inclusion of dynamic analysis enhance the model of the application?

Table II shows the number of vertices and interface-callback edges discovered and added to our extended control flow model thanks to DroidGraph's systematic exploratory tests. On average 52 components are found by the exploratory tests. The majority of these are interface elements declared in Java code rather than Android XML layout files. The largest impact is, on average, 49% of the missing interface-callback method links, that cannot be retrieved through static analysis, are discovered by our exploratory tests. These results show that while static analysis is effective, Android applications require a combination of static and dynamic analysis and that our efficient systematic exploratory tests are effective in complementing our static analysis in creating a comprehensive control flow model of Android applications.

TABLE II: Dynamic Analysis Model Enhancements.

App Name	Added elements		
	Vertices	Interface-Callback Edges	
	#	#	%
Ad-Free	89	89	18
Timetable	298	30	26
Activity Lifecycle	0	24	100
Battery Live	29	32	45
Camera Roll	18	35	19
Contact Book	7	48	67
Drinks	5	7	9
Git Quick Reference	4	7	70
GPSTest	432	23	40
Loyalty Card Keychain	20	38	42
MoClock	3	11	91
PIN Mnemonic	13	41	41
Pixel Filter	1	2	13
Simple Explorer	12	15	34
Simple Todo	13	22	49
Taskkeeper	8	15	83
Volume Control	5	8	50
Webradio	6	7	70
Wine Cellar	9	13	62

5.2 RQ2 Can systematically generated exploratory tests discover more of the application than currently used random tests?

Tables III and IV, and Figures 3 and 4 show a comparison of the coverage achieved by DroidGraph's systematic exploratory tests, as well as the average coverage achieve by both settings of Monkey over the 10 runs. In Tables III and IV, for each application, the maximum coverage achieved for this app is shown in bold.

DroidGraph's exploratory tests, on average, interact with 18% more of the AUT than Monkey. While Monkey is comparable in some apps, such as Battery Live, our approach is higher than Monkey's average coverage. Monkey delivered better coverage in only 2 test apps. For example, Pixel Filter achieves the highest coverage in the Monkey all interaction tests. This indicates that Pixel Filter is less click orientated, preventing our exploratory tests and the Monkey click only interaction tests from succeeding.

While our exploratory tests are successful in achieving better coverage of the application. The overall coverage achieved is still low in many of the apps. There are many reasons for low interface coverage. Interface elements are not always visible

TABLE III: Interface coverage (%) achieved by each approach

App Name	Exploratory Tests	Monkey Click			Monkey All		
		Avg	Min	Max	Avg	Min	Max
Pixel Filter	3.45	1.72	0.0	3.45	5.52	0.0	**6.9**
MoClock	**38.1**	20.48	9.52	**38.1**	16.67	4.76	28.57
Taskkeeper	**20.75**	12.64	9.43	18.87	7.55	3.77	11.32
Simple Explorer	**19.35**	13.23	9.68	16.13	10.97	3.23	16.13
Simple Todo	**26.53**	15.71	10.2	20.41	10.41	4.08	20.41
Activity Lifecycle	**66.67**	26.67	2.78	44.44	15.56	8.33	22.22
Webradio	**23.33**	4.0	3.33	6.67	8.0	3.33	10.0
Timetable	55.7	51.21	43.62	**59.73**	29.5	0.0	48.32
Drinks	**14.71**	10.29	0.0	**14.71**	7.94	2.94	**14.71**
Battery Live	21.57	19.02	5.88	**33.33**	7.06	1.96	11.76
PIN Mnemonic	14.29	20.45	15.18	**24.11**	2.86	0.89	4.46
GPSTest	**94.74**	8.42	5.26	10.53	10.53	10.53	10.53
Volume Control	10.0	4.2	4.0	**12.0**	2.8	0.0	8.0
Contact Book	**43.9**	14.27	9.76	19.51	6.22	0.0	13.41
Git Quick Reference	**35.0**	10.0	10.0	10.0	6.0	5.0	10.0
Ad-Free	**60.67**	15.73	5.62	46.07	34.04	0.0	46.07
Loyalty Card Keychain	**14.68**	8.72	9.17	**14.68**	5.6	3.67	10.09
Camera Roll	**13.73**	3.14	1.96	4.9	4.12	1.96	7.84
Wine Cellar	**20.31**	0.0	0.0	0.0	0.0	0.0	0.0

TABLE IV: Method coverage (%) achieved by each approach

App Name	Exploratory Tests	Monkey Click			Monkey All		
		Avg	Min	Max	Avg	Min	Max
Pixel Filter	55.32	56.06	55.32	**58.51**	54.04	52.13	56.38
MoClock	62.5	54.0	32.5	**70.0**	40.0	30.0	55.0
Taskkeeper	43.15	35.21	26.71	**44.52**	31.37	30.14	32.88
Simple Explorer	**28.26**	22.55	18.84	23.69	23.24	20.84	27.59
Simple Todo	**43.05**	36.14	33.39	41.53	32.15	25.76	39.49
Activity Lifecycle	**96.35**	55.11	18.98	81.75	44.16	33.58	55.47
Webradio	**53.79**	48.86	36.36	52.27	48.48	42.42	52.27
Timetable	10.55	5.4	8.26	**11.5**	6.27	0.0	9.96
Drinks	9.77	9.37	8.87	**9.83**	9.09	8.36	9.75
Battery Live	**85.21**	80.36	69.23	85.8	62.43	24.85	79.29
PIN Mnemonic	44.54	71.51	65.55	**78.15**	28.07	23.53	36.13
GPSTest	**17.86**	5.79	5.31	6.14	5.99	5.56	6.23
Volume Control	**70.34**	40.84	49.43	63.12	46.12	41.44	49.81
Contact Book	18.74	7.42	3.84	**39.17**	7.47	2.92	28.26
Git Quick Reference	**20.05**	15.24	15.24	15.24	15.17	15.06	15.42
Ad-Free	**4.99**	3.21	2.59	3.83	3.0	0.0	4.31
Loyalty Card Keychain	**38.42**	18.43	17.64	30.89	10.84	9.72	15.63
Camera Roll	**7.48**	3.39	3.33	3.68	4.2	3.29	6.31
Wine Cellar	**31.2**	20.4	20.4	20.4	20.4	20.4	20.4

until a specific action is executed in the app. For example, a submit button that will only appear after the user enters text into a text field. Some applications require content, such as contact apps, where a portion of the interface is only visible after the user has added a contact.

5.3 RQ3 What is the difference in efficiency between systematically and randomly generated exploratory tests?

Figures 5 and 6 respectively show the interface and method coverage achieved by DroidGraph's tests after each interaction on the application Simple Explorer[1], as well as the minimum, maximum, and average coverage achieved by the 10 runs of Monkey (click interactions only) on the app after each interaction. Table V shows the number of interactions required by

[1]Results for all applications are available in the paper's companion repository [18] and left out for space concerns.

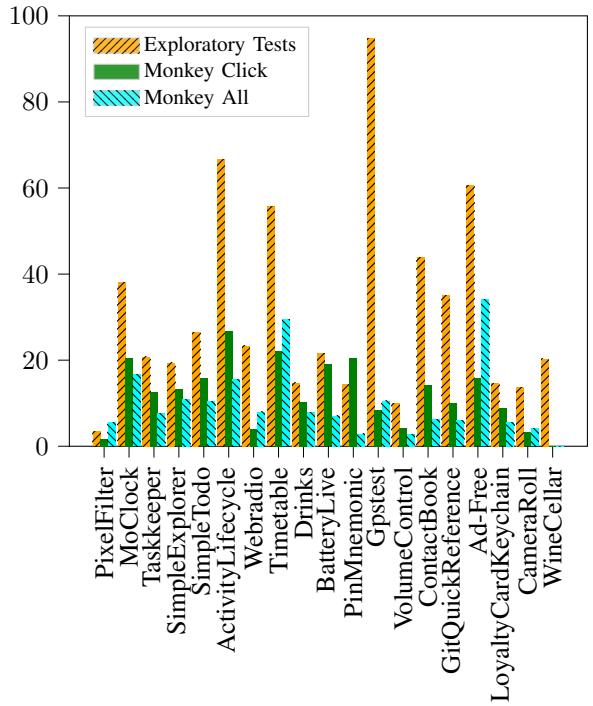

Figure 3: Interface coverage (%) achieved by Exerciser Monkey and by our method

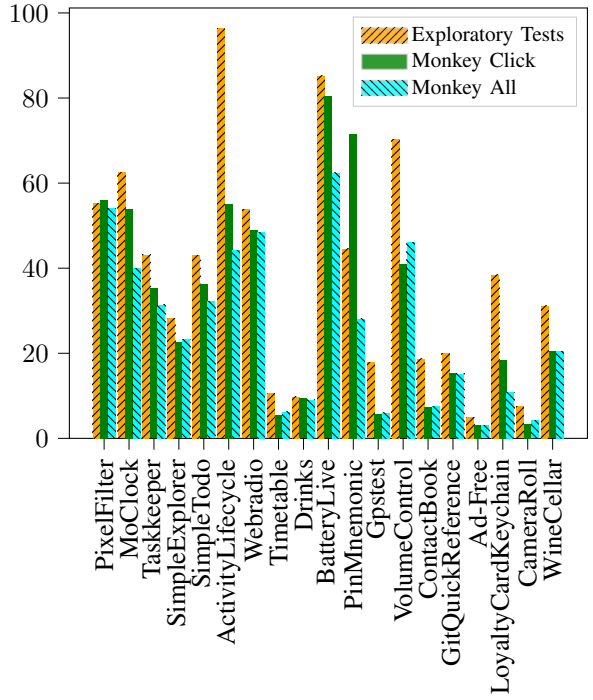

Figure 4: Method coverage (%) achieved by Exerciser Monkey and by our method

TABLE V: Number of interactions required by each approach to achieve maximum coverage of each application.

App Name	Exploratory Tests	Monkey Click
Webradio	25	81
Git Quick Reference	71	496
Pixel Filter	3	413
Ad-Free	36	437
Loyalty Card Keychain	31	497
Camera Roll	37	499
GPSTest	23	1
Contact Book	73	493
Drinks	16	488
Activity Lifecycle	90	492
Timetable	5	41
PIN Mnemonic	37	493
Wine Cellar	24	1
Volume Control	20	429
Simple Explorer	27	461
Battery Live	77	498
MoClock	10	489
Simple Todo	18	496
Taskkeeper	109	496

each approach to achieve its final coverage on each application. These results show that while Monkey can achieve coverage comparable to DroidGraph's exploratory tests in some cases it is far less efficient in the number of interactions required to achieve it. More interactions required means longer testing times as well as a lot of wasted time performing pointless interactions. DroidGraph's exploratory tests can achieve higher coverage of the AUT in, on average, 345 less interactions. Our exploratory tests execute less interactions than all of Monkey's tests, except for GPSTest and Wine Cellar, which reach their final coverage in only one interaction due to Monkey's inability to cover any of the application.

6. Threats to Validity

This work's validity is subject to different threats [36] that this section details, as well as how we addressed them.

Construct validity. The metrics we use to assess DroidGraph may not be suitable. Although, by construction, the elements added to the model in the dynamic analysis phase are correct (we can not observe a UI element or callback in the logs if it does not exist), the correctness of the model is hard to assess. Similarly, the completeness of the graph can not be assessed due to the complexity of the applications and the lack of ground truth. Instead, we used the coverage of the tests used in the dynamic analysis, a standard metric in testing, as a proxy metric. Indeed, any element covered by the tests will be added to the model, so full coverage would ensure a complete model. Furthermore, by directly counting the number of elements added to the model, we confirm that the dynamic analysis provides a more complete model than the static analysis alone.

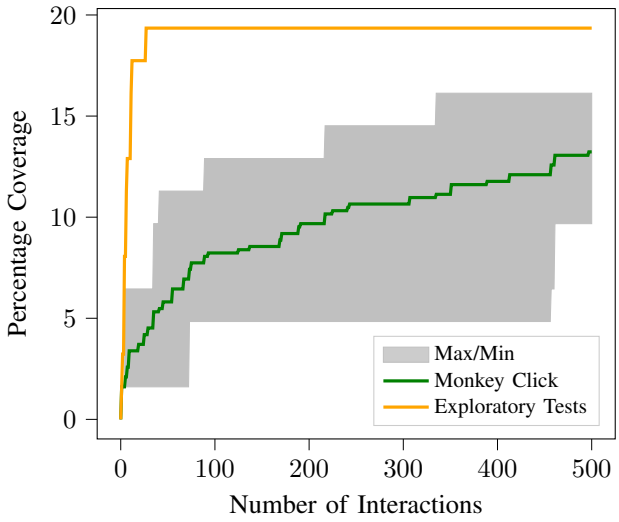

Figure 5: Interface coverage (%) achieved per interaction by Exerciser Monkey and by our exploratory test on the app Simple Explorer

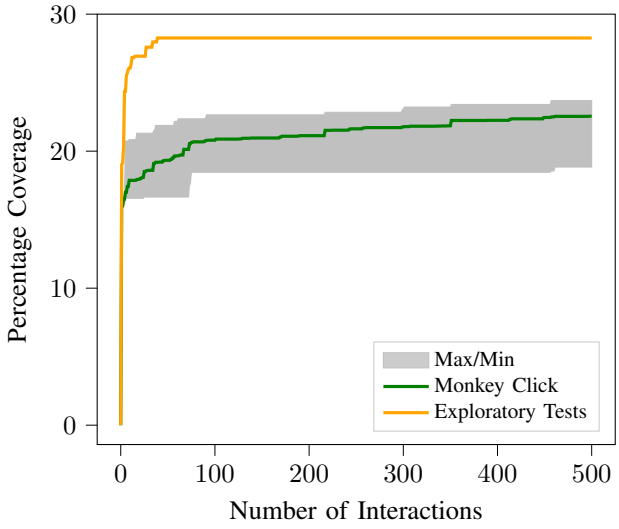

Figure 6: Method coverage (%) achieved per interaction by Exerciser Monkey and by our exploratory test on the app Simple Explorer

Conclusion validity. Exerciser Monkey relies on randomness to generate tests, which could impact results. To account for this randomness when comparing the systematic exploratory tests used in this work to augment the model through dynamic analysis with the widely used Monkey generated tests, we generated 10 test suites with Exerciser Monkey for each application in our experiments.

Internal validity. Results presented in this work could be the consequence of a faulty implementation. To mitigate this threat, we carefully tested the implementation of DroidGraph, and manually verified results where possible.

External validity. The approach adopted by Droidgraph may

not generalise. To mitigate this threat, we assessed it over a set of diverse applications that span multiple domains. We also make the implementation of Droidgraph available, allowing further investigations in the future.

7. Related work

This section outlines previous work in the areas of systematic and model-based Android testing.

Systematic input generation involves dynamically analysing the interface of an application and generating appropriate test inputs as they are found. This method proves effective in testing an application without prior knowledge of the interface structure or the underlying code. A good example of this technique is AndroidRipper [10], which maintains a state machine model as it systematically traverses the AUT's UI, called a GUI Tree. This GUI Tree model contains the set of GUI states and state transitions encountered during the ripping process. However, due to a lack of maintenance, AndroidRipper is unusable on newer Android versions. Automatic Android App Explorer (A3E) [8] implements two forms of exploration, Depth-First Exploration and Targeted Exploration. Depth-First Exploration uses a similar technique to that of AndroidRipper; the framework enters the application at a specified location and explores the application systematically. The aim of this approach is to explore the application in the same manner as a user i.e. by clicking on views to move through the application followed by pressing the back button. Targeted Exploration, conversely, is a directed approach that uses static byte-code analysis to extract a Static Activity Transition Graph, which is then explored systematically while the app runs on the phone. Like AndroidRipper, A3E is poorly maintained and the tool contains functional bugs, for example buttons with labels containing special characters cannot be clicked. Due to its outdated implementation and the need to run the target app under its instrumentation, A3E can cause apps to crash, preventing testing [37]. Another approach, CrashScope [9], uses static analysis to identify contextual features within activities such as network use. Having identified these features, CrashScope can then test the application in different states, for example with the network on or off. Once static analysis is complete, the tool dynamically extracts the GUI of each screen to identify any clickable, long clickable or text input views, with the intrinsic goal of triggering crashes. While CrashScope shows great potential as an automated testing tool, the main focus of the tool is on the generation of readable crash reports rather than improved effectiveness in detecting bugs. A well-known search-based approach is Sapienz [38], which employs a multi-objective search, combining random fuzzing, systematic and search-based exploration, string seeding and multi-level instrumentation.

Approaches employing systematic input generation show promise, with the aforementioned tools showing varying degrees of improvement in areas such as coverage, input redundancy and bug detection [8]–[10], [37], [38]. However, a significant limitation of this technique is its inability to guarantee complete application coverage. Model-based testing requires a formal model of the application, such as an event flow graph, control flow graph, finite state machine, etc., which is used to generate a test suite of device inputs. Many systematic approaches can be seen as model-based because they create a model while systematically testing the application. Stoat [12] and MobiGuitar [13] model the AUT as a finite state machine (FSM). Stoat uses both static and dynamic analysis, enhanced by a weighted UI exploration strategy, to explore the AUT's behaviours and construct a stochastic FSM [12]. MobiGuitar uses an enhanced version of Android Ripper, dynamically traversing the application in a breadth-first fashion to generate the FSM [13]. Stoat sets itself apart by using system events within tests. Instead of trying to model system events, it randomly injects various system-level events into its UI tests [12]. This simulates the random nature of state changes in a real environment e.g. when notifications are received. Stoat, however, is ineffective with regular gestures (e.g., Pinch Zoom, Move) and specific input data formats [12], while MobiGuitar's reliance on AndroidRipper means it faces the same pitfalls. Another approach to model-based testing is implemented in SwiftHand, which uses machine learning to generate and improve a model of the AUT during test execution. While initially employing a systematic approach, the learned model is used to generate test inputs that visit unexplored states of the AUT. Similar to CrashScope, the main focus of SwiftHand is not on improved fault detection in Android applications. Instead SwiftHand focuses on coverage of the Android AUT and on the reduction of application restarts in automated testing. Application restarts are required by all automatic exploration algorithms, for the exploration of additional states reachable from the initial state, but unlike other learning-based techniques, SwiftHand minimises the number of restarts by attempting to reach unexplored states using only user inputs. SwiftHand's experimental results show that it can achieve significantly better coverage than traditional random testing in a given time budget, while also reaching peak coverage. However, while SwiftHand excels in reaching coverage goals, the tool cannot handle text input, does not cater for system events and has not been proven capable of handling industry applications.

8. Conclusions and Future Work

The slow and maintenance heavy task of creating test suites coupled with the fast paced mobile industry has increased the need to eliminate manual testing methods as much as possible. Model-based test generation is one of the most common and successful approaches to support automated test generation. Understanding and modelling the test space is integral to producing comprehensive, dependable and effective tests.

This paper introduced DroidGraph, a framework to generate a comprehensive control flow model of Android applications using traditional static analysis and efficient systematic exploratory tests. DroidGraph provides a detailed model of an Android application, from back-end method statements to front-end user interface structures. This model can be used to support automated test generation. We apply DroidGraph

to 19 apps spanning 15 categories, and interact with 18% more of the app using our efficient exploratory tests compared with commonly used random exploration. We also achieve this increase with 345 less interactions. Integrating the dynamic analysis results provided by these tests complements our static analysis, and uncovers on average 51 more components and 49% more interface callback links in the application code.
Future work will explore the potential of Droidgraph when used with more complex tests that include more interaction types and further explore the application's behaviour in order to evaluate how complete of a model can be created. Furthermore, we also plan to study the use of Droidgraph's application model to generate strong and efficient test suites for Android applications.

Acknowledgement

This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094_P2 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero - the Science Foundation Ireland Research Centre for Software (www.lero.ie)

References

[1] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real challenges in mobile app development," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 15–24.

[2] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–10.

[3] M. Linares-Vásquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?" in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 613–622.

[4] F. Pecorelli, G. Catolino, F. Ferrucci, A. De Lucia, and F. Palomba, "Software testing and android applications: a large-scale empirical study," *Empirical Software Engineering*, vol. 27, no. 2, p. 31, 2022.

[5] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk, "Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing," in *ICSME*, 2017, pp. 399–410.

[6] Android Developers, "UI/Application Exerciser Monkey," https://developer.android.com/studio/test/monkey.html, 2012.

[7] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *FSE*, 2013, pp. 224–234.

[8] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 641–660.

[9] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Crashscope: A practical tool for automated testing of android applications," in *ICSE*, 2017, pp. 15–18.

[10] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," in *ASE*, 2012, pp. 258–261.

[11] W. Choi, G. Necula, and K. Sen, "Guided GUI testing of Android apps with minimal restart and approximate learning," *SIGPLAN Not.*, vol. 48, no. 10, pp. 623–640, 2013.

[12] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of Android apps," in *FSE*, 2017, pp. 245–256.

[13] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated model-based testing of mobile apps," *IEEE software*, vol. 32, no. 5, pp. 53–59, 2014.

[14] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of android apps: A systematic literature review," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 45–66, 2018.

[15] Android Developers, "The Activity Lifecycle," https://developer.android.com/guide/components/activities/activity-lifecycle, 2012.

[16] D. Amalfitano, A. R. Fasolino, P. Tramontana, and B. Robbins, "Testing android mobile applications: Challenges, strategies, and approaches," in *Advances in Computers*. Elsevier, 2013, vol. 89, pp. 1–52.

[17] J. Doyle, T. Saber, P. Arcaini, and A. Ventresque, "Improving mobile user interface testing with model driven monkey search," in *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2021, pp. 138–145.

[18] J. Doyle, T. Laurent, and A. Ventresque, "Supplementary material for the paper "Modelling Android applications with static analysis and systematic exploratory testing"," https://github.com/jordan2doyle1/DSA-2023-DroidGraph, 2023.

[19] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: converting android dalvik bytecode to jimple for static analysis with soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, 2012, pp. 27–38.

[20] Android Developers, "UI Automator," https://developer.android.com/training/testing/ui-automator, 2014.

[21] ——, "Monkeyrunner," https://developer.android.com/studio/test/monkeyrunner/index.html, 2015.

[22] D. T. Milano, "Androidviewclient," https://github.com/dtmilano/AndroidViewClient, 2016.

[23] Android Developers, "Espresso," https://developer.android.com/training/testing/espresso, 2016.

[24] N. Verma, *Mobile Test Automation With Appium*. Packt Publishing Ltd, 2017.

[25] H. Zadgaonkar, *Robotium automated testing for Android*. Packt Publishing Birmingham, 2013.

[26] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso, "Barista: A technique for recording, encoding, and running platform independent Android tests," in *ICST*, 2017, pp. 149–160.

[27] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for Android," in *ICSE*, 2013, pp. 72–81.

[28] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi, "Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem," in *ISPASS*, 2015, pp. 215–224.

[29] Android Developers, "Espresso test recorder," https://developer.android.com/studio/test/espresso-test-recorder.html, 2016.

[30] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot framework for Java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, vol. 15, 2011, p. 35.

[31] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[32] S. B. Anthony Desnos, Geoffroy Gueguen, "AndroGuard Documentation," https://androguard.readthedocs.io/en/latest/#, 2012.

[33] A. Desnos and G. Gueguen, "Android: From reversing to decompilation," *Proc. of Black Hat Abu Dhabi*, vol. 1, pp. 1–24, 2011.

[34] OpenJS Foundation, "Appium documentation," http://appium.io/docs/en/2.0/, 2012.

[35] F-Droid Limited and Contributors, "F-Droid - Free and Open Source Android App Repository," https://f-droid.org/en/, 2010.

[36] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[37] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "An empirical study of Android test generation tools in industrial cases," in *ASE*, 2018, pp. 738–748.

[38] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 94–105.