PADRAIG: Precise Android Automated Input Generation

Jordan Doyle, Thomas Laurent, and Anthony Ventresque
SFI Lero &, *School of Computer Science and Statistics*, *Trinity College Dublin*, Dublin, Ireland
doylej51@tcd.ie, tlaurent@tcd.ie, anthony.ventresque@tcd.ie
0009-0001-3448-351X, 0000-0002-0953-774X, 0000-0003-2064-1238

Abstract—Android automated test input generation has been a highly researched topic for over a decade and has shown promising results with a variety of approaches. Random input generation is commonly used and the easiest to maintain, but ultimately inefficient. Systematic and search-based approaches produce effective tests but require a disproportionally large generation runtime. Model-based approaches have the additional overhead of modelling the application under test (AUT) but they result in a faster test generation.

In this paper we present Precise AnDRoid Automated Input Generation (*PADRAIG*), a model-based test input generation framework that uses a detailed control flow model of the AUT to generate tests that can achieve higher line coverage, with a lower test generation runtime than the state of the art. We compare the line coverage achieved, and the generation runtime of *PADRAIG* against 3 state of the art tools, each of which uses a different test input generation technique. Our results, using 19 randomly selected Android apps from the F-Droid application store, show that *PADRAIG* achieves, on average, 16% more coverage of the AUT than the state of the art and it can generate tests with, on average, 84% less runtime.

Keywords–Android, Modelling, Automated, Testing, Test Generation

I. Introduction

Mobile applications can be seen in all aspects of modern life. In 2022, the global smartphone penetration rate was estimated at 68% of a global population of around 7.4 billion [1]. With this massive user base comes a large number of available apps and services with over 3 million apps, representing a revenue of approximately 10.4 billion USD in the Google Play Store alone [2]. With their increase in popularity and economic growth it is particularly important to ensure that mobile applications are adequately tested to ensure they are reliable. For instance, a sudden loss of data, or an inconsistent user experience can result in negative feedback and reviews which, coupled with the wide range of applications available, can result in a loss of users and therefore revenue [3].

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)

Despite years of research [4]–[14], Android testing is still largely performed manually. Automated test input generation presents a promising approach to alleviating that manual effort and has shown encouraging results in past research using different approaches and achieving varying levels of success. Random [15]–[19] test input generation is a commonly used and easy to maintain technique but is ultimately inefficient, resulting in a large, ineffective tests. Systematic [20]–[22] and search-based [23]–[27] techniques show promising results but require an excessive generation runtime. Model-based [28]–[31] test input generation is a widely used technique in research. Understanding and modelling the application under test (AUT) does create additional initial overhead, but results in a faster test generation and a more effective tests [14].

This paper introduces Precise AnDRoid Automated Input Generation (*PADRAIG*), a model-based framework for automated test input generation that uses a detailed control flow model of an Android application, to generate effective test inputs that achieve high coverage with a fast test generation runtime. *PADRAIG* is compared against 3 state of the art tools, Monkey [15], search-based test generation framework for Android apps with support for multi-objective generation (STGFA-SMOG) [27], and STOchastic model App Tester (Stoat) [30], each of which uses a random, search- and model-based test input generation technique, respectively. Our comparison of *PADRAIG* with the state of the art is used to explore the following research questions:

RQ1 Can *PADRAIG* generate tests that cover more of the application than state of the art tools that use a variety of generation techniques?

RQ2 How long do the search- and model-based techniques take to generate tests?

To study these questions, Monkey [15], STGFA-SMOG [27], Stoat [30], and our proposed approach *PADRAIG*, are applied to 19 randomly selected diverse apps. A comparison of *PADRAIG* and the state of the art shows that *PADRAIG* covers, on average, 16% more of the AUT and demonstrates a statistically *large* (see Section IV-D) improvement over the state of the art. *PADRAIG* is also deterministic and generates tests with an average of 84% less runtime compared with alternative search- and model-based approaches, while still increasing line coverage. *PADRAIG* as well as the experimental setup and results underlying these conclusions are made available,

and can be reproduced using the code and Docker images provided [32].

The remainder of this paper is structured as follows: Section II introduces the detailed control flow model used to guide *PADRAIG*s test generation and section III outlines the implementation of our proposed approach. Section IV details the experiments supporting the exploration of the above mentioned research questions. Section V describes the results of our experiments and discusses these results. Section VI provides a summary of related work in the area of Android test input generation. Finally, Section VII outlines possible threats to the validity of this research and Section IX concludes the paper.

II. Extended Control Flow Model

PADRAIG uses an extended control flow graph (eCFG) model created by DroidGraph and presented in our previous research [33], to generate precise test inputs. This section first defines the model, including its structure, and then briefly describes how DroidGraph generates this model from an application using both static and dynamic analysis.

1. Model definition

Definition 1 (Extended Control Flow Graph)**.** An *eCFG* 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 $(v_j, v_k) \in V^2$ and e_i represents 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.

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

- V_s is a set of statement vertices. While testing, the focus should be on the code written for the application rather than system or library code. Therefore, such statements are excluded 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 a 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 a set of interface vertices, representing a user interface (UI) element a user can interact with and providing input to the application.

Figure 1 shows an example subset of the eCFG representing an Android application: its internal structure (methods and statements) and the user interactions it contains. This example contains two UI controls, 'StartB' and 'SayHello', connected to their listener callback methods `startActivityB()` and `sayHello()` respectively. The statements within these

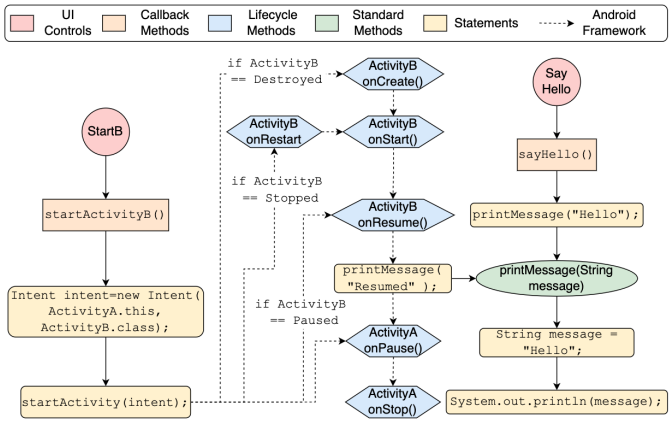

Figure 1: A subset of an eCFG, representing an Android application

methods are linked by edges in their execution order. Method call edges are also shown, for instance the statement `startActivity(intent)` calls the subsequent lifecycle methods. Some of the edges are shown as a dashed arrow. These edges are not contained in the graph because they originate from outside the application code and the edge used at runtime is determined by the Android framework. It decides which edge to follow based on the current state of the system or application, specifically the called activity. 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. In order to simplify the example, only statement vertices for the lifecycle method `ActivityB: onResume()` are included.

In comparison to a traditional control flow graph (CFG), the defined eCFG introduces UI controls, and the control flow to a linked method callback, providing the functional outcome of a user interaction.

2. Model Implementation

As defined above, the eCFG is composed of three types of vertices, that represent three elements of a mobile application. Each vertex type as well as their connected edges are collected using static analysis, dynamic analysis or a combination of both. The static analysis is performed using Soot - a Java optimisation framework, used to analyse, instrument, optimise and visualise Java and Android applications [34], FlowDroid - a fully context-, field-, object- and flow-sensitive taint analysis tool for Android applications [35], and AndroGuard - a python-based tool used for reverse engineering Android apps [36], [37]. The dynamic analysis uses Appium - an open-source project designed to facilitate automation of UI interactions with web and mobile platforms [38], [39], to perform an enhanced depth-first search (DFS) on the application UI using an instrumented version of the Android PacKage (APK). The vertices and edges of the eCFG are populated as follows:

- **Statement** vertices are created based on FlowDroid `UnitGraph` objects created for each method in the AUT.

Each `UnitGraph` contains a unit chain detailing all the statements and their execution order within the method. Intra-procedural call edges are determined using Soot's inter- and intra- procedural analysis.

- **Method** vertices are created by retrieving all the methods, for each class, identified by FlowDroid in the AUT. The methods and the inter-procedural calls between them are also gathered from the call-graph generated by AndroGuard. The model is limited to internal components by filtering external library methods such as AndroidX. Before adding a method to the model, it is categorised as an activity lifecycle method, input listener method, or a standard Java method.
- **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 controls declared in the Java source code are identified and added to the model during the dynamic analysis. The edge between UI controls and their associated listener callback methods are populated during the dynamic analysis, the instrumented APK provides a log of method calls and the UI interaction that initiated them.

As mentioned, the eCFG and DroidGraph are presented in our previous research [33] where we explore methods of modelling application code structures and the benefits of combining both static and dynamic analysis. This paper presents *PADRAIG*, a test input generator that uses DroidGraph to create an eCFG model and uses this model to generate strong tests in an efficient way.

III. *PADRAIG* Implementation

PADRAIG is composed of various tools and components that enable it to perform model-based test input generation for an Android application. Figure 2 provides an overview of all the processes involved and how they interact. The AUT is provided as an input, in the form of an APK file, to both DroidGraph and *PADRAIG*. DroidGraph first creates an eCFG of the AUT, as discussed in Section II. *PADRAIG* explores the AUT and generates inputs simultaneously using Appium [38], [39] and the eCFG created by DroidGraph. Appium is highly integrated with Android's UI-Automator which provides detailed knowledge of UI controls currently available to interact with, and their locations on the screen. Using the eCFG, the APK, and Appium coupled with an Android emulator, *PADRAIG* generates and outputs a test for the AUT, consisting of a sequence of inputs. The inputs that *PADRAIG* generates are currently limited to taps, while inputs such as swipes and text inputs are planned for future work.

Algorithm 1 shows how *PADRAIG* generates test inputs, given a budget chosen by the user in the form of N, the number of interactions a test should contain. The algorithm starts in the default launch activity of the AUT by launching the app on the Android emulator. Using Appium, it retrieves the UI controls available for the currently displayed activity, including the back button UI control (line 5 in algorithm 1). Each of the

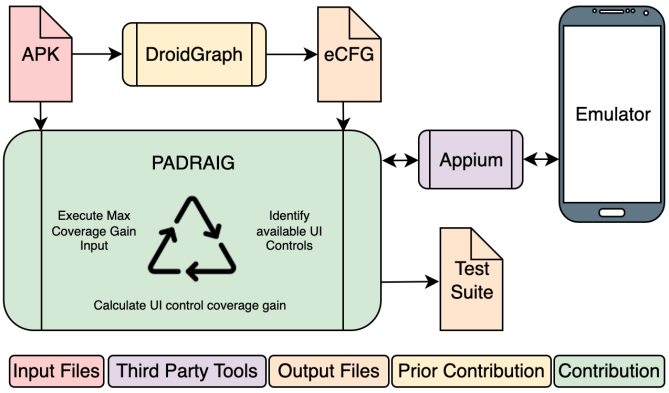

Figure 2: The component structure of the *PADRAIG* framework

available UI controls is assigned a fitness value based on the potential coverage gain it provides (line 8 in algorithm 1). Algorithm 2 shows how the potential coverage gain of a UI control is calculated. *PADRAIG* determines the next input by choosing the UI control with the maximum fitness value, i.e. the maximum potential coverage gain. It is possible that multiple UI controls are assigned the same fitness value. In this scenario, *PADRAIG* uses a frequency strategy that chooses the least used UI control from the inputs that have equal fitness values (line 12 in algorithm 1). The inputs already performed on the AUT are tracked so *PADRAIG* can determine which UI control has been used the least (line 16 in algorithm 1). When *PADRAIG* decides to interact with a UI control, the associated UI control vertex, as well as the vertices in the path starting from it, are marked as visited in the eCFG (line 18 in algorithm 1). Visited vertices reduce the potential coverage gain of a UI control and therefore the fitness of repeated inputs. This allows for repeated inputs when needed but does not allow excessive repetition.

As mentioned, the fitness value of the UI controls are determined by the potential coverage gain they provide. Algorithm 2 shows how the fitness value for a UI control vertex v is calculated. *PADRAIG* traverses the paths starting at this vertex, updating the fitness value based on the vertices it encounters, until the paths end. The returned value is a count of the number of vertices that will potentially be visited as a result of the input, but have not already been visited previously. Additionally vertices demonstrating application features, such as the launch of a new activity, increase the fitness value of the UI control, as they indicate potentially unexplored territory in the AUT. The calculated coverage gain is an overestimate of the coverage gain that will be achieved by using the UI control. For example, when an if-else statement is encountered, both branches are included in the potential coverage gain, despite only one branch executing each time the UI control is used. A more accurate coverage value would require a data flow analysis which is supported by FlowDroid, the primary static analysis tool used to generate the eCFG, but is not included in *PADRAIG*.

Algorithm 1 *PADRAIG* generating a sequence of inputs

Require: $graph$, the eCFG of the AUT
Require: N, the number of inputs to generate.
Ensure: $test$, a test, i.e, a sequence of inputs.

```
1: test ← ∅
2: tracker ← ∅
3: launchApp()
4: for count ← 0 to N do
5:     controls ← getAvailableControls()
6:     covGains ← ∅
7:     for c ∈ controls do
8:         g ← graph.calculateCoverageGain(c)
9:         covGains.add(c, g)
10:    maxControls ← maxValue(covGains)
11:    if maxControls.size() > 1 then
12:        next ← tracker.leastFrequent(maxControls)
13:    else
14:        next ← maxControls.get(0)
15:    next.click()
16:    tracker.addInput(next)
17:    test.add(next.getAdbCommand())
18:    graph.markCoveredVertices(next)
19: return test
```

Algorithm 2 Calculating the potential coverage gain of a UI control vertex in an eCFG

Require: v, a UI control vertex in the app eCFG
Ensure: g, potential coverage gain of the UI control vertex v

```
1: procedure CALCULATECOVERAGEGAIN(v)
2:     if v.hasVisit() then
3:         g ← 0
4:     else
5:         g ← 1
6:     for e ∈ v.getOutGoingEdges() do
7:         t ← e.getEdgeTarget()
8:         g ← g+ calculateCoverageGain(t)
9:     if v.getAttribute(type) == STATEMENT and
       v.hasFeature() then
10:        g ← g + 1
11:    return g
```

IV. Experiments

This section describes the experiments conducted to assess the test input generation capabilities of *PADRAIG* compared to the state of the art. First, it outlines the research questions explored. Then, it introduces the applications used as benchmarks. Then, it describes the protocols and metrics used to investigate the research questions. Finally, it discusses a statistical analysis performed on the experiment results. The experimental setup and results are made available online [32].

1. Research Questions

In order to understand the contribution of *PADRAIG*, the following research questions are explored:

RQ1 **Can *PADRAIG* generate tests that cover more of the application than state of the art tools that use a variety of generation techniques?**
This research question compares the line coverage achieved by *PADRAIG* and 3 state of the art automated test input generation frameworks, Monkey [15], STGFA-SMOG [27], and Stoat [30]. It explores how much of the applications' code a test produced by each tool execute, reflecting how complete a test suite the tools can create.

RQ2 **How long do the search- and model-based techniques take to generate a test?**
This research question explores the runtime required by search- and model-based test input generation techniques to produce a test that can achieve suitable coverage of the AUT and demonstrates the feasibility of the approach being used in practice. The runtime of model-based test input generation techniques also include the time taken to generate a model of the AUT in order to encompass the entire test generation process. Note: Monkey is not included in order to maintain a fair comparison, its random nature makes the test generation artificially fast, i.e. it does not generate tests using a "smart" method.

2. Subject applications

In order to study the above research questions, each test input generation technique was applied to a diverse set of Android applications. A random selection of applications were chosen from the F-Droid market place [40], which contains over 4000 apps. In order to represent the different types of applications developed in practice, one application was randomly selected from each of the 20 categories (excluding Games) of applications found in F-Droid. Before the selection was made, unsuitable apps were filtered out based on 3 criteria:

1) apps in the games category, as they often include game development frameworks such as unity, which are not supported by *PADRAIG*.
2) apps with an Android software development kit (SDK) version 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.

Three apps used in previous research [41] are also included to demonstrate that previous results traversing the eCFG are consistent with the approach used by *PADRAIG*. Table I shows the list of apps used in the experiments. The lines of code (LoC) for each app is obtained using the Statistic [42] plugin in Android Studio. All corresponding APK files are made available in [32].

3. Experimental Procedure

In order to investigate the above research questions, each of the frameworks, Monkey [15], STGFA-SMOG [27], Stoat [30],

TABLE I: Applications used to compare *PADRAIG* with the state of the art

App name	Category	Version	LoC
Activity Lifecycle	Science & Education	1	909
Ad-Free	Internet	41	5,608
Battery Live	Theming	13	1,062
Camera Roll	Multimedia	36	21,006
Contact Book	Phone & SMS	1	2,904
Drinks	Reading	32	1,820
Git Quick Reference	Development	7	2,832
GPSTest	Navigation	18093	134,547
Loyalty Card Keychain	Money	39	5,998
MoClock	Time	4	499
PIN Mnemonic	Security	6	2,304
Pixel Filter	Graphics	24	1,435
Simple Explorer	System	67	13,426
Simple Todo	Time	5	3,216
Taskkeeper	Writing	6	1,626
Timetable	Science & Education	17	8,477
Volume Control	System	32	2,184
Webradio	Connectivity	5	1,709
Wine Cellar	Sports & Health	4	3,804

and *PADRAIG* are applied to all the applications in Table I. In order to account for randomness, Monkey, STGFA-SMOG, and Stoat are executed 10 times for each application. *PADRAIG* uses a deterministic algorithm so it does not contain randomness, and therefore is executed once per application. The maximum number of inputs per test in [27] and [30] was 50 and 30 respectively, so each tool was given a budget of 50 interactions per test for a fair comparison.

Monkey [15] was executed under two settings: limiting it to only click interactions (Monkey Click), and allowing it to use all interactions it supports (Monkey All). Both settings also had a 500 ms threading time and were limited to packages belonging to the AUT. These settings are applied to reduce stress testing on the AUT and ensure Monkey generated inputs are only applied to the AUT. The same settings presented in [27] were allocated to STGFA-SMOG, a population of 10 tests and search budget of 10 generations. The finite state machine (FSM) model construction and test generation that Stoat performs is time based. As presented in [30] Stoat was given a budget of one hour to generate an FSM and another two hours to generate a test.

The overall line coverage achieved by each of the generated tests was measured using ACVTool [43] which does not use the original source code. The measured line coverage is based on a Smali representation of the bytecode. A comparison of the coverage achieved for each application by Monkey [15], STGFA-SMOG [27], Stoat [30], and *PADRAIG* serves to answer RQ1, while comparing the runtime required to generate the tests answers RQ2. All experiments and results can be reproduced using the code and Docker images provided [32].

4. Statistical Analysis

In order to demonstrate the significance and level of improvement provided by *PADRAIG* over Monkey, STGFA-SMOG, and Stoat, each approach is compared using the Mann-Whitney U test, a non parametric test telling whether there is a significant difference among the results. The value $\alpha = 0.05$ is used as a confidence value of the null hypothesis that there is no significant difference. In case of significant difference, Vargha and Delaney's $\hat{A}_{12}$ effect size is used to assess the strength of the significance; if $\hat{A}_{12}$ is greater than 0.5, the results of *PADRAIG* are significantly better than those of Monkey, STGFA-SMOG, and Stoat. By following Kitchenham et al.'s classification [44], the following categories of strength are identified: negligible when $\hat{A}_{12} \in (0.5, 0.556)$, small when $\hat{A}_{12} \in [0.556, 0.638)$, medium when $\hat{A}_{12} \in [0.638, 0.714)$, and large when $\hat{A}_{12} \geq 0.714$. Similar categories can be identified for $\hat{A}_{12} < 0.5$, i.e., when *PADRAIG* is significantly worse. In this comparison, we count the number of experiments where one approach is better than the other, and with which strength.

V. Results

This section describes the results obtained from the experiments discussed in Section IV and how they relate to the defined research questions.

1. RQ1 Can *PADRAIG* generate tests that cover more of the application than state of the art tools that use a variety of generation techniques?

Table II, and Figure 3 show a comparison of the line coverage achieved by each approach. In Table II, for each application, the maximum line coverage achieved for that app is shown in bold.

The tests generated by *PADRAIG* achieve higher line coverage of the AUT than the average achieved by Monkey with all interactions, Monkey with only click interactions, STGFA-SMOG, and Stoat in 18, 17, 16, and 16 out of 19 apps respectively. While the coverage achieved by the state of the art is comparable in some cases, for example the Drinks app, *PADRAIG* covers, on average 16%, and up to 50% more of the AUT than the state of the art for the majority of the AUTs. There are cases where *PADRAIG* did not perform as well as the state of the art, for example, Monkey, STGFA-SMOG, and Stoat achieved higher line coverage than *PADRAIG* in the Timetable app. There are many factors that could cause this, for instance, the AUT functionality could be gesture orientated or require system events which *PADRAIG* does not currently support. Despite these few instances, the results clearly demonstrate that *PADRAIG* is not only effective at covering the AUT but better than the state of the art for the majority of the AUTs.

The line coverage achieved by *PADRAIG* and the state of the art appears to be low in many of the apps. There are a number of explanations for this, for example some applications require content, such as a contact app, where a portion of the interface is only visible after the user has added a contact. Also larger

TABLE II: Coverage (%) achieved by *PADRAIG* and the state of the art

App Name	*PADRAIG*	Monkey All			Monkey Click			STGFA-SMOG			Stoat		
		Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
Activity Lifecycle	**59.8**	19.2	11.24	28.19	19.99	13.62	24.77	23.51	12.55	28.39	25.01	21.64	41.07
Ad-Free	**41.11**	19.57	18.13	27.34	18.44	18.13	19.4	21.75	18.22	25.23	18.14	18.14	18.14
Battery Live	**85.08**	26.9	25.63	28.29	27.95	25.98	30.4	34.48	25.63	78.44	49.63	26.03	63.47
Camera Roll	**14.54**	5.53	5.43	5.69	5.6	5.43	6.91	7.25	5.5	10.12	6.2	5.71	9.06
Contact Book	**34.47**	6.41	6.41	6.41	8.13	6.41	11.58	6.88	6.41	8.48	8.39	6.55	13.76
Drinks	27.88	24.1	23.75	25.0	24.33	23.75	25.09	24.97	23.75	26.56	27.91	25.09	**29.91**
Git Quick Reference	31.49	29.5	22.84	32.65	30.3	28.65	33.1	35.48	29.96	36.41	32.02	28.53	**37.66**
Gpstest	**43.42**	11.25	11.09	11.44	11.18	11.09	11.55	22.52	21.16	25.59	11.71	11.2	16.09
Loyalty Card Keychain	**22.41**	6.75	5.81	7.87	6.01	5.81	6.63	6.63	5.81	7.46	6.61	6.3	7.72
Moclock	**45.87**	11.45	6.71	21.84	19.61	6.71	35.57	25.23	18.49	34.87	31.98	29.49	44.46
Pin Mnemonic	**6.69**	3.48	3.32	3.79	3.8	3.6	4.07	3.92	3.44	4.34	3.34	3.32	3.41
Pixel Filter	**67.08**	56.27	54.17	60.57	56.84	54.17	58.94	52.83	29.47	60.92	56.27	56.03	58.43
Simple Explorer	**41.21**	30.15	29.29	31.63	30.69	28.91	33.03	32.15	29.63	35.84	30.47	29.72	34.6
Simple Todo	27.27	11.3	11.01	11.99	12.65	11.01	18.54	13.55	11.01	18.24	15.64	12.09	**33.31**
Taskkeeper	31.79	15.75	10.72	18.76	11.39	10.72	16.93	16.69	10.72	40.2	16.73	10.92	**42.73**
Timetable	5.92	6.06	5.91	7.38	7.22	5.91	9.18	8.34	6.76	**10.97**	6.04	5.87	6.88
Volume Control	**72.29**	33.67	30.44	39.48	37.16	33.54	41.08	27.76	4.64	40.55	35.17	32.53	46.6
Webradio	34.83	32.36	29.4	39.52	37.04	28.37	40.81	37.95	28.77	39.76	38.25	30.09	**42.89**
Wine Cellar	**24.81**	13.11	11.9	15.54	11.9	11.9	11.9	12.9	11.74	21.03	12.46	11.74	18.95

applications may require a larger number of inputs in order to cover a greater portion of the code.

Table III further demonstrates the level of significant improvement provided by *PADRAIG* over the state of the art. It shows that *PADRAIG*s model-based approach is statistically significantly better than Monkey with all interactions, Monkey with only click interactions, STGFA-SMOG, and Stoat in 17, 16, 16, and 16 out of 19 apps respectively.

2. RQ2 How long do the search- and model-based techniques take to generate a test?

The runtime for creating a test for an application is not commonly evaluated in past research. However, it is arguably a compelling factor in the adoption of an automated test input generation technique. Automated test input generation is intended to remove the slow, labour intensive process of creating tests manually, and therefore needs to be fast and efficient. The faster tests are created and executed on the AUT, the faster faults can be detected and fixed. Efficient generation techniques can lead to more extensive and comprehensive test suites that can keep up with the fast pace of the mobile industry.

While Monkey is a test input generator, the algorithm used to determine suitable inputs and its computational complexity can be compared with that of a simple random number generator. As such, Monkey's test generation runtime is artificially fast and will always be several orders of magnitude better than any alternative approach, regardless of the line coverage they achieve. Therefore, when comparing the runtime of *PADRAIG* with the state of the art, Monkey is not included in order to maintain a fair comparison.

TABLE III: Statistical significance of the coverage improvements between *PADRAIG* and the state of the art

App Name	Monkey		STGFA-SMOG	Stoat
	All Inputs	Click Inputs		
Activity Lifecycle	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓
Ad-Free	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓
Battery Live	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓
Camera Roll	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓
Contact Book	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓
Drinks	✓✓✓✓	✓✓✓✓	✓✓✓✓	✗✗✗✗
Git Quick Reference	≡	≡	✗✗✗✗	≡
Gpstest	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓
Loyalty Card Keychain	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓
Moclock	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓
Pin Mnemonic	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓
Pixel Filter	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓
Simple Explorer	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓
Simple Todo	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓
Taskkeeper	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓
Timetable	✓✓✓✓	✗✗✗✗	✗✗✗✗	✓✓✓✓
Volume Control	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓
Webradio	≡	≡	✗✗✗✗	✗✗✗✗
Wine Cellar	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓

≡: no statistically significant difference between the approaches. ✓: *PADRAIG* tests are better than the approach on top for the metric, ✗ means that it is worse; the num. of symbols is the strength: negligible (✓, ✗), small (✓✓, ✗✗), medium (✓✓✓, ✗✗✗), large (✓✓✓✓, ✗✗✗✗)

Table IV shows a comparison of the runtime required by STGFA-SMOG, Stoat, and *PADRAIG* to generate a test for each AUT. The runtime shown for *PADRAIG* and Stoat includes the time taken to generate a model of the AUT, and the minimum runtime required for each app is shown in bold. The runtime required by *PADRAIG* is on average 2.4 hours or

TABLE IV: Runtime (in minutes) of search- and model-based techniques

App Name	*PADRAIG*	SMOG			Stoat		
		Avg	Min	Max	Avg	Min	Max
Activity Lifecycle	**17.7**	156.59	142.88	187.33	192.72	192.52	192.78
Ad-Free	**17.85**	185.12	167.17	207.03	189.71	163.52	192.63
Battery Live	**18.1**	146.09	126.57	169.33	192.6	192.45	192.62
Camera Roll	**22.3**	159.52	137.88	185.27	192.6	192.6	192.62
Contact Book	**17.9**	142.66	133.73	154.95	191.95	190.35	192.6
Drinks	**22.65**	150.88	143.47	162.22	192.6	192.47	192.62
Git Quick Reference	**23.6**	146.45	131.43	169.85	192.59	192.45	192.62
Gpstest	**48.12**	168.97	143.0	226.38	192.61	192.58	192.62
Loyalty Card Keychain	**22.45**	154.34	144.88	185.88	192.68	192.47	192.78
Moclock	**19.88**	138.62	124.58	162.88	192.61	192.6	192.62
Pin Mnemonic	**18.87**	139.89	130.58	165.38	192.61	192.58	192.62
Pixel Filter	**65.43**	155.47	132.0	190.12	192.58	192.57	192.58
Simple Explorer	**21.52**	147.12	127.03	174.28	192.59	192.58	192.6
Simple Todo	**19.45**	140.6	123.88	165.65	192.06	191.7	192.62
Taskkeeper	**16.82**	134.19	119.57	158.25	186.86	185.95	188.43
Timetable	**31.58**	148.57	139.28	156.7	192.59	192.47	192.62
Volume Control	**57.68**	150.46	137.63	181.58	192.59	192.48	192.6
Webradio	**19.8**	132.59	125.53	142.72	192.58	192.45	192.6
Wine Cellar	**20.77**	149.31	133.05	175.73	192.6	192.6	192.62

84% less less than the runtime required by STGFA-SMOG, a search based technique, and Stoat, an alternative model-based technique. Due to Stoat being time based, an argument can be made for lowering the allowed time for Stoat to generate an FSM and test inputs to achieve the same speed as *PADRAIG*. However, as seen in Table II and Figure 3, Stoat fails to generate tests with better line coverage of the AUT than *PADRAIG* despite its current time budget, which is much larger than *PADRAIG*'s. This suggests that, given a time budget equal to *PADRAIG*'s runtime, Stoat would perform even worse. Search-based techniques, such as STGFA-SMOG, are computationally expensive, they require many tests to be created and run before returning the best test in the population, this therefore increases the runtime required.

While the scalability of each approach was not evaluated in this paper, the runtime achieved by *PADRAIG* appears unaffected by the size of the AUT, for example, *PADRAIG*s runtime using the app Simple Explorer with 13,426 LoC is 21.52 minutes. *PADRAIG* is also deterministic, which means that it can generate effective tests with a lower runtime, while providing the same level of success for every test it generates. Table V also illustrates the level of significant improvement provided by *PADRAIG* over the state of the art. It shows that the test generation runtime of *PADRAIG*s model-based approach is statistically significantly better than the state of the art in all the AUTs.

VI. Related work

This section outlines previous work in the area of automated test input generation for Android applications. This area has grown to such an extent over the last decade that it can be further classified into subcategories based on the methods used. The most used methods are random, systematic, search-, and model-based input generation.

1. Random

Using a random approach for generating test inputs is the simplest method but it is generally very inefficient and leads to a high number of redundant (provide no contribution to the test) and repetitive (lower the diversity of the test) interactions. An example of this is Monkey [15], a program that runs on an emulator or device and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. In practice, Monkey does not support keyboard input or system notifications, and most of the random inputs applied to the interface do nothing within the application (e.g., tapping an area of the screen with no actionable UI element). Dynodroid [17], another well-known random input generator, tries to overcome this issue by applying a Frequency Strategy and Biased-Random Strategy, to increase the efficiency of the inputs it selects. The frequency strategy uses the inputs that have been least frequently used in the past and the biased-random strategy uses the inputs that are relevant in most contexts. Unlike Monkey, Dynodroid also supports keyboard input and system notifications, however, this is possibly its downfall. For instance, in order to provide system events (e.g, keyboard input, notifications, changing device orientation), Dynodroid needs to instrument the Android framework and in doing this it became hard to update and maintain, which has led to the framework becoming outdated

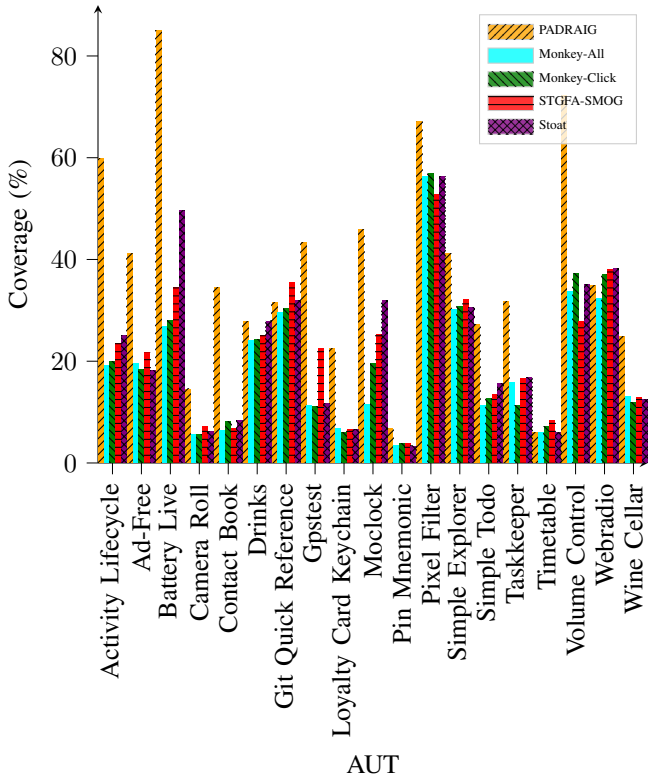

Figure 3: Coverage (%) achieved by *PADRAIG* and the state of the art

TABLE V: Statistical significance of the runtime improvements between *PADRAIG* and the state of the art

App Name	STGFA-SMOG	Stoat
Activity Lifecycle	✓✓✓	✓✓✓
Ad-Free	✓✓✓	✓✓✓
Battery Live	✓✓✓	✓✓✓
Camera Roll	✓✓✓	✓✓✓
Contact Book	✓✓✓	✓✓✓
Drinks	✓✓✓	✓✓✓
Git Quick Reference	✓✓✓	✓✓✓
Gpstest	✓✓✓	✓✓✓
Loyalty Card Keychain	✓✓✓	✓✓✓
Moclock	✓✓✓	✓✓✓
Pin Mnemonic	✓✓✓	✓✓✓
Pixel Filter	✓✓✓	✓✓✓
Simple Explorer	✓✓✓	✓✓✓
Simple Todo	✓✓✓	✓✓✓
Taskkeeper	✓✓✓	✓✓✓
Timetable	✓✓✓	✓✓✓
Volume Control	✓✓✓	✓✓✓
Webradio	✓✓✓	✓✓✓
Wine Cellar	✓✓✓	✓✓✓

≡: no statistically significant difference between the approaches. ✓: *PADRAIG* tests are better than the approach on top for the metric, ✗ means that it is worse; the num. of symbols is the strength: negligible (✓, ✗), small (✓✓, ✗✗), medium (✓✓✓, ✗✗✗), large (✓✓✓✓, ✗✗✗✗)

and unused. More recently, MonkeyImprover [19] proposed refactoring the graphical user interface (GUI) of the AUT to make UI controls that lead to more complex background code more prominent, and thus increasing the likelihood of being interacted with by Monkey. This method shows great promise but requires further evaluation as the approach was only applied to one application. Muangsiri et al. [18] also reduced redundant random interactions by using a behavioural model populated by mining usage logs and mapping statistical insights to the components in a GUI tree. They evaluated the coverage achieved by their approach on three applications, and compared against state of the art test input generators. Their approach is successful in two of the three applications against previous automated solutions. Despite the approach proposed by Muangsiri et al. [18], Dynodroid, and MonkeyImprover performing better than Monkey, showing an increased application coverage and less interaction redundancy, Monkey has remained one of the most popular frameworks available for automated test input generation. This could be attributed to Monkey being maintained and packaged as part of the Android framework and its simple usage, resulting in the majority of research comparing its results with results from Monkey.

2. Systematic

Systematic input generation, sometimes referred to as model-learning, involves dynamically analysing the interface of an application and generating appropriate test inputs as they are found, using a predefined traversal strategy, for example a depth-first exploration. 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 [20], which maintains a state machine model as it systematically traverses the UI of the AUT, called a GUI tree. This GUI tree model contains the set of GUI states and state transitions encountered during the systematic traversal. However, due to a lack of maintenance, AndroidRipper is unusable on newer Android versions. The framework Automatic Android App Explorer (A3E) [21] implements two forms of exploration, a depth-first exploration and a targeted exploration. The 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. 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 [12]. ACTEve [16] proposes a concolic approach that symbolically tracks events from where they originate to where they are handled. While this technique

does not require prior knowledge of the AUT structure, it does require that the AUT and the Android SDK be instrumented. Another approach, CrashScope [22], 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.

3. Search-based

Search-based testing is a popular method of generating tests, usually involving the use of a genetic or evolutionary algorithm. Similarly to systematic approaches, this technique does not require prior knowledge of the interface structure or the underlying code structure, and often begins with randomly generated tests that the search algorithm is applied to. An example of this is AGRippin [24] which uses a combination of genetic and hill climbing techniques to generate tests that are more effective at covering an AUT. The aim of AGRippin is to increase the coverage of each generated population. The fitness values used to improve each population are based on the overall fitness of the current population as well as the fitness of each individual test, where each test is a sequence of inputs to the AUT. EvoDroid [23] is one of the oldest search-based frameworks and uses an evolutionary approach. Despite being a search-based technique, EvoDroid does model aspects of the AUT, specifically an interface model comprising the UI structure of the AUT, retrieved from the Android application XML layout files, and a disconnected call graph, generated using MoDisco [45]. The call graph is disconnected due to the event-based nature of Android applications. These models are used to generate the original inputs and sequence of events that the evolutionary algorithm is applied on to create tests that provide better coverage of the AUT. A well-known search-based approach is Sapienz [25], which employs a multi-objective search combining random fuzzing, systematic and search-based exploration, string seeding, and multi-level instrumentation. Sapienz' success is well known due to its acquisition by Facebook after publication. Unfortunately since then it has become closed source and the publicly available version has become outdated and unusable on newer Android versions. More recently, ADAPTDROID [26] proposed a search-based technique to adapt existing GUI tests across similar applications because common functionalities often result in common GUI tests. Utilising a donor test from another app, ADAPTDROID can generate a test suite containing tests that have similar input semantics and test oracles as the donor. An evolutionary algorithm is used to assess the similarity of a sequence to the donor. STGFA-SMOG [27], another search-based approach for Android test input generation, focuses on non-functional properties, using a genetic algorithm based on the NSGA-II algorithm. Despite its main focus it does allow class, method and line coverage as possible fitness values for generated tests and shows great promise by improving line coverage as well as quality attributes such as CPU and network usage. Unfortunately STGFA-SMOG is only compared against a random test input generation technique. All the mentioned approaches have been shown to increase the coverage of their respective AUTs when compared against state of the art tools, but search-based methods have shown a significant disadvantage, they require a very large runtime on even the best hardware, due to the number of test executions that are required and the computation complexity of the algorithms involved.

4. Model-based

Model-based testing requires a formal model of the application, such as a CFG, FSM, event flow graph (EFG), etc. which is used to generate a test consisting of device inputs. Many systematic approaches can be seen as model-based because they create a model while systematically testing the application. Stoat [30] and MobiGuitar [29] model the AUT as an FSM. Stoat uses both static and dynamic analysis, enhanced by a weighted UI exploration strategy, to explore the AUTs behaviours and construct a stochastic FSM [30]. MobiGuitar uses an enhanced version of Android Ripper, dynamically traversing the application in a breadth-first fashion to generate the FSM [29]. 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 test inputs [30]. 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 [30], while the reliance of MobiGuitar on AndroidRipper means it faces the same pitfalls. A3E [21], mentioned in Section VI-B, also has a targeted approach that uses static bytecode analysis to extract a static activity transition graph (ATG), which is then explored systematically while the app runs on the device. Unfortunately the targeted approach suffers from the same issues as its systematic depth-first counterpart. Another approach to model-based testing is implemented in SwiftHand [28], 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 it focuses 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. CrawlDroid [31] using

a GUI model, avoids using interactions that invoke the same background code (by grouping UI elements with similar input types and paths from the root of the GUI model), thus not effecting the overall coverage of the test. To achieve this, it groups UI controls in a GUI state and implements a feedback-based exploration strategy to only trigger interactions that can improve the code coverage. While CrawlDroid is successful in finding app crashes in a large selection of test apps, it is not compared against another automated test input generator.

5. Comparison of *PADRAIG* and other approaches

PADRAIG combines aspects of the different techniques discussed above to achieve fast generation of effective tests. Random inputs have been shown to introduce redundancies so *PADRAIG* does not include them in its generation process. This allows *PADRAIG* to produce deterministic and reproducible results, producing strong tests on every run. However, *PADRAIG* does employ a frequency strategy, as seen in Dynodroid, in order to reduce unnecessary repetition while still allowing repeated inputs when needed. Unlike systematic approaches *PADRAIG* uses a pre-populated model of the AUT to guide its test input generation. This enables *PADRAIG* to generate test inputs using knowledge of the entire model, rather than a model that is created during the test input generation. This places *PADRAIG* in the category of Model-based tools, as discussed above. Search-based techniques do not require a model of the AUT but require a much larger test generation runtime to generate adequate tests. *PADRAIG* does not use search algorithms but it does evaluate inputs based on their potential coverage gain which is determined based on the model structure, in a similar way to the fitness values used by search-based techniques to evaluate members of a population. As a model-based tool, *PADRAIG* is most similar to previous approaches in this category. The most significant difference between *PADRAIG* and previous model-based tools is the type of model used in the test input generation. Past model-based test input generation approaches have largely used models that focus on the UI of the AUT. *PADRAIG* uses an eCFG that is composed of the entire applications structure, including not only application UI elements but also low level statement and procedural structure, allowing it to not only produce tests covering the different UI elements and interaction behaviours of the AUT, but also covering the different methods and branches in the application's code, i.e., the different internal behaviours of the system.

VII. Threats to Validity

This work's validity is subject to different threats [46] that this section details, as well as how they are addressed.

Construct validity. The metrics used to assess *PADRAIG* may not be suitable. However, the coverage of the AUT, a standard metric in testing, is used as a proxy metric that reflects the proportion of possible actions and system behaviours that are tested.

Conclusion validity. State of the art tools, Monkey, STGFA-SMOG, and Stoat utilise randomness to generate tests, which could impact results. To account for this randomness when comparing *PADRAIG* with the state of the art, 10 tests are generated with each tool for each application in the experiments and a statistical analysis is applied to the results.

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

External validity. The approach adopted by *PADRAIG* may not generalise. To mitigate this threat, *PADRAIG* is assessed over a set of diverse applications that span multiple domains. Also the implementation of *PADRAIG* is made available, allowing further investigations in the future.

VIII. Future Work

A first avenue for future work is the study of app user content and its effect on the application coverage of testing tools. The coverage achieved in this paper and in related research is arguably low. This could be attributed to the lack of user content in the AUT. For example, in an application that manages a users' contacts, many of the application features and UI controls are not available until the user has added content, in this example, a contact. This could be resolved in two ways 1) the content is loaded into the AUT, thus enabling all application features, prior to the test generation, or 2) a search strategy is developed that first focuses on generating content through the UI in the same manner as a user.

An extension of the work presented in this paper is also a possibility for future work. For example, including more interaction types, such as swipes and text input, could further improve *PADRAIG* and the results presented. The model-based approach used by *PADRAIG* could also be extended to take advantage of more sophisticated search and machine learning techniques to further increase the coverage achieved as well as target more test criteria. Finally a broader exploration and analysis of the eCFG to ascertain further structural insights and provide a model-based test input generator such as *PADRAIG*, guidance to both increase the coverage presented in this paper, as well as increase its effectiveness in discovering application defects.

IX. Conclusion

Smartphones have grown to be an essential part of people's lives, and defects within mobile applications can not only affect the revenue generated by these apps, but also severely impact their users. Despite years of research [4]–[14] Android testing is still largely performed manually. Automated test input generation presents a promising approach to alleviating that manual effort and to provide tests that can keep up with the fast pace of the mobile industry.

This paper introduced *PADRAIG*, a model-based framework that uses an eCFG model and generates effective test inputs that can achieve higher line coverage of an AUT with a faster test generation runtime than state of the art Android test generation tools. In particular, comparison between *PADRAIG* and 3 state of the art tools, each of which uses a different

automated test input generation technique: Monkey a random approach [15], STGFA-SMOG a search-based technique [27] and, Stoat [30] an alternative model-based approach shows *PADRAIG*'s advantages.

Monkey, STGFA-SMOG, Stoat and *PADRAIG*, were applied to 19 diverse apps and demonstrate that *PADRAIG* covers, on average 16%, and up to 50% more of the AUTs, with an average of 84% less test generation runtime. The comparison of *PADRAIG* and the state of the art shows that *PADRAIG* can achieve higher line coverage of the AUT in at least 16 out of 19 apps and demonstrates a *large* (see Section IV-D) statistically significant improvement. *PADRAIG*s's increased coverage, faster runtime, and its deterministic generation make it suitable for providing reliable tests in the current fast paced, competitive environment of mobile application development.

REFERENCES

[1] GSMA, "Global smartphone penetration rate as share of population from 2016 to 2022," https://www.statista.com/statistics/203734/global-smartphone-penetration-per-capita-since-2005/, 2023, accessed on 30/11/2023.

[2] Statista, TechCrunch, "Worldwide consumer spending on mobile apps and games in Google Play and the Apple App Store as of 3rd quarter 2022 (in billion U.S. dollars)," https://www.statista.com/statistics/183469/app-stores-global-revenues/, 2022, accessed on 30/11/2023.

[3] P. Roma, G. Perrone, and F. Valenti, "An empirical analysis of revenue drivers in the mobile app market," in *Proceedings of POMS 24th Annual Conference*, 2013.

[4] 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.

[5] 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.

[6] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, A. Bertolino *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[7] 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.

[8] D. Amalfitano, N. Amatucci, A. M. Memon, P. Tramontana, and A. R. Fasolino, "A general framework for comparing automatic testing techniques of android mobile apps," *Journal of Systems and Software*, vol. 125, pp. 322–343, 2017.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] A. Samir, H. Amin, and N. Badr, "A survey on automated user interface testing for mobile applications," *International Journal of Intelligent Computing and Information Sciences*, pp. 1–11, 2022.

[15] Android Developers, "UI/Application Exerciser Monkey," https://developer.android.com/studio/test/monkey.html, 2012, accessed on 20/06/2023.

[16] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

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

[18] W. Muangsiri and S. Takada, "Random gui testing of android application using behavioral model," *International Journal of Software Engineering and Knowledge Engineering*, vol. 27, no. 09n10, pp. 1603–1612, 2017.

[19] S. Paydar, "Automated gui layout refactoring to improve monkey testing of android applications," in *2020 CSI/CPSSI International Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*. IEEE, 2020, pp. 1–9.

[20] 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.

[21] 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.

[22] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Crashscope: A prac-

tical tool for automated testing of android applications," in *ICSE*, 2017, pp. 15–18.

[23] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 599–609.

[24] D. Amalfitano, N. Amatucci, A. R. Fasolino, and P. Tramontana, "Agrippin: a novel search based testing technique for android applications," in *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, 2015, pp. 5–12.

[25] 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.

[26] L. Mariani, M. Pezzè, V. Terragni, and D. Zuddas, "An evolutionary approach to adapt tests across mobile apps," in *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 2021, pp. 70–79.

[27] T. Gereziher, S. Gebrekrstos, and G. Gay, "Search-based test generation targeting non-functional quality attributes of android apps," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2023, pp. 1518–1526.

[28] 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.

[29] 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.

[30] 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.

[31] Y. Cao, G. Wu, W. Chen, and J. Wei, "Crawldroid: Effective model-based gui testing of android apps," in *Proceedings of the 10th Asia-Pacific Symposium on Internetware*, 2018, pp. 1–6.

[32] J. Doyle, T. Laurent, and A. Ventresque, "Supplementary material for the paper "Padraig: Precise Android Automated Input Generation"," https://github.com/jordan2doyle1/QRS-2024-Padraig, 2024.

[33] ——, "Modelling android applications through static analysis and systematic exploratory testing," in *2023 10th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 2023, pp. 94–104.

[34] 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.

[35] 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.

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

[37] S. B. Anthony Desnos, Geoffroy Gueguen, "AndroGuard Documentation," https://androguard.readthedocs.io/en/latest, 2012, accessed on 15/01/2022.

[38] OpenJS Foundation, "Appium documentation," https://appium.io/docs/en/2.1/, 2012, accessed on 30/11/2023.

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

[40] F-Droid Limited and Contributors, "F-Droid - Free and Open Source Android App Repository," https://f-droid.org/en/, 2010, accessed on 28/01/2022.

[41] 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.

[42] "Statistic Plugin for JetBrains IDEs," https://plugins.jetbrains.com/plugin/4509-statistic, 2009, accessed on 30/05/2024.

[43] A. Pilgun, O. Gadyatskaya, Y. Zhauniarovich, S. Dashevskyi, A. Kushniarou, and S. Mauw, "Fine-grained code coverage measurement in automated black-box android testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–35, 2020.

[44] B. Kitchenham, L. Madeyski, D. Budgen, J. Keung, P. Brereton, S. Charters, S. Gibbs, and A. Pohthong, "Robust statistical methods for empirical software engineering," *Empirical Software Engineering*, vol. 22, pp. 579–630, 2017.

[45] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot, "Modisco: A model driven reverse engineering framework," *Information and Software Technology*, vol. 56, no. 8, pp. 1012–1032, 2014.

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