Exploiting the Saturation Effect in Automatic Random Testing of Android Applications

Domenico Amalfitano*, Nicola Amatucci*, Anna Rita Fasolino*, Porfirio Tramontana*,
Emily Kowalczyk† and Atif M. Memon†
*Department of Electrical Engineering and Information Technologies
University Federico II of Naples, Via Claudio 21, Naples, Italy
{domenico.amalfitano, nicola.amatucci, fasolino, ptramont}@unina.it
†Department of Computer Science, Univ. of Maryland, College Park, MD, USA
eekowal@gmail.com, atif@cs.umd.edu

***Abstract*—Monkey Fuzz Testing (MFT), a form of random testing, continues to gain popularity to test Android apps because of its ease of use. (Untrained) programmers use MFT tools to fully automatically detect certain classes of faults in apps. A challenge for these tools is the lack of a stopping criterion—programmers currently typically stop these tools when they run out of time. In this paper, we use the notion of the *Saturation Effect* of an MFT tool on an app under test to define a stopping criterion, parameterized by the app's preconditions and the tool's configurations. We have implemented our approach in the AndroidRipper MFT tool. We experimentally report results on 18 real Android app subjects. We show that the saturation effect is able to stop testing when test adequacy has been achieved without wasting test cycles.**

I. Introduction

Random testing is a black-box software testing technique where programs are tested by generating random, independent inputs. Over the last several decades, variants of random testing have gained popularity in automated quality assurance testing for both conventional software applications as well as graphical user interface (GUI) frontends. These random testing techniques have several benefits over other testing methods: they are fully automatic, inexpensive, relatively easy to use, and surprisingly effective at finding bugs [1]–[3]. For example, Miller, Fredrickson, and So used random testing to reveal a wealth of command-line faults within Unix utilities [4]; similarly, Miller, Cooksey and Moore used random testing to reveal faults in 10 out of 135 command-line utilities and 22 out of 30 GUI-based utilities within MacOS applications [5]. The general validity and importance of random testing has been further evaluated and supported in a seminal work by Duran & Ntafos amongst others [1]–[3].

The growing use of random testing is particularly evident in the mobile app realm, where many platforms and developers have adopted *Monkey Fuzz Testing* (MFT), a technique that sends random button presses and mouse events to an app.[1] This growth is partially due to two factors. First, the recent wide diffusion of mobile devices and the growing success of mobile apps has seen the software engineering community devote more time and resources towards mobile app engineering [6]. Because of the competitive nature of mobile app markets, there is an increasing need for techniques and tools for effectively supporting their development lifecycle. Among the most critical activities of mobile app lifecycle is testing, which is the most frequently used quality assurance activity for these applications [7]. Second, because many of the apps are developed by amateur programmers (or hobbyists), they naturally prefer fully automatic testing techniques that do not require formal background in software testing or engineering [5].

Although tools for MFT are growing in availability and popularity, there are still open issues regarding their general use. One particular issue is how a tester may determine when the testing process should be stopped. For example, MFT tools are commonly used to implement simple testing processes where the tool is run on an old, slow computer of the testing lab and a tester periodically checks its progress [8]. Although this approach is reasonable, it is difficult for the tester to know when the testing process has reached a point where no further code coverage or fault detection can be achieved. As a result, the tester has no other option but to make an educated guess regarding two choices: stop testing or allow the process to run until some later termination criteria are met. The first choice may stop the program prematurely resulting in untested code, and the latter may stop the process later than needed and result in wasted time and computing resources. Both cases are non-optimal and rely on some expertise of the tester to use a proper termination criterion and point. Many testers let the test process run until a certain amount of time has elapsed without the process discovering new faults. While this solution is functional, it is hardly optimal.

In this paper, we relieve responsibility from the tester by automatically determining a termination point based on the exploitation of the *saturation effect*, a well-known phenomena where the rate of convergence of a test case towards a specific test adequacy criterion decreases as the amount of test execution increases [9]. The result is that the testing process may stop finding faults in the program under test. The point at which no more progress is made towards achieving a test adequacy criterion is called saturation point, and we propose it is also the optimal termination point for the program during random testing.

[1]E.g., see monkeyfuzz.codeplex.com and developer.android.com/tools/help/monkey.html

Determining the saturation point of a program can be elusive and is dependent on the preconditions of the app and the configuration settings of the tool chosen by the tester. For instance, in the Android platform we have observed that the capacity of a GUI ripper to discover faults and to cover the app source code sensibly depends on several preconditions including the types of events fired on the GUI, the timing between consecutive events, or input values provided to the input fields of the GUI [10]. Such choices are dependent on both the tester and the MFT tool the tester chooses to use. Therefore, in order to automatically determine a saturation point of the program, we provide as input a specific preconditions of the app and a configuration of the tool until the saturation effect is detected. More specifically, we determine the saturation point based on the simultaneous execution of several random testing sessions. By exploiting the predictability property of random testing shown by Arcuri et al. [11], we periodically assess the difference in code coverage between the active sessions and stop the testing process when this difference is below a chosen critical threshold.

We have implemented our fully automatic technique by means of an infrastructure including a MFT tool targeting the Android platform and a simulation environment to run each test. To validate the implemented approach, we selected 18 Android apps from the Google Play Store and tested each with our testing infrastructure. The study showed that the termination points of all processes were also saturation points.

Our paper makes the following research contributions.

- We formalize a stopping criterion for automated testing based on saturation effect.
- We recognize the importance of preconditions of the app and the configuration settings of the tool on the saturation effect, and hence, supply these as input to our process.
- We demonstrate proof-of-concept and experimentally evaluate our approach.

The rest of our paper proceeds as follows: in Section II, we describe standard MFT tools, in Section III we present a testing process based on MFT techniques that automatically stops at a termination point that is a potential saturation point, in Section IV we presents a preliminary study we performed for exploring the feasibility of the proposed testing process and its capability of reaching the saturation effect, in Section V we conducted a second experiment aimed at expanding the results of the previous study, in Section VI we consider potential validity threats to the results, in Section VII we review related work and in Section VIII we offer concluding remarks and intents for future work.

II. Monkey Fuzz Testing Tools & Trade-Offs

In this section, we offer a more detailed discussion on MFT tools and their use. Specifically, we discuss how they can be configured, provide a generic algorithm for their testing process, and discuss the well-known inefficiency and reliability problem associated with the method. We also define several variables which will be used through out the rest of this paper.

Monkey Fuzz Testing tools were first developed as a method of stress testing for both conventional software applications and those applications with a GUI front-end. They perform such testing by sending sequences of random keyboard or mouse events to the subject applications, with the aim of discovering crashes or other inconsistencies in the application's behavior.

MFT tools for mobile apps differ slightly from those for desktop GUI-based applications because they can be configured to send both user and system events. User events are generated in response to external user actions involving the touch screen (such as tap, swipe, etc.) or other parts of the device (such as pressing a device button, rotating the device, etc.). In contrast, system events are not directly triggered by users. They may be generated from the device's sensors, hardware equipment, running applications, or other non-user dependent events that are typical of mobile phones (such as the arrival of a phone call or a SMS message).

Traditionally, both mobile and desktop MFT tools offer options which can be configured to implement different behaviors. E.g., it is often possible to set the delay between consecutive triggered events, the types of events to fire (such as click, tap, or other), how often an event gets triggered, the number of events to fire, etc. Before testing an application, test engineers are required to configure such options (*MFTSet*) and choose the preconditions of the application under test (*AUTPr*), that is the state of the application under test (*AUT*) before the test run. Once the tool is configured, it starts generating random events and sending them to the (*AUT*) that will reach a new state S. The process stops when a termination condition is satisfied. This behavior can be described by the generic algorithm reported in Figure 1.

```
1: procedure GenericRandom(AUT, MFTSet, AUTPr)
2:     S ← Launch(AUT, AUTPr)
3:     Termination ← FALSE;
4:     while (!Termination) do
5:         nextEvent ← Selection(S, MFTSet);
6:         S ← Execute(S, nextEvent, MFTSet);
7:         Termination ← EvaluateTermination();
8:     end while
9: end procedure
```

Fig. 1: Generic GUI Random Testing Algorithm

The termination condition may be based on aspects of the process, such as the number of events that have been fired or the amount of time spent testing, or it may be based on some adequacy measurement that determines whether sufficient testing has been executed. For instance, when using the statement coverage criterion as an adequacy measurement, we can stop testing if all the statements have been executed, or the percentage of executed statements is greater than a given threshold [12].

The choice of the termination condition is always a relevant problem with automatic testing processes, since it is able to affect their effectiveness and cost. This stop condition is even more important with random testing techniques, which are

notoriously affected by problems of reliability and efficiency. The reliability problems of random testing depend on the randomness of this technique: if the same tool is launched multiple times, even from the same initial preconditions, the testing results may sensibly differ, due to the randomness of the sequence of events sent to the application. On the other hand, the inefficiency problems of random testing depend on the risk of wasting excessive testing effort if we try to achieve non-reachable test adequacy levels. It is well-known indeed that, any testing method is affected by a *Saturation Effect*, that is the tendency of the method of limiting its ability to expose faults in a program under test [13]. After reaching this limit, continuing testing the same method may cause significant waste of testing efforts. This limit is also called a *Saturation Point* in the literature and several authors tried to exploit it to define a termination point of testing [14].

These two problems are well illustrated by Fig. 2, which reports the code coverage percentage (as the number of sent events grows) of an example application that was achieved by two different testing runs of the same MFT tool. In each run we used the same tool configuration and started from the same application preconditions, but used different seeds for generating different sequences of random events.

As the figure shows, there is an initial unreliability zone (instability phase) of the process (between 0 and about 3,000 events) where the testing runs achieve different and floating coverage results. The code coverage achieved by the former run (represented by the continuous line in the figure) is indeed initially lower than the latter's ones (represented by the dashed lines), but there is a trend inversion after about 500 events. On the other hand, after about 4,700 fired events, the coverage degrees of the two runs seems to converge and to reach a zone showing a possible saturation effect.

This code coverage trend yields to an important consequence. If the tester stopped the tool after less than 4,000 events, the test adequacy would be different depending on the considered run. Vice-versa, after firing more than 4,000 events, the coverage results of different testing runs tend to be similar and independent of chance. However, if the tester continues and fires more than 4,700 events, they will do nothing but waste both time and computing resources due to the saturation effect. Ideally, a tester would be able to identify the number of events large enough to overcome this instability zone and small enough to avoid entering the inefficiency zone. The tester could then stop the process at a point that represents an optimal trade-off between effectiveness and cost. Since beyond this point there is no an improvement of the testing adequacy, one could consider this point the app's saturation point according to the definition given by Sherman et al. [14].

III. The Testing Process

In this section, we present a testing process based on MFT techniques that automatically stops at a potential saturation point of the process, defined as the point in testing where additional fired events result in no improvement in test adequacy. In order to reach this objective, our implementation utilizes

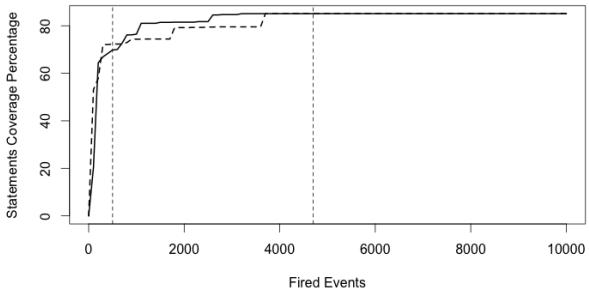

Fig. 2: Code coverage of two testing runs of an MFT tool

a pool of random testing *sessions*. All sessions in the pool are given the same initial conditions, but are fed a different seed and sequence of consecutive random events to fire on the AUT. Exploiting the predictability property of random testing shown by Arcuri and Briand [11], one can accurately infer that there is a point P of this process where the difference in code coverage achieved by all sessions is equal to zero. We suggest this point of least difference may represent a saturation point of testing, indicating that all sessions reached the same test adequacy. On the basis of this property, we further propose that such a point is the application's optimal termination point. Of course, this calculated difference is dependent of the chosen pool of test sessions. The smaller the number of considered sessions, the greater the probability their code coverage be coincident at a *premature* saturation point. Vice-versa, the greater the number of sessions, the greater the likelihood they converge at an *authentic* saturation point. In the sub-sections that follow, we present our test process implementation in greater detail.

A. The Testing Process Implementation

The testing process description requires the following definitions:

- *AUT*: it is the application under test.
- *AUTPr*: it is the set of *AUT* preconditions.
- *MFT*: it is a Monkey Fuzz Testing tool configured according to the *MFTSet* settings.
- S_i: it is a testing **S**ession of the MFT tool that sends a sequence of consecutive random events to the *AUT*.
- $S = \{S_1, \ldots, S_k\}$: it is a set of $k > 1$ testing sessions S_i. All the sessions start from the same initial state of the AUT but have different seeds.
- *SST*: it is the **S**et of executable **ST**atements composing the source code of the *AUT*.
- $SCS(S_i, n)$: it is the **S**et of **C**overed **S**tatements of the application under test, after n events fired in S_i.
- $CSP(S_i, n)$ it is the **C**overed **S**tatements **P**ercentage achieved by the session S_i after n random fired events. It is expressed by the following formula:

$$CSP(S_i, n) = \frac{\mid SCS(S_i, n) \mid}{\mid SST \mid} \times 100 \qquad (1)$$

- *CSSC*(S, n): it is the **C**umulative **S**et of **S**tatements **C**overed by the testing sessions belonging to S after n fired events. It is defined by equation (2):

$$CSSC(S, n) = \bigcup_{i=1}^{k} (SCS(S_i, n)) \tag{2}$$

- *CCSP*(S, n): it is the **C**umulative **C**overage **S**tatement **P**ercentage reached by the testing sessions belonging to S after n fired events. It is defined by equation (3):

$$CCSP(S, n) = \frac{\mid CSSC(S, n) \mid}{\mid SST \mid} \times 100 \tag{3}$$

- *TerCond*(S, n): It is a predicate that is true after n events, if each session of S reached a statement coverage percentage that is equal to the cumulative one. In other words the predicate is true when all the sessions have actually covered the same statements of the *AUT*. It is evaluated by means of equation (4).

$$\begin{aligned} &TerCond(S, n) = TRUE \iff \\ &CSP(S_i, n) == CCSP(S, n) \; \forall \; S_i \; \in S \end{aligned} \tag{4}$$

- *TerP*(*TP*): it is the **Ter**mination **P**oint of the testing process *TP* representing the minimum number of events at which the termination condition is verified.
- *TerL*(*TP*): it is the **Ter**mination **L**evel indicating the cumulative coverage statement percentage reached by the testing process *TP* up to *TerP*.

```
1: procedure TESTINGPROCESSEXECUTION(AUT, AUTPr, MFT, MFTSet, k)
2:     S[] ← initSessions(AUT, AUTPr, k);
3:     terCon ← FALSE;
4:     samplingStep ← STEP;
5:     fe ← 0;
6:     while (!terCon) do
7:         fireNextEvent(AUT, MFT, MFTSet, S[]);
8:         fe++;
9:         if (fe == samplingStep) then
10:            SCS[] ← evalCov(S[]);
11:            CSSC ← evalCCov(SCS[]);
12:            terCon ← evalTerCon(CSSC, SCS[]);
13:            samplingStep ← samplingStep + STEP;
14:        end if
15:    end while
16:    stopSessions(S[]);
17:    terP ← fe;
18:    terL ← evalPercCov(AUT, CSSC)
19: end procedure
```

Fig. 3: Testing Process Algorithm

The Testing Process *TP* is a quintuple (*AUT*, *AUTPr*, *MFT*, *MFTSet*, k). The process is iterative and requires the periodic monitoring of the statement coverage percentages of k random sessions with a predefined sampling step. It is described by the pseudo code in Fig. 3. At each iteration, each session S_i had fired a number fe of events. The algorithm relies on the variables and method's invocations described below.

1) Constants and Variables:

- *terCon*: it is a boolean variable assuming the value of the *TerCond*() predicate.
- *samplingStep*: it is an integer variable > 0, representing the sampling step, i.e., the number of fired events after which the termination condition will be evaluated.
- *STEP*: it is a integer constant > 0 defining the sampling period of the algorithm.
- *fe*: it is an integer variable representing the number of events that have been fired by all the sessions at a given iteration of the algorithm.
- *k*: it is an integer representing the number of testing sessions executed by the testing process.
- *S[]*: it is an array of k testing sessions.
- *SCS[]*: it is an array of statement sets. The i_{th} element of this array represents the set of statements that have been covered by the i_{th} random testing session after *fe* fired events.
- *CSSC*: it is the set of statements that have been covered by all the sessions after *fe* fired events.
- *terP*: it is an integer variable related to the termination point.
- *terL*: it is a double variable representing the value of the termination level.

2) Methods:

- *initSessions(AUT, AUTPr, k)*: it launches the execution of k instances of the *AUT* from the same preconditions *AUTPr*, and starts k testing sessions belonging to the array $S[]$.
- *fireNextEvent(AUT, MFT, MFTSet, S[])*: in each testing session of $S[]$, the *MFT* sends a random event to the *AUT* according to its settings *MFTSet*.
- *evalCov(S[])*: it evaluates the set of statements that are covered by each session of $S[]$.
- *evalCCov(SCS[])*: it evaluates the cumulative set of statements *SCS*[] that have been covered by all the testing sessions.
- *evalTerCon(CSSC,SCS[])*: it computes the covered statement percentage reached by each testing session and the cumulative coverage percentage. Then, it evaluates the predicate described by equation (4).
- *stopSessions(S[])*: it stops the execution of the testing sessions belonging to $S[]$.
- *evalPercCov(AUT,CSSC)*: it evaluates the cumulative coverage statement percentage of *AUT* statements at the end of the process execution.

B. The Testing Infrastructure

We now present the developed software infrastructure used to execute our testing process. This implementation targets the Android mobile platform.

The infrastructure includes two types of components, namely *Testing Process Coordinator* and *Testing Session Executor*. The former component is responsible for starting, ending, and managing the results of the testing sessions'

execution. The latter component is in charge of running the random testing sessions on the emulated Android platforms and collecting data resulting from them. At run time, just a single instance of Testing Process Coordinator is needed, while many instances of the Testing Session Executor component can potentially be deployed and run on different nodes of a distributed architecture. Fig. 4 shows an example infrastructure, including two Testing Session Executor components.

Each Testing Session Executor component includes three software modules: Android Emulator, Driver and Loader. The Driver component implements the specific MFT technique and iteratively sends the next random user event to the GUI of the subject application. The Android Emulator provides the emulated execution platform and consists of the Android Virtual Device (AVD) provided by the Android SDK [2]. The instrumented Application Under Test is run on the AVD under the control of a Robot component that actually fires the event on the current GUI and scrapes it.

The AUT is instrumented by the EMMA Library [3] in order to generate Code Coverage Files. The Loader module fetches these files from the AVD and provides them to the Coverage Repository that is deployed on the Testing Process Coordinator component of the architecture. EMMA is an open-source toolkit for measuring and reporting Java code coverage. It supports coverage types such as class, method, line and basic block. Moreover, EMMA can detect when a single source code line is partially covered, which can happen when the source code has branches that are not exercised by the tests [4]. Referring to the algorithm in Fig. 3 the $i-th$ element of $SCS[]$ is actually the EMMA run-time coverage data (which basic blocks have been executed) that are stored in files having *.ec* extensions. To obtain the $CSSC$ we exploited the $merge$ feature provided by EMMA.

As to the Process Coordinator component, it includes an Engine module that launches the testing sessions on the different Testing Session Executor components, gets their coverage results, and periodically assesses the termination condition. To evaluate the termination condition, it runs Emma scripts to compute the code coverage percentages reached both by the sessions and their union. It then executes scripts to evaluate whether the termination condition has been reached. While the termination condition is not true, the Engine commands the Driver to send further random events, otherwise it stops all running sessions.

This architecture was implemented using Java technologies as well as features provided by the Android Debug Bridge (ADB) [5].

Since the implementation of Driver and Robot modules depend on the MFT technique involved in the process, we developed two versions of the Driver module that implement

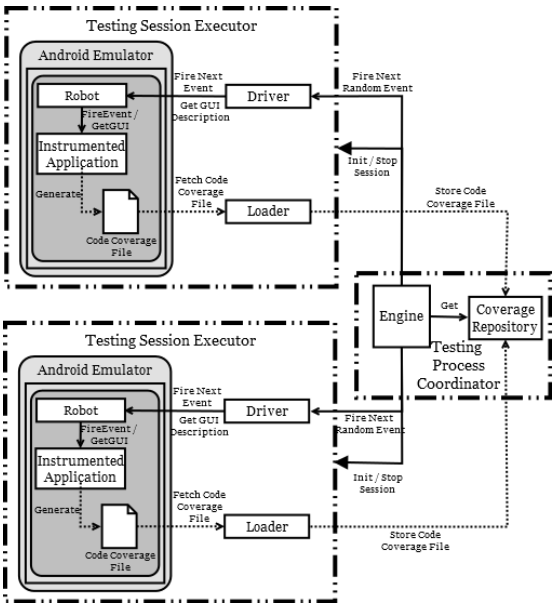

Fig. 4: Overview of the overall testing infrastructure

the fuzz testing technique used by the AndroidRipper tool [10] and the one exploited by the Android Monkey tool [6], respectively. The first version of the Driver directly delegates the Robot component to interact with the AUT by means of the APIs provided by the Android Instrumentation library [7]. Specifically, the Robot exploits the Robotium library [8] both to fire events on the AUT and to get an instance of its GUI at run-time. The second version of the Driver was implemented using the androidmonkey library [9]. It is a copy of the original Android Monkey Tool and is a library made for testing and analysis purposes. In this version, the Driver component runs test scripts that invoke the Robot, which in turn runs JUnit test cases and exploits the designated library to send event(s) to the AUT.

IV. An exploratory study

This section presents a preliminary case study which assesses the feasibility of the proposed testing process and its capability of reaching the saturation effect. To this aim, we used the proposed testing process and infrastructure to test a real Android application named SimplyDo. This app is medium sized (1,281 LOC) and provides a simple shopping and TODO list manager for Android. During the process, we used the MFT technique implemented by the AndroidRipper tool. In its basic configuration, this tool is able to fire events on GUI widgets having at least one event handler registered for the event with a delay of $1000ms$ between consecutive events.

[2]Android Standard Development Kit, https://developer.android.com/sdk/index.html

[3]EMMA: a free Java code coverage tool, http://emma.sourceforge.net/index.html

[4]http://emma.sourceforge.net/faq.html#q.fractional.examples

[5]Android Debug Bridge, http://developer.android.com/tools/help/adb.html

[6]Monkey, http://developer.android.com/tools/help/monkey.html

[7]Android Testing Fundamentals: Instrumentation, http://developer.android.com/tools/testing/testing_android.html\\\#Instrumentation

[8]Robotium, http://code.google.com/p/robotium/

[9]AndroidMonkey, https://code.google.com/p/androidmonkey/

In the first phase of the study, we performed the process multiple times. Each time we used the same AVD configuration but either different preconditions for the subject app or different MFT tool configurations. More specifically, we defined two different tool configurations, C1 and C2, and two different app preconditions, P1 and P2. C1 and C2 are defined as follows:

- C1: Given a GUI with a ListView widget, Android Ripper fires 'click' events only on the first three items of the list.
- C2: Given a GUI with a ListView widget, Android Ripper fires 'click' events on all of the list items.

Similarly, we defined the two preconditions, P1 and P2, as:

- P1: SimplyDo has been installed on the device, but has never been launched.
- P2: SimplyDo has been launched, and contains two TODO lists and one item in the first TODOs list.

Table I reports the four combinations of tool settings and app preconditions we considered for each process run.

TABLE I: Testing process variants and results

TP	MFTSet	AUTPr	TerL	TerP
TP11	C1	P1	76.56%	4,000
TP12	C1	P2	76.32%	3,800
TP21	C2	P1	85.1%	4,700
TP22	C2	P2	84.63%	4,300

For each variant, we ran $k = 12$ testing sessions in parallel, where each session was executed on a different PC. The automatically obtained $TerP$ and $TerL$ values for each variant are reported in Table I. In order to obtain a more complete view of each process's trends, we ran up to 10,000 events for each session to observe coverage even after the application had reached its termination point. Figure 5 illustrates the obtained statement coverage percentage trends for each of the variants.

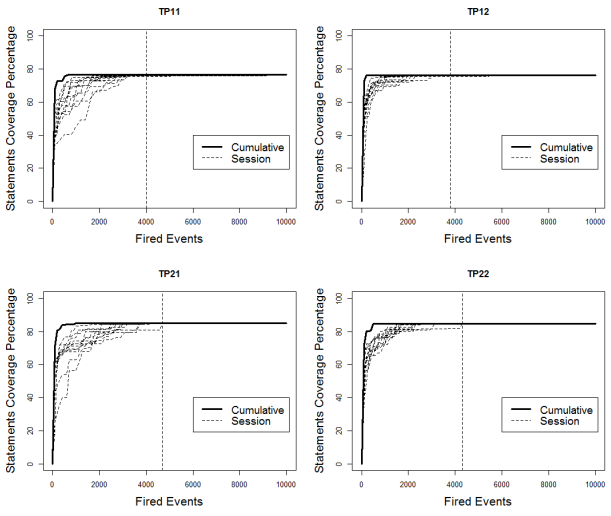

Fig. 5: Coverage trends of the four testing process variants

The obtained trends seem to suggest that all processes reached the saturation effect. To confirm this datum, we manually analyzed both the code coverage reports produced by the Java Code Coverage Tool EMMA and the part of the application's code left out by the sessions. The aim of our analysis was to decipher whether the resulting uncovered code was in fact reachable. If so, it would seem that the termination points were *not* all saturation points. On the other hand, if the uncovered code was found to be *unreachable*, we could conclude that every process's termination point was also its saturation point and our hypotheses would still hold.

At the end of the analysis, we observed that in fact all the uncovered statements could never be exercised by the testing processes due to the following reasons:

1) MFT tool configuration: Some code was not reachable given the MFT tool configurations. For example, because AndroidRipper was not configured to fill in the EditText widgets with null values, the related *NullPointerException* handling code could not be covered. Moreover, since AndroidRipper did not use the keyboard device to fill the EditText widgets, the related *onEditorAction* handlers were actually unreachable. Eventually, the C1 settings used in TP11 and TP12 were able to fire events only on the first three items of any menu list, and then the code associated with the other menu items could never be triggered.

2) AVD configuration: We also witnessed that part of the app's source code could never be executed due to the settings of the AVD. For example, the code that should be executed when the device does not include the SD card was not reachable because the AVD used in the testing environment was equipped with an *emulated* SD card.

3) AUT preconditions: Another set of statements were not reachable on the basis of the AUT preconditions. For example, in the process variants TP12 and TP22 that launched the app from the P2 precondition, three LOCs executing *CREATE TABLE* SQL queries were never exercised. These queries are executable only when the app is first launched after iinstallation.

4) Unreachable code of the AUT: The remaining uncovered statements were unreachable, because there was no control flow path to it from the rest of the program [15]. As an example, we found classes and methods included in the source code but never used by the rest of the program.

As a result, we can conclude that all reachable code of the AUT was covered by the testing processes, and therefore, saturation was *always* reached at a process's termination point.

To further confirm our results, we analyzed whether the choice of $k = 12$ sessions composing the process had influenced the obtained results. To this aim, we post-processed the coverage data and evaluated the termination points proposed by processes made of k=2, 3, 4 and 8 sessions. We permuted without repetitions the 12 testing sessions obtaining 66 simulations of testing processes made by $k = 2$ testing sessions, 220 testing processes made by $k = 3$ testing sessions, 495 testing processes made by $k = 4$ testing sessions and 495 testing processes made by $k = 8$ testing sessions. We obtained that the termination levels were always the same for testing processes composed of $k > 2$ sessions, and conclude

that $k = 12$ sessions is an adequate choice to obtain reliable process results.

Lastly, in order to assess whether the process is able to reach the saturation independently of the exploited MFT technique, we performed another testing process involving the same application but a different MFT technique implemented by Monkey Tool. It fires events belonging to different classes of User Events and System Events chosen at random on the basis of a given adjust percentage. User events includes touchscreen events that are fired on randomly chosen points of the screen irrespective of the actual presence of a UI widget in that point. We executed a single testing process made by twelve random testing sessions, starting from the initial P2 state of the AUT and configured the tool with its default adjust percentages. Moreover, we set at $100ms$ the delay between consecutive events.

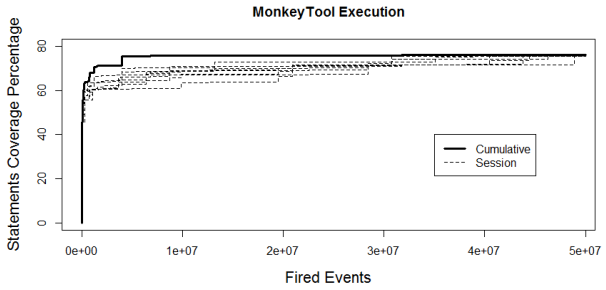

Fig. 6: Coverage trends obtained by Monkey Tool

Figure 6 reports the Statement Coverage Percentage trend achieved by the process, where $TerP = 5 \times 10^7$ and $TerL = 75.57\%$. The manual analysis of the code statements left uncovered by the process confirms the uncovered code is in fact unreachable due to the same motivations listed above. This result, therefore, further confirms that the proposed process reached the saturation effect independently from the considered MFT technique, even if this effect was reached after many more events than the former process (after about 5×10^7 events rather than about $4,000$ events).

V. Experimentation

To extend the validity of the results achieved by the exploratory study, we conducted an experiment aimed at answering the two Research Questions reported below.

R.Q.1: Is the proposed testing process able to reach the Saturation at the termination point?

R.Q.2: Which are the main factors able to affect the effectiveness of the proposed testing process at the termination point?

We addressed these research questions in the context of Android mobile applications, using the MFT techinque implemented by AndroidRipper.

A. Subjects

For this experiment we selected 18 open source Android applications. We chose apps published on the Google Play store, belonging to different Google Play categories and having different source code complexity, expressed in terms of number of statements. Table II reports for each application an identifier (**AUT ID**), its name, Google Play Category (GPC) and the number of source code statements given by $\mid SST \mid$. As data show the AUTs belonged to 12 different GPCs and their size varied from 184 to 3860 LOCs.

TABLE II: Characteristics of the Applications Under Test

AUT ID	AUT Name	GPC	|SST|
AUT1	AardDict	Book	2,097
AUT2	AndroidLevel	Tools	623
AUT3	BatteryCircle	Tools	249
AUT4	BatteryDog	Tools	463
AUT5	Bites	Lifestyle	967
AUT6	Fillup	Transportation	3,807
AUT7	JustSit	Lifestyle	273
AUT8	ManPages	Productivity	292
AUT9	MunchLife	Entertainment	184
AUT10	NotificationPlus	Productivity	283
AUT11	Pedometer	Health	809
AUT12	QuickSettings	Productivity	2,841
AUT13	Taksman	Tools	226
AUT14	TicTacToe	Brain	493
AUT15	TippyTipper	Finance	999
AUT16	Tomdroid	Productivity	3,860
AUT17	Trolly	Shopping	364
AUT18	WorldClock	Travel	1,149

B. Metrics

To assess the Saturation effect at the termination point of the process we decided to measure the residual percentage of code statements left uncovered by the testing process until the termination point. If the Saturation is reached then this residual quantity is zero, or approximately equal to zero.

To evaluate this residual percentage, we used the following sets and metrics:

- $UnS(TP)$: it is the **Un**reachable **S**tatements set, $UnS(TP) \subseteq SST$ of the AUT, made of all the AUT statements that are unreachable by the process TP.
- $ReSRS(TP, TerP)$: it is the **Re**sidual **S**et of **R**eachable **S**tatements by TP at the termination point. It represents the set of statements that are potentially reachable by TP but have not been covered by it until $TerP$. This set is given by the following difference:

$$SST - CSSC(TP, TerP) - UnS(TP). \quad (5)$$

- $RePRS(TP, TerP)$: it is the **Re**sidual **P**ercentage of **R**eachable **S**tatements. It is given by equation (6)

$$RePRS(TP, TerP) = \frac{\mid ReSRS(TP, TerP) \mid}{\mid SST \mid} \times 100 \quad (6)$$

If $RePRS(TP, TerP) = 0$ we can say that all the reachable code of the application has been actually covered by TP, so the process reached the saturation.

TABLE III: Experimental Results

AUT ID	TerL	RePRS	AUT ID	TerL	RePRS
AUT1	71.14%	0%	AUT10	41.24%	0%
AUT2	62.68%	0%	AUT11	74.75%	0%
AUT3	92.89%	0.40%	AUT12	48.50%	0%
AUT4	81.60%	0.42%	AUT13	92.79%	0%
AUT5	57.88%	0%	AUT14	99.64%	0.17%
AUT6	84.03%	0%	AUT15	87.86%	0%
AUT7	70.92%	0%	AUT16	69.96%	0%
AUT8	77.53%	0%	AUT17	80.88%	0%
AUT9	98.86%	0%	AUT18	97.37%	0%

C. Experimental Procedure

We had to configure the processes to test each subject application. Each process included $k = 12$ sessions and we used the same AndroidRipper configuration to test all the AUTs. AndroidRipper was configured for sending events on the GUI widgets having at least a registered listener and for emulating the pressure both of the *back* button and of the *openMenu* one of the mobile device. We set a delay of $500\ ms$ between two consecutive events and configured AndroidRipper for filling in the EditText widgets with random numeric values. Moreover, for each AUT we defined a specific initial precondition and we used the same AVD configuration in each process execution: the emulated devices were all equipped with Android Gingerbread (2.3.3) and have 512 MByte of RAM and 64 MByte of memory on emulated SD Card. The processes were executed by exploiting the testing infrastructure that was configured to run on 13 different PCs running Windows 7 64 Bit Operative System equipped with an Intel I5 3GHz processor and 4GB of RAM.

To answer the first research question, at the end of each process we measured the $RePRS$ values. To measure this metric we evaluated the $UnS(TP)$ set by means of a manual analysis of the statements uncovered by each TP. To answer the second research question, we assessed the effectiveness of the process at the termination point by evaluating its Termination Level $TerL$. Moreover, we looked for the motivations that did not allow the complete coverage of the source code statements. This research was made by manual analysis too.

To be confident about the results of the manual analyses, they were performed by two different teams of software engineers (each one including a Ph.D. student and a graduate student), and then the obtained results were validated by a third team including two researchers in software engineering.

D. Results

Table III reports for each AUT the termination level $TerL$ and the residual percentage of reachable statements $RePRS$ achieved by the performed testing processes.

1) Saturation results: As the data show, 15 times out of 18 the residual percentage of reachable statements was 0%. In the remaining three cases, this percentage was negligible, being lower then 0.5%. As to AUT3, the RePRS was 0.40%, indicating that a single line of code over 249 of its SST was potentially reachable, but it was not actually reached. As to AUT4, the residual percentage of reachable statements was 0.42% due to only two potentially executable but not actually executed statements over 463. In regards to AUT14, the residual percentage of reachable statements was even smaller, 0.17%, with only 0.8 potentially executable statements that were not covered during the process.

On the basis of these results we could answer **R.Q. 1** and conclude that the proposed testing process was able to reach the Saturation at the termination point in all the considered cases.

2) Effectiveness results: We analyzed in detail the uncovered code of the AUTs with the aim of understanding why this code was not reached during the process. We found the motivations reported below.

(1) *Part of the code was not reached depending on the initial state of the applications under test*. This motivation was true for two applications, namely Bites and FillUp. Since the considered preconditions set them in a state *successive to the first AUT execution on the device*, then the statements of some SQL queries needed to configure the supporting databases was actually unreachable. This code can be indeed executed only when these applications are launched for the first time after their installation on the device.

(2) *Part of the code was not reached depending on the configuration of the device we exploited in the testing processes*. This motivation was verified for 8 applications out of 18. We found code statements that could be executed only if the apps were installed on specific devices, i.e., the ones belonging to the *Motorola* and *eInk Nook* families. Some other statements were unreachable because they can be executed only on Android O.S. platforms different from the 2.3.3 version. Other statements were made unreachable by the hardware limitations of the device emulator, i.e., the absence of sensors, WiMAX connectivity, Bluetooth and a physic LED and the impossibility of changing the connection status (Wi-Fi on/off, 3g on/off). Part of the source code was unreachable because some specific apps, like GMail, were not present on the device emulator. Eventually, some statements could not be reached because they can be triggered only when the device does not present an SD card, while we equipped the emulator with an SD.

(3) *Part of the code was not reached depending on both the settings and the limitations of the MFT tool*. This motivation was verified for 11 applications out of 18. The MFT tool was configured to fill in the editText widgets with random integer values. As a consequence, some statements whose execution requires specific user input values (such as, a valid e-mail address or valid URLs) could never be covered. Parts of the statements were actually unreachable, given the limitations of the MFT tool that is not able to fire all the types of event handled by the subject apps. As an example, AndroidRipper is not able to emulate the pressure of some device buttons (such as Volume Up, Volume Down and Search), to interact with the device Trackball, to fire specific events like the gesture ones, to emulate the changes of the values read by sensors, to send Intents, to interact with some widgets like Preferences and webViews.

(4) *Part of the code was not reached because it is actually unreachable code of the AUT*. This motivation was verified for all the considered applications. Some apps presented activities declared in the manifest.xml but never opened, menus defined but never enabled, or classes that are never instantiated. In an application, there was a fraction of source code related to the creation of XML files that could never be executed since the AUT lacked of the permits for writing on the SD Card. In some applications, there were parts of code related to the interaction with external services that are not actually available.

In conclusion the motivations that emerged from this analysis coincided with the same ones revealed by the exploratory study, so we could answer **R.Q. 2** by claiming that the main factors affecting the effectiveness of the proposed testing processes were (1) *the preconditions of the AUT*, (2) *the configurations of the testing platform*, (3) *the limitations and the configuration of the MFT technique*, and (4) *the existence of unreachable statements in the source code of the AUT*.

E. Lessons learned

At the end of the experiment, we were able to learn some lessons about the proposed testing process. A first lesson regards its termination criterion. According to it, in the experiment the termination points were reached when each session composing the process reached the same code coverage as the cumulative one. This stop condition allowed the process to reach the saturation effect. Analyzing the experimental data further, we were able to derive an alternative stop condition. The data suggested to us indeed that the process could be stopped, without loss of code coverage, as soon as the coverage of one of the sessions reached the cumulative one. As an example, Fig. (7) shows a zoom in the code coverage trends achieved by the testing process TP22 presented in Section IV and highlights two possible termination points $TerP$ and $TerP^*$, where the second point is obtained by using the new termination condition $TerCond^*$ expressed by equation (7):

$$\begin{aligned} &TerCond^*(S,n) = true \iff \\ &\exists\ S_i\ \in S \mid CSP(S_i,n) == CCSP(TP,n) \end{aligned} \tag{7}$$

To validate this intuition, we evaluated the new Termination points and levels that could be achieved by the new termination condition for each AUT. Table IV shows the obtained results. The termination levels obtained using the new termination condition were the same as the ones reported in Table III. As the data show, the new termination condition significantly reduces the number of events needed to reach the same test adequacy as the one achieved at the saturation point, and the reduction rate varies between 24.48% and 93.58%. This new criterion may be successfully used to improve the efficiency of the process, leaving its test adequacy unchanged.

A second lesson we learned from the exploratory study is that the proposed process is able to reach different Saturation Levels, depending on the preconditions of the AUT and the settings of the MFT technique. This lesson suggests to us a new variant of the testing process that iteratively selects

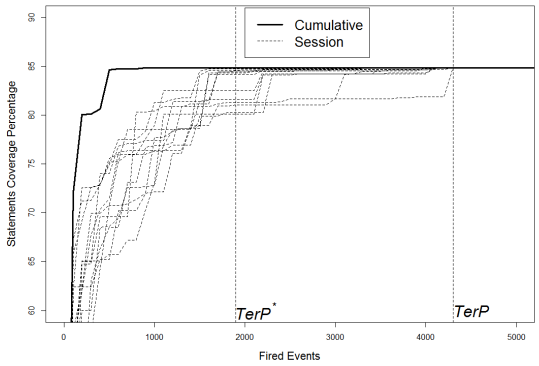

Fig. 7: Two termination points of TP22 execution

TABLE IV: Termination Points

AUT ID	$TerP^*$	$TerP$	AUT ID	$TerP^*$	$TerP$
AUT1	300	4000	AUT10	100	500
AUT2	100	700	AUT11	2300	4600
AUT3	100	500	AUT12	15200	30700
AUT4	100	800	AUT13	300	800
AUT5	7400	9800	AUT14	10300	20100
AUT6	53400	95600	AUT15	500	7200
AUT7	1900	5300	AUT16	98200	147800
AUT8	500	7800	AUT17	200	600
AUT9	300	700	AUT18	1400	3600

different app-tool configurations until the saturation effect is detected. As an example, the tester may create a set of app- and tool- configurations. The new process iteratively picks a configuration and runs the MFT tool until the saturation point. The process ends when the set of app- and tool-configurations is exhausted. This new process may achieve better test adequacy results than the former one. Fig. (8) shows the cumulative coverage trend we obtained by the new testing process where we consecutively run the four testing variants TP12, TP11, TP22 and TP21 described in IV. However, further

Fig. 8: Cumulative Coverage achieved in the new process variant

experiments should be performed in order to assess the validity of these two process variants.

VI. Threats to Validity

In this section we discuss some potential threats to the validity of our studies and how we tried to limit them.

Threats to external validity, i.e. threats to the generalization of the conclusions. A first threat to the external validity is related to the subject selection. We tried to limit this threat by selecting 18 Android applications that are different in size and type and that can be considered as a meaningful sample of real world Android applications. A second threat is related to the generalization of our conclusions to other configurations of the MFT tool and to other MFT tools. The exploratory study presented in section IV partially addressed this threat by exploiting two different configurations of AndroidRipper and another MFT tool.

Threats to internal validity, i.e. threats due to external factors that are not under control. In order to limit this threat we carried out the experiments in the environment described in subsection V-C in which we fixed and controlled all the characteristics of the Android execution environment and the application preconditions, too. There are no threats related to uncontrolled external resources because the selected applications had dependencies only on static external resources.

The threats to validity related to the dependency of the results on the number of testing sessions have been addressed in section IV with respect to the SimplyDo application for which we concluded that the obtained termination level was always the same if $k > 2$ testing sessions were considered.

Threats to construct validity, i.e. threats due to measure accuracy. A possible threat to construct validity is related to the accuracy of our manual evaluation of the set of unreachable statements of each subject app. In order to limit this threat, we repeated the evaluation three times for each app, each one using a different team including graduate students, Ph.D. students and researchers, all skilled in Android development and testing.

VII. Related Works

Nevertheless in the past random testing was considered less promising than systematic testing techniques [16], many other works in literature empirically demonstrated their effectiveness in many different contexts [17], [18], [19], [20], [21]. Arcuri et al. [11] addressed random testing from a theoretical point of view by proposing a mathematical model for describing the effectiveness of random testing and comparing it against partition testing. Moreover they presented some novel theoretical results regarding effectiveness, scalability and predictability of random testing. This work focused on testing techniques that randomly choose input values from the input domain, while our work focuses on techniques that choose events from the event domain.

Several studies in literature studied the predictability of the performance of random testing techniques. Ciupa et al. [22] [23] explored the predictability of random testing in terms of fault detection capability in the context of object-oriented programs written in Eiffel. More recently, Furia et al. [24] have searched for a law able to relate the number of executed random test cases and the number of found faults and failures in Eiffel and Java applications. Sherman et al. [14] exploited the existence of a saturation effect occurring when the increase in coverage over a window of test runs (of a given size) is less than a fixed threshold and used this effect to define new adequacy criteria in concurrency testing.

The use of code coverage as an indicator of test adequacy is very common in literature, in particular when the total number of existing faults is unknown and when using testing strategies that are able to found only some typologies of failures (e.g. only crashes). Several empirical studies in literature demonstrated the existence of a positive correlation between code coverage and fault effectiveness of a test suite [25], [26], [27], [28], [29]. Only Wei et al. [30] demonstrated, on the contrary, in the context of Eiffel programs and design by contract development, that code coverage is not always sufficiently correlated with the fault detection capability.

Besides Monkey and AndroidRipper that have been already described in the paper, Dynodroid [31] implements a MFT technique that is able to fire at random just 'relevant' events: an event is relevant if there is at least an application event handler registered to that event. Hao et al. [32] propose PUMA, a MFT tool that can be customized by means of a scripting language called PumaScript in order to support different testing tasks in the context of Android applications. Liu et al. [33] propose an Adaptive Random Testing approach for Android applications, obtained by migrating the proposal of Chen et al. [34] to the mobile context in which inputs are represented by event sequences. This technique mixes the aspects of a random uninformed testing technique with the heuristic optimization proposed by an Adaptive Random Testing technique. Another relevant contribution in the context of mobile applications is the MFT tool Vanarsena [35] that is able to automatically find crashes in the context of Windows Phone applications.

VIII. Conclusions and Future Works

In this paper we addressed the problem of stopping a random testing process at a cost-effective point, where test adequacy is maximized and no testing effort is wasted. We presented a fully automatic MFT process that is able to find this point by exploiting the *saturation effect* and the *predictability property* of random testing techniques. The process is supported by a a software infrastructure we developed for testing Android mobile applications. The validity of the approach in finding saturation points has been shown by an experiment where we tested 18 real Android applications. An analysis of the experimental results allowed us to understand which factors are affecting the effectiveness of random testing processes. As future works we plan to improve the experimentation in order to extend the validity of the proposed approach. In particular, to extend the validity of this approach in the mobile context we propose to exploit the process for testing a wider number of AUTs by means of other MFT tools even for different mobile operating systems. Moreover, we'd like to apply this approach in other contexts, i.e., web or desktop applications.

REFERENCES

[1] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Trans. Softw. Eng.*, vol. 10, no. 4, pp. 438–444, Jul. 1984. [Online]. Available: http://dx.doi.org/10.1109/TSE.1984.5010257

[2] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of windows nt applications using random testing," in *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, ser. WSS'00. Berkeley, CA, USA: USENIX Association, 2000, pp. 6–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267102.1267108

[3] K. Sen, "Effective random testing of concurrent programs," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 323–332. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321679

[4] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[5] B. P. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of macos applications using random testing," in *Proceedings of the 1st International Workshop on Random Testing*, ser. RT '06. New York, NY, USA: ACM, 2006, pp. 46–54. [Online]. Available: http://doi.acm.org/10.1145/1145735.1145743

[6] T. Wasserman, "Software engineering issues for mobile application development," *FoSER 2010*, 2010.

[7] H. Muccini, A. Di Francesco, and P. Esposito, "Software testing of mobile applications: Challenges and future research directions," in *Automation of Software Test (AST), 2012 7th International Workshop on*, june 2012, pp. 29 –35.

[8] N. Nyman, "Using monkey test tools," *oftware Testing & Quality Engineering Magazine*, pp. 18–21, 2000.

[9] M. R. Lyu *et al.*, *Handbook of software reliability engineering*. IEEE computer society press CA, 1996, vol. 222.

[10] D. Amalfitano, A. Fasolino, P. Tramontana, B. Ta, and A. Memon, "Mobiguitar – a tool for automated model-based testing of mobile apps," *Software, IEEE*, vol. PP, no. 99, pp. 1–1, 2014.

[11] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, pp. 258–277, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1109/TSE.2011.121

[12] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, Dec. 1997. [Online]. Available: http://doi.acm.org/10.1145/267580.267590

[13] P. Kapur, H. Pham, A. Gupta, and P. Jha, "Testing-coverage and testing-domain models," in *Software Reliability Assessment with OR Applications*, ser. Springer Series in Reliability Engineering. Springer London, 2011, pp. 131–170. [Online]. Available: http://dx.doi.org/10.1007/978-0-85729-204-9_4

[14] E. Sherman, M. B. Dwyer, and S. Elbaum, "Saturation-based testing of concurrent programs," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 53–62. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595706

[15] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 2, pp. 378–415, Mar. 2000. [Online]. Available: http://doi.acm.org/10.1145/349214.349233

[16] G. J. Myers, *Art of Software Testing*. New York, NY, USA: John Wiley & Sons, Inc., 1979.

[17] S. C. Ntafos, "On comparisons of random, partition, and proportional partition testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pp. 949–960, Oct. 2001. [Online]. Available: http://dx.doi.org/10.1109/32.962563

[18] R. Hamlet, *Random Testing*. John Wiley & Sons, Inc., 2002. [Online]. Available: http://dx.doi.org/10.1002/0471028959.sof268

[19] D. Hamlet, "When only random testing will do," in *Proceedings of the 1st International Workshop on Random Testing*, ser. RT '06. New York, NY, USA: ACM, 2006, pp. 1–9. [Online]. Available: http://doi.acm.org/10.1145/1145735.1145737

[20] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Trans. Softw. Eng.*, vol. 17, no. 7, pp. 703–711, Jul. 1991. [Online]. Available: http://dx.doi.org/10.1109/32.83906

[21] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Experimental assessment of random testing for object-oriented software," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 84–94. [Online]. Available: http://doi.acm.org/10.1145/1273463.1273476

[22] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer, "On the predictability of random tests for object-oriented software," in *Software Testing, Verification, and Validation, 2008 1st International Conference on*, April 2008, pp. 72–81.

[23] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer, "On the number and nature of faults found by random testing," *Softw. Test., Verif. Reliab.*, vol. 21, no. 1, pp. 3–28, 2011.

[24] C. A. Furia, B. Meyer, M. Oriol, A. Tikhomirov, and Y. Wei, "The search for the laws of automatic random testing," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13. New York, NY, USA: ACM, 2013, pp. 1211–1216. [Online]. Available: http://doi.acm.org/10.1145/2480362.2480590

[25] P. Piwowarski, M. Ohba, and J. Caruso, "Coverage measurement experience during function test," in *Proceedings of the 15th International Conference on Software Engineering*, ser. ICSE '93. Los Alamitos, CA, USA: IEEE Computer Society Press, 1993, pp. 287–301. [Online]. Available: http://dl.acm.org/citation.cfm?id=257572.257635

[26] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proceedings of the 16th International Conference on Software Engineering*, ser. ICSE '94. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 191–200. [Online]. Available: http://dl.acm.org/citation.cfm?id=257734.257766

[27] P. G. Frankl and O. Iakounenko, "Further empirical studies of test effectiveness," in *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '98/FSE-6. New York, NY, USA: ACM, 1998, pp. 153–162. [Online]. Available: http://doi.acm.org/10.1145/288195.288298

[28] X. Cai and M. R. Lyu, "The effect of code coverage on fault detection under different testing profiles," in *Proceedings of the 1st International Workshop on Advances in Model-based Testing*, ser. A-MOST '05. New York, NY, USA: ACM, 2005, pp. 1–7. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083288

[29] A. S. Namin and J. H. Andrews, "The influence of size and coverage on test suite effectiveness," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 57–68. [Online]. Available: http://doi.acm.org/10.1145/1572272.1572280

[30] Y. Wei, B. Meyer, and M. Oriol, "Is branch coverage a good measure of testing effectiveness?" in *Empirical Software Engineering and Verification*, ser. Lecture Notes in Computer Science, B. Meyer and M. Nordio, Eds. Springer Berlin Heidelberg, 2012, vol. 7007, pp. 194–212. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-25231-0_5

[31] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 224–234. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491450

[32] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '14. New York, NY, USA: ACM, 2014, pp. 204–217. [Online]. Available: http://doi.acm.org/10.1145/2594368.2594390

[33] Z. Liu, X. Gao, and X. Long, "Adaptive random testing of mobile application," in *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, vol. 2. IEEE, 2010, pp. V2–297.

[34] T. Chen, F. Kuo, R. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," *JSS*, vol. 83, no. 1, pp. 60–66, 2010.

[35] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan, "Automatic and scalable fault detection for mobile applications," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '14. New York, NY, USA: ACM, 2014, pp. 190–203. [Online]. Available: http://doi.acm.org/10.1145/2594368.2594377