Towards Black Box Testing of Android Apps

Yury Zhauniarovich*, Anton Philippov*, Olga Gadyatskaya†, Bruno Crispo*‡, Fabio Massacci*
*University of Trento, Italy; ‡DistriNet, KU Leuven, Belgium
{yury.zhauniarovich, anton.philippov, bruno.crispo, fabio.massacci}@unitn.it
†SnT, University of Luxembourg, Luxembourg
olga.gadyatskaya@uni.lu

***Abstract*—Many state-of-art mobile application testing frameworks (e.g., Dynodroid [1], EvoDroid [2]) enjoy *Emma* [3] or other code coverage libraries to measure the coverage achieved. The underlying assumption for these frameworks is availability of the app source code. Yet, application markets and security researchers face the need to test third-party mobile applications in the absence of the source code. There exists a number of frameworks both for manual and automated test generation that address this challenge. However, these frameworks often do not provide any statistics on the code coverage achieved, or provide coarse-grained ones like a number of activities or methods covered. At the same time, given two test reports generated by different frameworks, there is no way to understand which one achieved better coverage if the reported metrics were different (or no coverage results were provided). To address these issues we designed a framework called BBOXTESTER that is able to generate code coverage reports and produce uniform coverage metrics in testing without the source code. Security researchers can automatically execute applications exploiting current state-of-art tools, and use the results of our framework to assess if the security-critical code was covered by the tests. In this paper we report on design and implementation of BBOXTESTER and assess its efficiency and effectiveness.**

I. INTRODUCTION

The mobile application area is booming. Today there are more than 1 million of third-party applications in Google Play[1] and more than 1 million apps in Apple AppStore[2] markets – and these are only two (though major) players among mobile app markets. In the last several years the ubiquitousness of mobile devices made them a primary target for adversaries. This has forced big application markets to verify apps before publishing them for users. Yet, the developers submit only final application packages for verification [4], [5]. Thus, these untrusted third-party packages for Google's and Apple's vetting systems are essentially black boxes. Therefore, techniques for automatic test generation and execution that can uncover malicious or buggy behaviour are in high demand [6], [23].

The peculiarities of the Android platform (e.g., no central entry point in applications, an event-based model of app development, high dependency on system events and user input, etc.) make the programs written for this platform complicated for black box testing approaches. Despite the growing number of frameworks and tools for black box manual (Robotium [7], Espresso [8], etc.) and automated (Google Monkey [9], Dynodroid [1], PUMA [10], Brahmastra [11], A3E [12], to name just a few) test generation and execution, this area is still in its early development.

The accompanying challenge for the black box test cases development is the problem of discovering and measuring the code coverage of the tests. Without understanding what code parts are covered it is difficult to estimate when test generation and execution could be finished. The coverage challenge is especially acute for security testing. While test suite effectiveness for quality assurance can be evaluated based on metrics other than coverage [13], dynamic detection of malware that hides malicious behaviour or behavioural evaluation of app similarity require special attention to the code coverage [14], [15], [16].

However, even logging which methods have been executed during a run is not a trivial task when app sources are not available. Successful implementations to fulfil this task in Android involve byte-code instrumentation [17] or operating system modification [15]. Measuring other code coverage metrics is also a tough task. Current implementations of app testing systems either work on open-source applications relying on existing tools, such as *Emma* [3], to measure code coverage (e.g., [18], [2]), or operate on instrumented binaries (for example, [12], [19], [14], [20]). At the same time, in the latter case the coverage metrics are usually coarse-grained (e.g., the percentage of shown activities or methods invoked), while the former approach cannot be used in a truly black box environment. Given the discrepancies in code coverage metrics used in the existing black box testing frameworks, it is not trivial for a practitioner to decide which automated testing framework to choose for testing, and how to select test generation strategies based on the time budget available [21].

We address these issues by designing a framework for code coverage analysis. Our framework, called BBOXTESTER, generates coverage reports for further analysis in a black box manner, i.e., having no access to the sources of an app. Our main contributions made in the scope of this work can be summarised as follows:

- We designed and implemented BBOXTESTER – a framework that reports what methods, classes and packages are executed during the testing without the need for source code of the app. Our framework can combine the data obtained during several consecutive runs and produce a cumulative coverage report.
- Our tool measures code coverage metrics (% of basic blocks covered, % of method calls and % classes) without the app source code. It can be used to assess code coverage attained during manual testing or achived by automatic tesing systems.
- BBOXTESTER allows a tester to specify the moment when to start and finish collection of coverage infor-

[1]http://www.appbrain.com/stats/number-of-android-apps
[2]https://www.apple.com/au/ipod-touch/from-the-app-store/

mation. It contains an exception logging mechanism and thus can be also used for bugs detection. We release BBoxTester as open source[3] to drive further research in this area.

- To show that our tool can be easily integrated with testing frameworks we implemented 3 automated testing engines and assessed their effectiveness with BBoxTester on a set of real applications. The engines implement 3 simple strategies for automated testing relying on randomly generated input sequences produced by Google Monkey [9]. Our results for code coverage achieved by these engines are quite interesting. For instance, we show that, given the same time budget available, it is better to try several runs of automated testing on smaller input sequences (1000 events) than 1 or 2 runs with larger input sequences (10000 events). This conclusion contrasts approaches of some state-of-art testing tools that consider only single runs with sufficiently big number of input events (e.g., [22]).
- We studied the bugs detected and recorded by BBoxTester during the evaluation. In our experiments we located 15 different bugs in 13 applications. The analysis shows that even very simplistic automated testing strategies can be very useful in bugs discovery.

The rest of the paper is organised as follows. We introduce some Android details relevant for understanding BBoxTester in Sec. II, overview the design of BBoxTester in Sec. III and its implementation in Sec. IV. We introduce the methodology and app dataset selected for BBoxTester evaluation in Sec. V, and summarise the evaluation results in Sec. VI. We look at the bugs discovered during evaluation in Sec. VII, and discuss the limitations and future extensions of our work in Sec. VIII. Finally, we overview the related work in Sec. IX, and conclude with Sec. X.

II. Android App Background

Android applications are distributed in the form of `apk` files. An `apk` file is a zip archive of app resources that has a predefined structure. It usually contains one or more `.dex` files, an `AndroidManifest.xml` file, `META-INF` directory, and several other inclusions with native libraries, resources and assets files.

Most of Android apps are programmed in Java. Java code is compiled into `.class` files and then transformed into the custom Dalvik bytecode (using the *dx* tool [25]) that is distributed in the form of `dex` files. The file format limits the number of method references inside a single `dex` file to 65K. Yet, some complex apps or those that extensively use different libraries may sometimes reach this limit. To cope with this problem developers use different techniques to reduce the number of method references inside a `dex` file, e.g., they merge several methods into one, process `dex` files with code optimizers, which remove unused routines, etc. Google also proposed a solution for dealing with this problem – developers can divide methods into several `dex` files and use *dynamic class loading* to load additional functionality at runtime.

[3]https://github.com/zyrikby/BBoxTester

`AndroidManifest.xml` is an `xml` file that describes components which constitute an Android app, application parameters, and its privileges in the system. Additionally, the name of an instrumentation class, which allows to monitor interactions of the system with the application under test (AUT for short), is also declared here.

The `META-INF` folder contains data related to the code signing process. Each Android application must be cryptographically signed to be installed. Usually, an app is signed with developer's self-signed certificate. This certificate is used for assurance that the code of the original application and its updates come from the same place, and to establish trust relationships between apps of the same developer. If something is modified in an Android package the signature will become invalid and this app will not be installed on a device.

A. Emma

Emma [3] is an open-source utility for measuring and reporting the code coverage in testing of Java applications, and it is now also included in the Android SDK. *Emma* supports coverage reports at the class, method, line and basic block levels. The core coverage unit is a basic block. All other coverage metrics are derivatives from the ones obtained at the basic block level.

For Java applications *Emma* supports 2 instrumentation modes: the on-the-fly mode that uses a special class loader to instrument classes during the application execution, and the offline mode that instruments all classes in advance. Then *Emma* is used to generate a coverage report after the application testing is over.

However, due to the peculiarities of the Android platform, *Emma* can only instrument Android apps in the offline mode. The standard procedure for that is the following: all `.class` files obtained after the compilation of Java source code are instrumented with *Emma*, and then transformed into Dalvik executables. There is no built-in functionality that allows developers to instrument the Dalvik bytecode directly.

B. Monkey

The Android SDK includes the UI/Application Exerciser Monkey tool [9], or simply Monkey, which is able to generate pseudo-random user actions and feed them in an AUT. It can be used as a standalone tool for the stress testing, or alternatively, be integrated as the trigger of the events in the more advanced systems, which can analyse the results of the Monkey execution. We use Monkey in BBoxTester to run a simplistic automated testing.

III. Framework Overview

BBoxTester is a framework that allows testers to track what code has been executed during testing and to measure fine-grained code coverage metrics of AUTs without access to the source code of the apps. It does not depend on any specific software installed on the mobile device and can be used in functional testing of Android apps. Our framework instruments an application and feeds it to a testing system. This system runs the app and, when the tests are done, BBoxTester

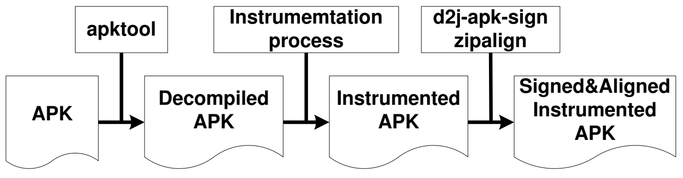

Fig. 1. *Instrumenter* workflow

generates a code coverage report based on the data obtained during application's execution.

Our implementation (in Python) relies on a number of existing well-known tools, such as *apktool* [26], *dex2jar* [27] and *Emma* [3], *dx* [25], *zipalign* [28], *adb* [29], and *Monkey* [9]. However, the modular architecture of our framework allows to substitute the external tools used with their equivalents (e.g., *Emma* to JaCoCo [30], or *dex2jar* to *dare* [31]).

In our current implementation we leverage *Emma* as it is more commonly used in the Android community. The usage of *Emma* allows us to compare our results with the ones already obtained by other researchers, who also usually rely on *Emma*. Additionally, the capabilities of this tool, including its ability to merge code coverage results or to filter out results for some classes (e.g., filter out classes related to advertising libraries), are very useful for researchers and practitioners.

BBoxTester has three main components: *Instrumenter*, *Executor* and *Reporter*. *Instrumenter* is responsible for the instrumentation of an application and allows us to measure code coverage of the app without its source code. The *Executor* component runs the instrumented application on a device and extracts runtime information obtained during the testing. *Reporter* uses the data produced during the instrumentation and execution phases to create code coverage and bug reports (if any bugs were found by BBoxTester during the testing).

Our approach does not require neither modification of the firmware nor phone rooting; only instrumentation of the app to be tested is needed (described in Sec. IV). Developer options must be enabled on the phone to allow BBoxTester to interact with an AUT via *adb*. Additionally, as we modify an app package during the instrumentation, we sign it with a new certificate. This can be a limitation for testing of some apps, e.g., those that check for being repackaged [24] verifying the certificate during execution. We will further discuss this limitation in Sec. VIII.

IV. Design and Implementation

In the following subsections we describe design and implementation details of the main components of our system.

A. Instrumenter

Instrumenter is the main component of BBoxTester. Figure 1 provides an overview of the *Instrumenter* workflow during the instrumentation of an application. Using *apktool*, we decompile an `apk` and extract `AndroidManifest.xml` and the `.dex` files. After instrumentation, the app is compiled back with the same tool into an `apk` file, which is then signed with our certificate and aligned with the *zipalign* tool.

The instrumentation process is presented in Figure 2. *Instrumenter* distinguishes 3 types of files inside the decompiled `.apk` file: the `.dex` files, `AndroidManifest.xml` and, finally, resource files.

As *Emma* cannot instrument the Dalvik bytecode directly, we first transform `.dex` files into Java files. Thus, as the first step, *Instrumenter* retrieves from the decompiled `apk` all files with the `.dex` extension, and processes them using the *dex2jar* utility [27], which transforms the Dalvik bytecode into the Java bytecode. The Java bytecode (in the form of `.jar` files) is then instrumented using the *Emma* tool [3].

After the instrumentation, we recompile the instrumented Java files back to `.dex` files with the standard *dx* utility [25]. We compile the main `classes.jar` file, which corresponds to the `classes.dex` file, together with `emma_device.jar` (to enable Emma's code coverage functionality) and our special instrumentation class file `EmmaInstrumentation.class`. All other files containing code are compiled to `.dex` without additional dependencies (because all necessary dependencies are already in the main code file).

Our instrumentation class `EmmaInstrumentation` is used to track all interactions of the AUT with the rest of the system. We implement it extending the `android.app.Instrumentation` class [32]. When the instrumentation is enabled, this class is initialized before other application components. This class dynamically registers a special broadcast receiver responsible for the invocation of the coverage report generation routine and test finishing. In our system, the event of test finishing and report generation is fired by the *Executor* component (described in the following subsection). Additionally, `EmmaInstrumentation` intercepts and logs the exceptions thrown in the AUT. Together with the context information (also collected by BBoxTester) these data provide a valuable source of information about the bugs in the application.

The instrumentation class must be registered in the `AndroidManifest.xml` file to be recognised by the system. Thus BBoxTester adds to the manifest a special instrumentation tag with the appropriate attribute values (the *name* attribute specifies the full name of the instrumentation class, and *targetPackage* defines the package name of the AUT). So as the instrumentation class stores runtime information (runtime coverage data and bug reports) on device's external storage, the AUT requires the appropriate permission to be able to write there. During the instrumentation of the manifest file this information is also added.

During the implementation of our framework, we found out that `emma_device.jar` contains a set of non-java resources required for proper operation of *Emma* on the Android platform. At the same time, the *dx* utility ignores these resource files. To overcome this limitation, we extract the required files and add them directly to the final `.apk` file.

B. Executor

After an app is instrumented, it is ready for execution on a device. *Executor* facilitates testing process of the instrumented app on a device. This component installs an application, starts

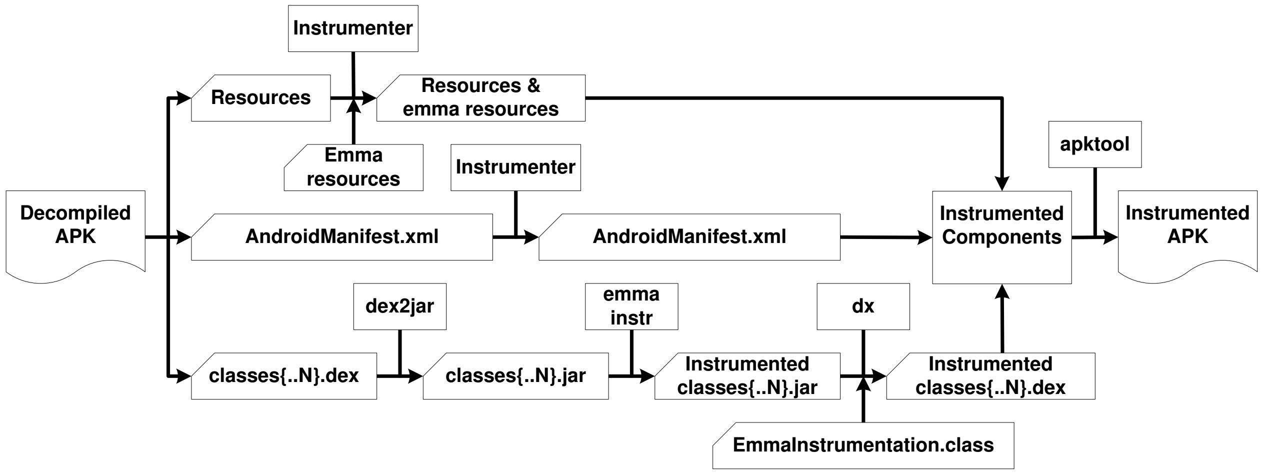

Fig. 2. Instrumentation process

its execution, finishes the run, downloads the results, and uninstalls the app. Between the start and finish events the tester may run her own testing strategies (e.g., the ones proposed in [1], [12], [33], [2]) or may explore the app manually.

During start of the instrumented app, *Executor* takes some parameters passed as intent extras to our developed `EmmaInstrumentation` class. The parameter `reportFolder` specifies the path on the device to store the coverage results and exceptions descriptions (if they occurred during the analysis). The parameter `proceedOnError` defines if the analysis of the app should proceed in case an uncaught exception has been detected, while `generateCoverageReportOnError` sets if runtime code coverage data should be generated in this case (coverage reports based on the data may help to locate the error).

The analyst can stop the testing and generate a final report at any convenient time. All data obtained during the run, including the exception description, runtime coverage reports obtained on exceptions and in the end of the run are stored into the report folder. After the analysis is finished, *Executor* downloads this directory locally on a computer.

C. Reporter

Reporter heavily relies on the functionality provided by *Emma*. To build code coverage reports *Emma* uses two types of files: `.em` files containing metadata information obtained during the instrumentation of the Java code, and `.ec` files comprising runtime coverage data. BBoxTester generates one `coverage.em` file during the instrumentation phase. At the same time, during app execution one or several `.ec` files[4] may be generated. BBoxTester also leverages *Emma*'s ability to merge information from several runtime files to produce cumulative code coverage reports.

BBoxTester reports on the coverage metrics collected by *Emma* at the class, method, basic block and line levels, and marks executed code parts. Unfortunately, full debug information is absent in compiled apps, thus, coverage at the line level is currently missing in our reports. Moreover, for the same reason, the differentiation which code parts have been executed during the testing can be done only at the level of methods and up. In the future work we plan to overcome these limitations.

[4]There can be one final runtime coverage report and/or several reports generated when errors occur.

V. Evaluation Methodology

The BBoxTester evaluation follows two main goals:

- discover how effective and efficient BBoxTester is for measuring code coverage in testing third-party *apk* files,
- investigate and compare several strategies for automated code exploration to explore the best approaches for black box testing and to set the ground for future comparison of automated testing frameworks.

More details on these goals and the evaluation setting follow.

Evaluation of simple code exploration strategies. Google provides the Monkey tool [9] for pseudo-random input events generation that can be useful in automated testing of third-party apps. Moreover, many state-of-art automated testing tools, such as EvoDroid [2], PUMA [10], Dynodroid [1], SwiftHand [33], etc., use Monkey as the baseline to measure improvements in code coverage or bug detection. While these tools can be very effective, they might not be as efficient as Monkey, or they might not be as easy to use for an average developer as Monkey, which is already integrated in the Android SDK. Our hypothesis is that simple automated code-base exploration strategies, such as sequences of pseudo-random events generated by Monkey, can still be useful for app testing, and we use BBoxTester to test this hypothesis. As the measures of usefulness we use the achieved code coverage and the number of discovered bugs in applications (these are the standard approaches in the state of art [1], [12], [11]).

To validate our assumption we implemented automated testing engines implementing 3 simple black box testing strategies and assess them with BBoxTester. The first strategy (further denoted as *mo* for brevity) uses only the Monkey tool, which generates a sequence of pseudo-random UI input events. Monkey does not wait for the results of the events, thus, allowing a tester to inject thousands of actions in a short period of time. The sequence is generated based on a seed value that can be specified using an optional parameter. Additionally, the number of pseudo-random events can also be defined. For each selected number of events (1000, 2000, 3000, 5000, and 10000) we performed 10 experiments. We used a pre-specified seed value to have the ability to repeat experiments with the same input parameters. Notice, that we were still not able to repeat the system state across experiments; thus, sometimes the results of experiments with the same input parameters can be different. In each experiment we used a fixed throttle value (period of time between two successive events) equal 50 ms.

The second strategy, abbreviated *ma*, runs only the main activity of an app. This activity is invoked when a user clicks on the app icon. Routines of the main activity are often used to start other components (e.g., long running services) and, thus, it supposedly contains and invokes a large portion of the code. This consideration justifies launching only the main activity as an automated testing strategy, for example, if the time budget is very limited. We set out to test this conjecture.

The third testing strategy, called *mi*, generates explicit (for activities and services) and implicit or explicit (for broadcast receivers) intents to invoke app components declared in the `AndroidManifest.xml` file, trying to launch all available app components. Implicit intents are not tried for activities, because this action usually calls either system components or other apps, not always the AUT, and services, because Google prohibits the usage of implicit intents to start services [34]. With this strategy we tried to observe if the code coverage achieved by the main activity can be improved/complemented by launching other available app components.

Dataset of apps for testing. To evaluate the simple strategies described above, we would like to compare their results with some state-of-art automated testing/input generation frameworks. We have chosen A3E [12], Dynodroid [1] and SwiftHand [33], because the code coverage in comparison with Monkey and the original app files used for testing were available. Therefore, we treat the three sets of apps reported in the respective papers as our dataset for BBOXTESTER evaluation.

The first dataset has been provided by the developers of the A3E tool [12]. It consists of 31 apps. The second dataset is based on the one used for the Dynodroid [1] validation[5]. We extracted the source code of 52 applications used for assessment of this tool, compiled them and used as the second dataset. The third dataset is based on the apps selected for the SwiftHand [33] tool validation. Unfortunately, the original apk files used for SwiftHand validation were not available for our experiments[6]; only instrumented versions of app packages were provided with the tool. To reconstruct the dataset we manually decompiled the instrumented apks and obtained package names and versions of apps used for the validation. Based on this information, we explored F-Droid and personal webpages of the applications in order to obtain the versions used in SwiftHand's experimental setup. Out of 10 apps used in the original paper we managed to find 6 apk files with the same version, and 3 app source code of the specified version. Out of these 3 apps we managed to build 2. The third app contains deprecated methods and cannot be built for newer API versions.

[5]Available at http://pag-www.gtisc.gatech.edu/dynodroid/data/ at the time of writing.

[6]Personal communication with the authors.

TABLE I. NUMBER OF APPS IN THE DATASETS, NUMBER OF SUCCESSFULLY INSTRUMENTED APPLICATIONS AND TIME TAKEN FOR THE INSTRUMENTATION.

Dataset	# apps	# apps instrumented	Instrumentation time, s
A3E	31	9	702
Dynodroid	52	45	612
SwiftHand	8	6	100
TOTAL	91	60	1414

VI. EVALUATION RESULTS

a) Effectiveness and efficiency of BBoxTester: We executed our instrumentation engine on all applications from the chosen datasets. Table I summarises the results of the instrumentation phase. From the A3E dataset, our tool successfully instrumented 9 out of 31 apps and failed to instrument 22 apps. All instrumentation errors were caused by the third-party tools used in the instrumentation procedure. Specifically, 20 applications caused an exception in the *dx* utility, and 2 applications were not processed correctly by the *zipalign* tool. It took 702 seconds to instrument this dataset.

In the Dynodroid dataset BBOXTESTER was able to instrument 45 out of 52 applications in 612 seconds. The other 7 applications[7] caused the same exception in the *dx* utility. Finally, from the SwiftHand dataset, 6 out of 8 apps were instrumented, and 2 apps[8] raised an exception in the *dx* utility. The time taken to instrument this dataset is 100 seconds. Notice that for all successfully instrumented apps we were able to execute test runs.

Our framework relies on a number of external open tools that process the code differently. Due to the discrepancies in the operation of these tools, not all applications were successfully instrumented. In Section VIII we discuss the failures in instrumentation in more details. As for efficiency, we can see the time taken for app instrumentation is negligible comparing to the actual time of app testing. Moreover, the apps need to be instrumented only once for all testing runs. Thus, this offline processing is not important for efficiency. For the online overhead of code coverage logging, we can rely on *Emma* numbers that report to cause 5-20% of runtime overhead [3]. Overall, we can conclude that efficiency of BBOXTESTER is acceptable, and the bottleneck for its effectiveness is the quality of third-party tools used in the framework for instrumentation.

b) Strategies evaluation: Figure 3 summarises the experiment results for 10 runs per each input events number (1000, 2000, 3000, 5000 and 10000) chosen for Monkey strategy. It shows that the average code coverage (in basic blocks) slightly grows with more events generated (data from our full dataset).

Figure 4 demonstrates how some simple strategies explored perform in terms of basic blocks coverage on our full dataset.

[7]The `K-9 Mail` application was in both A3E and Dynodroid datasets

[8]The `org.liberty.android.fantastischmemo` app was in both the Dynodroid and SwiftHand datasets

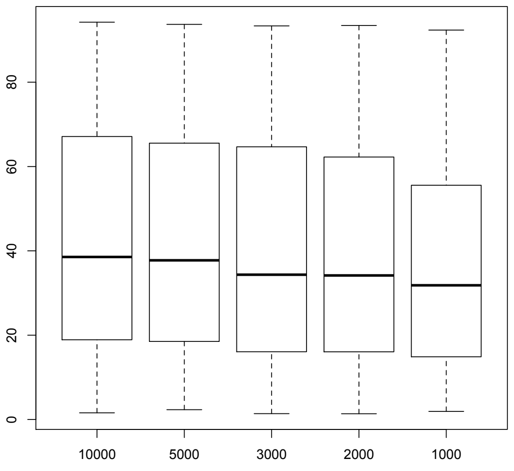

Fig. 3. Average coverage with Monkey (% of basic blocks)

TABLE II. SUMMARY ON NUMBERS OF EXPERIMENTS THAT DID NOT GENERATE COVERAGE REPORTS (IN A SINGLE RUN; NOT IN ALL 10 RUNS).

	1000	**2000**	**3000**	**5000**	**10000**
A3E	21	20	23	21	25
Dynodroid	38	51	52	64	68
SwiftHand	13	15	18	20	19
Missing total	72	86	93	105	112
Experiments total	600	600	600	600	600

Summary statistics are given in Table III. In this table we also report the total number or apps that failed during all runs of Monkey (listed as # of N/A) for a set of events. In Section VII we give more details about these crashes and overview the app bugs that we encountered.

Interestingly, from Figure 4 and Table III we can see that cumulative coverage on 1000 Monkey events can be better than cumulative coverage attained by Monkey on 10000 events. This can be explained by a bigger chance that app crashes due to some internal bugs in a 10000 events run than a 1000 events run. The chance of unsuccessful runs on 10000 events is significantly higher then on 1000 (see Table II for details).

Thus, we can conclude that *more events with less runs can be less optimal strategy for automated testing, if the goal is to increase the code coverage* (e.g., in the security testing to detect malware, when more behaviours need to be uncovered), taking into account the time budget available and the increased chances for apps to crash in longer executions. If a security analyst is interested in increased code coverage, she should try more runs on shorter input sequences. *If the goal is to find bugs, then bigger input event sequences can be justified.*

In the same time, we have found examples of applications in our data set that have shown much higher coverage results in some rare runs (1 in 30 approx.). These are applications: `com.gluegadget.hndroid`, `com.chmod0.manpages`, `com.teleca.jamendo`, `com.kvance.nectroid`, `org.smerty.zooborns`, `com.templaro.opsiz.aka`; all from the Dynodroid dataset. In additional experiments (100 runs with 1000 events) we have confirmed that it is possible for Monkey to attain higher coverage for these applications only occasionally. We discuss these cases in details in Section VIII.

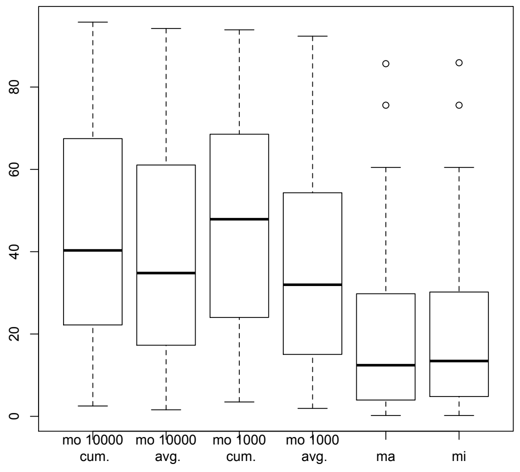

Fig. 4. Comparison of coverage for different strategies that we explored: Monkey with 1000 and 10000 events cumulative and average, and main activity and main intent triggering (% of basic block covered)

TABLE III. SUMMARY STATISTICS FOR THE CODE COVERAGE FOR THE INVESTIGATED STRATEGIES (IN % OF BASIC BLOCKS COVERED). FOR THE *mo* STRATEGY WE REPORT CUMULATIVE, AVERAGE AND THE BEST RUN COVERAGE IN 10 EXPERIMENTS.

Statistics	**Monkey strategy**						**Others**	
	10000 ev.			1000 ev.				
	cum.	avg.	best run	cum.	avg.	best run	*ma*	*mi*
min	3.49	1.59	3.00	3.49	1.93	3.10	0.20	0.20
1st Qu.	23.45	17.36	20.77	22.17	15.14	18.16	4.11	4.85
Median	40.31	34.81	40.22	45.53	31.90	40.07	12.42	13.44
Mean	44.77	39.25	42.92	45.84	35.58	41.68	19.73	20.57
3rd Qu.	67.55	62.45	64.12	67.95	53.64	62.25	29.55	30.17
max	95.77	94.23	95.77	93.91	92.36	93.57	85.68	85.91
# N/A	6			5			0	

Regarding the simpler strategies *ma* and *mi*, we can see from the boxplots and summary statistics that they are not very effective in achieving good code coverage. The median coverage for *ma* is only 12.42% of basic blocks, and for *mi* is 13.44%. These strategies can be practically useful only as a very short preliminary step to actual testing, or in cases when the app crashes under any random input sequence. As evident from Table III, these strategies never resulted in consistent app crashes, unlike Monkey. These two strategies have almost the same code coverage, with *mi* resulting in only slightly better numbers. However, as we will show in the following section, *mi* is effective in detecting bugs in apps.

From Table III we can see, somewhat evidently, that *running functional testing with random input only once can result in underestimation of actual coverage that can be achieved by the testing system.* More effective strategy to achieve better code coverage is to run several tests and compute cumulative value. This seems self-evident, however, in most of the research papers on Android apps testing only single-run coverage (or its average on several runs) is reported, not the cumulative coverage.

c) Comparison with A3E coverage: The A3E system [12] is the only one for which the reported code coverage metrics (% of methods covered) is also measured by BBOXTESTER. In [12] the authors report mean coverage of 36.46% achieved on the full dataset of 31 apps. BBOXTESTER instrumented 9 of them, and produced coverage statistics for

7 apps (2 apps generated exceptions). For these 7 apps, mean cumulative methods coverage produced by BBoxTester on 1000 events is 20.28%. This is in line with our expectations – the exploration by Monkey is not as sophisticated as A3E's one. On our full dataset the method coverage mean is 46.68% (cumulative on 1000 runs by Monkey). However, the apps from the Dynodroid and SwiftHand datasets are smaller and less complicated. Therefore, we can expect that A3E coverage on these apps will also be higher.

d) Comparison with Dynodroid's Monkey: As a sanity check we compare the results of Monkey's coverage reported for the Dynodroid evaluation in [1] with our Monkey's coverage in a similar setting. In [1] the authors ran Monkey 3 times with 10000 events and chose the best run. We therefore, chose the best run in the first 3 for 10000 events. Notice, that our code coverage measure is basic blocks on binaries, while Dynodroid reports coverage as source code LOCs. Therefore, we did not expect equal results, but it was interesting to see if the results would be similar (strongly correlated) or not.

The correlation coefficient between the data points sets (the Dynodroid Monkey's values versus our Monkey's values) is 0.45 (values $\geq$ 0.5 are traditionally interpreted as strong correlation). Pearson's product-moment correlation test shows 95% confidence interval for correlation coefficient [0.16, 0.67], with p-value for null-hypothesis 0.0033. We can interpret these results as "somewhat" correlated.

Our experiments were conducted in the same setting as the Dynodroid evaluation, on the same app set, and we used the same Monkey tool. However, from the data we have we cannot conclude that the results are quite similar. Thus, given two automated testing framework reporting code coverage in different metrics, it is difficult to establish which one actually provides better coverage. This conclusion reinforces the need for a toolkit like BBoxTester that can compare results of different testing frameworks using the same coverage metrics.

VII. Application Bugs Discovered in Testing with BBoxTester

BBoxTester is able to report uncaught exceptions that were thrown during testing. These errors cause termination of the app execution, therefore stopping the coverage measurement as well. Table IV summarizes the bugs discovered by BBoxTester during the evaluation experiments.

In total, we have detected 15 bugs in 13 different applications of our dataset. The bugs were mainly discovered in activities; however, we also found one bug in an application component, one bug in a service, and one bug was located in an unspecified component (in Table IV its type is null). The analysis shows that both the *mo* and *mi* strategies are useful in bugs discovery. The *ma* strategy is less efficient in that regard. Using this strategy, we managed to discover only 2 bugs in the `bbc.mobile.news.ww` app. However, the same bugs were also detected using the *mo* and *mi* strategies. Thus, combining these two strategies can bring additional benefits in terms of bugs discovery. Notice that the bugs we report were not listed as detected by Dynodroid [1].

Almost all exceptions detected are general Java exceptions. At the same time, we detected a custom exception raised in the package `com.irahul.worldclock`. As we had access to the source code of this application, we were able to conclude that even though the developers throw an exception in the code, it is not intercepted anywhere in the app. This may be an evidence that the intended behaviour of the app is to crash in case of the incorrect input.

Additionally, not all of the experiments generated the runtime coverage files. In some cases, there was no information for either "on error" or "on stop" events, what made it impossible to generate code coverage reports for such experiments. It should be mentioned that these cases were not really rare. For the *mo* strategy, 110 out of 450 experiments for A3E dataset, 275 out of 2250 for the Dynodroid dataset, and 85 out of 300 for the SwiftHand dataset did not contain runtime data. We performed manual analysis of these cases to understand the cause of these faults. We found out that some exceptions were not intercepted by the watchdog component that processes exceptions occurred in an AUT. This forces the application process to be stopped by the Android OS. The process is killed with all threads including the one that generates runtime coverage reports and, thus, the task of generation of a file with the coverage data is not accomplished.

We repeated some test runs with the same parameters as in the original test runs in a controlled experiment, observing their execution with the *logcat* utility. In this analysis, we have discovered two scenarios. For 3 apps we were able to reproduce the errors that caused the app crash, and all the experiments for the application at hand (50 experiments) finished without the runtime coverage data when we used the *mo* strategy. At the same time, for the *ma* and *mi* strategies applied to the same apps, the runtime coverage data were successfully generated. This happened for two app packages (`com.rechild.advancedtaskkiller` and `cz.romario.opensudoku`) in the A3E dataset, and one in the Dynodroid dataset (`com.everysoft.autoanswer`). In the A3E dataset these two applications resulted in 100 experiments conducted without the runtime coverage information produced. We observed the errors appearing in the log file, but they were not discovered in the error file produced by our framework.

For the rest of cases, our repeated experiments finished successfully. We believe that former unsuccessful runs could be caused by some specific states of the application itself or the operating system. These exceptions are raised under certain conditions and are difficult to reproduce. However, these crashes were also not logged by our framework, thus, it is difficult to conclude with certainty what has caused these exceptions.

VIII. Limitations and Future Work

BBoxTester measures only code coverage of Dalvik code, i.e., it cannot calculate the coverage of native code. Thus, if an app heavily relies on some functionality implemented in native code, the coverage numbers obtained with BBoxTester may not be reflecting the real code coverage (including the native code coverage). This is a common limitation for all tools that mostly target the Dalvik bytecode.

BBoxTester relies on a number of external tools, namely *dx*, *Emma*, *zipalign*, *dex2jar* and *apktool*. The retargeting

TABLE IV. DISCOVERED BUGS: MO - WITH MONKEY STRATEGY; MA - WITH MAIN ACTIVITY STRATEGY; MI - WITH MAIN INTENTS STRATEGY

Package Name	Strategy	Type	Exception
A3E			
bbc.mobile.news.ww	mo / ma / mi	application	java.util.MissingResourceException
bbc.mobile.news.ww	mo / ma / mi	activity	java.lang.NullPointerException
com.devuni.flashlight	mo	activity	java.lang.NullPointerException
Dynodroid			
com.android.keepass	mi	activity	java.lang.IllegalArgumentException
com.angrydoughnuts.android.alarmclock	mo	service	java.lang.IllegalStateException
com.irahul.worldclock	mi	activity	com.irahul.worldclock.WorldClockException
org.smerty.zooborns	mi	activity	java.lang.NullPointerException
com.teleca.jamendo	mi	activity	java.lang.NullPointerException
com.teleca.jamendo	mo	activity	java.lang.NullPointerException
jp.sblo.pandora.aGrep	mo	activity	java.util.regex.PatternSyntaxException
org.wikipedia	mi	null	java.lang.ClassNotFoundException
SwiftHand			
cri.sanity	mo	activity	java.lang.NullPointerException
jp.gr.java_conf.hatalab.mnv	mi	activity	java.lang.NullPointerException
net.fercanet.LNM	mi	activity	java.lang.NullPointerException
org.jessies.dalvikexplorer	mo	activity	java.util.MissingResourceException

process from the Dalvik bytecode to the Java bytecode and then its compilation back to Dalvik may introduce additional errors, thus, preventing applications from being successfully instrumented. As we discussed in Section VI, there was a number of apps that were not instrumented by BBOXTESTER. The majority of errors were discovered in *dx* during compilation of the Java instrumented bytecode, more precisely, the instrumentation of the Java bytecode performed by *Emma* later prevented the *dx* tool from successful compilation of the Java instrumented bytecode. Similar results were also reported by other Android researchers. For instance, Bartel et al. [35] managed to compile successfully 33 out of 39 apps with instrumented Java bytecode. Some developers deliberately design their applications to cause crashes in decompilation tools such as *dex2jar* [36]. This also influences on the correct operation of our framework.

One potential solution is to use components that produce less errors. For instance, the *Dare* [31] retargeter can be used instead of *dex2jar*, and the *JaCoCo* [30] code coverage tool can be used instead of *Emma*. However, we believe it will be more practical to develop a new framework that will work directly on the Dalvik bytecode level. This will prevent the system from using retargeting tools and compilers, thus, eliminating two components that caused the majority of errors. This is even more important in the light of the upcoming change of the *dx* compiler to *Jack & Jill* [37], the future versions of which will not necessary rely on `.class` files. We leave this as a future work.

During our experiments with the Monkey tool we observed that for some apps in random experiments the code coverage values were significantly higher than in the rest of experiments. This is explained by presence of some code in the apps that is triggered rarely and its invocation is not connected with the UI events generated by the Monkey tool. This triggering may be caused by the following events: a) generated by the OS itself (for instance, quite often the subscribed broadcast `CONNECTIVITY_CHANGE` is fired when connectivity details changed); b) subscribed by the application itself (e.g., alarms or sensor listeners); c) invoked by other applications (for instance, through an implicit intent). While the first and partially the second cases are tried to be addressed by the tools like Dynodroid [1], the third is still in its early developments, only partially addressed in [38]. Our tool can help in measuring the effectiveness of the future developments in these areas.

BBOXTESTER can be further improved in several aspects. During the experiments we observed inability of our framework to capture all exceptions thrown in AUTs. To capture exceptions we rely on the functionality provided by the Android framework. Thus, the inability to intercept all exceptions is not the limitation of our tool but the one of the Android OS. In any case, currently we are investigating this issue to further improve BBOXTESTER. Another direction of our future research is mapping of covered code parts to decompiled source code. This future extension of our system is of particular our interest because it will allow us to observe dead code and perform its analysis.

IX. RELATED WORK

Given that BBOXTESTER is a system to measure code coverage obtained in testing that we used to evaluate some simple automated testing strategies, our work falls in the category known as systems for automated mobile app testing.

A. White-box testing approaches

We classify testing techniques that require the app source code as white-box. For instance, these are systems that rely on symbolic and concolic execution, e.g., ACTEVE [39], Java Path Finder for Android [40], Collider [22].

Other white-box testing approaches examples are systems that generate test cases and dynamically analyse applications based on their source code. Some notable works in this area include EvoDroid [2], Dynodroid [1], and the system proposed by Avancini and Ceccato for testing app communications [38].

For white-box testing there are already tools for code coverage monitoring available, such as *Emma* and JaCoCo. Our tool will be handy in situations when the source code of apps is not available, thus when white-box testing approaches will not be applicable.

B. Grey-box and black-box testing techniques

In the absence of the source code automated testing of third-party apps can become more tricky. We classify systems that do not require source code as black-box approaches (though, technically most of those are grey-box as they manipulate with binaries and extract some structural information). Such systems as SwiftHand [33], A3E [12], Troyd [41], Caiipa [42], DroidFuzzer [43], PUMA [10], AppDoctor [20], Orbit [44], and AndroidRipper [21] provide working solutions for automated testing of mobile apps without source code.

For BBoxTester PUMA [10] is the most relevant testing framework, as it allows to create automatic GUI exercisers for mobile apps. It incorporates a tool similar to Monkey, for which the user can define UI exploration strategies; these for instance can be strategies that we have evaluated in this paper. PUMA effectively decouples automatic Monkey-based app exploration from the app properties analysis. Our functionality for driving automated testing is less developed, as we only support very simple strategies at the moment. In the same time, PUMA currently lacks code coverage reporting. Thus, PUMA and BBoxTester can be used in synergy.

Many frameworks, such as CopperDroid [16], PuppetDroid [14], DroidRacer [45], AppsPlayground [46], DECAF [47], Brahmastra [11], VanarSena [19], and SmartDroid [48] utilise application testing as a way to detect malicious or vulnerable applications. All these systems innovate in the automated testing without source code. BBoxTester has also identified some bugs in applications under testing, yet bug detection is not the primary goal of the system. BBoxTester is in fact complimentary to the various strategies to automatically exercise app code and intelligently generate input events proposed in these works. Our tool can be used as a part of any of these systems to provide more precise code coverage results, and it can be used to compare effectiveness of different approaches on the same code coverage metrics.

Some approaches (Dynodroid, SwiftHand, AppDoctor, VanarSena, Collider) use the Monkey tool as a benchmark for effectiveness, or even gather Monkey's results as inputs for their work (the system in [49]). With the data from our strategies evaluation, we can see that these approaches can be further improved. For example, Collider compares with Monkey's results obtained from 6000 events, as the authors report stabilisation of code covered. This may be true for smaller applications, like the ones used in the SwiftHand dataset. However, we can see from our data that it is not necessarily the case for larger applications, where many runs of Monkey, even with a smaller input event sequences, might be needed to exhibit low-probability input events.

C. Dynodroid, A3E and SwiftHand

The Dynodroid system [1] for automatic input generation is able to generate a set of relevant input events (UI events, such as button clicks, and system events, such as broadcast receiver events and system service events) and system events that can be consumed by a running app. The system was evaluated for two aspects: source code coverage with respect to other input generation approaches (Dynodroid vs. Monkey vs. manual execution) and discovery of bugs in apps. Source code coverage (as LOCs) in Dynodroid is obtained from *Emma*. The evaluation results (on the apps included into our dataset) show that Dynodroid and Monkey achieve comparable coverage but Monkey requires more input events. Our findings show that using cumulative coverage on shorter runs (1000 events only) can lead to noticeable improvements in coverage achieved by Monkey.

A3E is a system for systematic exploration of Android applications built on top of Troyd [12]. The A3E framework automatically explores the app code on device by injecting user-like GUI actions and generating callbacks to invoke activities. The framework follows two strategies: depth-first exploration that corresponds to systematic user-like exploration of an app, and targeted exploration that imitates calls of app activities from other apps and system services. To support targeted exploration A3E first builds a static activity transition graph with the help of SCanDroid [50]. Then A3E tries to exercise all available GUI elements and to cover to all exportable activities in the graph. The evaluation on 25 real apps (that we used as a part of our dataset) has reported the method coverage at 29% with the targeted exploration, and 36% with the depth-first exploration.

SwiftHand is a system for automatically generate test input sequences for Android apps [33]. One of the main intuitions behind the approach is that it tries to avoid app restarts as the most time-expensive operation. SwiftHand learns an approximate GUI model of the app (that is based on the application state and the user inputs enabled in it) while executing the test, and explores yet unvisited paths in this model. It is also able to reply already covered paths. To enable this exploration SwiftHand requires binary instrumentation, just like BBoxTester. SwitfHand was evaluated on 10 open source applications; the branch code coverage ranges from 19.6% to 72.2% for the evaluated apps.

We used the evaluation results reported for these systems to benchmark BBoxTester, however, as we reported in Sec. VI, it is not possible to conclude which of two testing frameworks achieves better code coverage unless the same coverage metrics is used in them. The A3E framework that actually uses a comparable code coverage metrics achieves better coverage on 7 apps. As the next step for our work, we plan to integrate SwiftHand and A3E with BBoxTester to enrich them with more precise code coverage metrics, and to be able to compare these tools on a bigger dataset.

X. Conclusion

Existing approaches to measure code coverage, which is an important testing effectiveness metrics, in testing frameworks for Android are either too coarse-grained (e.g, percent of triggered activities or percent of invoked methods) or fine-grained but work only if app source code is available. We presented BBoxTester – a framework that measures code coverage metrics in testing Android applications. Our system complements automated application testing platforms allowing them to measure fine-grained code coverage metrics and detect bugs in applications under a test.

We evaluated our framework on a set of real applications using 3 simple testing strategies, and we gained some insights into selection of automated test generation strategy given a limited time budget. For example, our experiments show that cumulative coverage is a must for all studies using Monkey as a benchmark. Our results also demonstrate that even these 3 simple strategies can be useful in bug finding process. Indeed, during our experiments with our framework we managed to locate bugs in 13 different apps out of 60 used for evaluation.

Acknowledgements: We would like to thank the authors of the A3E system [12] for sharing their testing apps with us. This work was partially supported by EIT Digital through the MobileShield project.

REFERENCES

[1] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input Generation System for Android Apps," in *Proceedings of ESEC/FSE'2013*.

[2] R. Mahmood, N. Mirzaei, and S. Malek, "EvoDroid: Segmented Evolutionary Testing of Android Apps," in *Proceedings of FSE'2014*, 2014.

[3] EMMA: A free Java code coverage tool. http://emma.sourceforge.net/.

[4] Google Play Launch Checklist. http://developer.android.com/distribute/tools/launch-checklist.html.

[5] App Store Publishing Guidelines. https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/SubmittingYourApp/SubmittingYourApp.html.

[6] J. Voas, S. Quirolgico, C. Michael, and K. Scarfone, "Technical Considerations for Vetting 3rd Party Mobile Applications (Draft)," NIST, NIST Special Publication 800-163 (Draft), 2014.

[7] Robotium. https://code.google.com/p/robotium/.

[8] Espresso at android-test-kit. https://code.google.com/p/android-test-kit/wiki/Espresso.

[9] UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html.

[10] S. Hao, B. Liu, S. Nath, W. Halfond, and R. Govindan, "PUMA: Programmable UI-Automation for Large Scale Dynamic Analysis of Mobile Apps," in *Proceedings of Mobisys'2014*, 2014.

[11] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall, "Brahmastra: Driving Apps to Test the Security of Third-Party Components," in *Proceedings of Usenix Security'2014*, 2014.

[12] T. Azim and I. Neamtiu, "Targeted and Depth-first Exploration for Systematic Testing of Android Apps," in *Proceedings of OOPSLA'2013*, 2013, pp. 641–660.

[13] L. Inozemtseva and R. Holmes, "Coverage is not Strongly Correlated with Test Suite Effectiveness," in *Proceedings of ICSE'2014*, 2014, pp. 435–444.

[14] A. Gianazza, F. Maggi, A. Fattori, L. Cavallaro, and S. Zanero, "PuppetDroid: A User-Centric UI Exerciser for Automatic Dynamic Analysis of Similar Android Applications," arXiv:1402.4826, 2014.

[15] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci, "StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications," in *Proceedings of CODASPY '15*, 2015.

[16] A. Reina, A. Fattori, and L. Cavallaro, "A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors," in *Proceedings of EuroSys'2013*, 2013.

[17] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen, "I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications," in *IEEE Mobile Security Technologies (MoST)*, 2012.

[18] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou, "A Whitebox Approach for Automated Security Testing of Android Applications on the Cloud," in *Proceedings of AST'2012*.

[19] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan, "Automatic and Scalable Fault Detection for Mobile Applications," in *Proceedings of MobiSys'2014*, 2014.

[20] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor," in *Proceedings of EuroSys'2014*.

[21] D. Amalfitano, N. Amatucci, A. Fasolino, U. Gentile, G. Mele, R. Nardone, V. Vittorini, and S. Marrone, "Improving Code Coverage in Android Apps Testing by Exploiting Patterns and Automated Test Case Generation," in *Proceedings of WISE'2014*, 2014.

[22] C. Jensen, M. Prasad, and A. Moller, "Automated Testing with Targeted Event Sequence Generation," in *Proceedings of ISSTA'2013*, 2013.

[23] Y. Zhauniarovich, O. Gadyatskaya, and B. Crispo, "Demo: Enabling trusted stores for Android," in *Proc. of CCS*. ACM, 2013, pp. 1345–1348.

[24] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. L. Spina, and E. Moser, "FSquaDRA: Fast detection of repackaged applications," in *Proc. of DBSec*, ser. LNCS, vol. 8566. Springer, 2014, pp. 130–145.

[25] Tools Help: Platform tools. https://developer.android.com/tools/help/index.html.

[26] ApkTool: A tool for reverse engineering Android apk files. https://code.google.com/p/android-apktool/.

[27] Dex2Jar: Tools to work with android .dex and java .class files. https://code.google.com/p/dex2jar/.

[28] Zipalign. https://developer.android.com/tools/help/zipalign.html.

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

[30] JaCoCo Java Code Coverage Library. http://www.eclemma.org/jacoco/.

[31] D. Octeau, S. Jha, and P. McDaniel, "Retargeting Android Applications to Java Bytecode," in *Proceedings of FSE'2012*, 2012, pp. 6:1–6:11.

[32] Android API Reference: Instrumentation. http://developer.android.com/reference/android/app/Instrumentation.html.

[33] W. Choi, G. NEcula, and K. Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," in *Proceedings of OOPSLA'2013*, 2013.

[34] Intents and Intent Filters. http://developer.android.com/guide/components/intents-filters.html.

[35] A. Bartel, J. Klein, M. Monperrus, K. Allix, and Y. Le Traon, "Improving Privacy on Android Smartphones through In-vivo Bytecode Instrumentation," arXiv preprint arXiv:1208.4536, 2012.

[36] A. Apvrille. (2013, December) Sophisticated DEX obfuscation or Proguard configuration issue? http://bit.ly/1vosShb.

[37] E. Lafortune. (2014, November) The upcoming Jack & Jill compilers in Android. https://www.saikoa.com/blog/the_upcoming_jack_and_jill_compilers_in_android.

[38] A. Avancini and M. Ceccato, "Security Testing of the Communication among Android Applications," in *Proceedings of AST'2013*, 2013.

[39] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated Concolic Testing of Smartphone Apps," in *Proceedings of FSE'2012*, 2012.

[40] N. Mirzaei, S. Malek, C. Pasareanu, N. Esfahani, and R. Mahmood, "Testing Android Apps Through Symbolic Execution," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, 2012.

[41] J. Jeon and J. S. Foster, "Troyd: Integration Testing for Android," University of Maryland, Tech. Rep. CS-TR-5013, 2012.

[42] C.-J. Liang, N. Lane, N. Brouwers, L. Zhang, B. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao, "Caiipa: Automated Large-scale Mobil App Testing through Contextual Fuzzing," in *Proceedings of MobiCom'2014*, 2014, pp. 519–530.

[43] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag," in *Proceedings of MoMM'2013*.

[44] W. Yang, M. Prasad, and T. Xie, "A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications," in *Proceedings of FASE'2013*, 2013.

[45] P. Maiya, A. Kanade, and R. Majumdar, "Race Detection for Android Applications," in *Proceedings of PLDI'2014*, 2014.

[46] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: Automatic Security Analysis of Smartphone Applications," in *Proceedings of CODASPY'2013*, 2013.

[47] B. Liu, S. Nath, R. Govindan, and J. Liu, "DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps," in *Proceedings of Usenix NSDI'2014*, 2014.

[48] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "SmartDroid: an Automatic System for Revealing UI-based Trigger Conditions in Android Applications," in *Proceedings of SPSM'2012*.

[49] C. Hu and I. Neamtiu, "Automating GUI Testing for Android Applications," in *Proceedings of AST'2011*, 2011.

[50] A. Fuchs, A. Chaudhuri, and J. S. Foster, "SCanDroid: Automated Security Certification of Android," University of Maryland, Tech. Rep. CS-TR-4991, 2009.