COSMO: Code Coverage Made Easier for Android

Andrea Romdhana*,¶, Mariano Ceccato†, Gabriel Claudiu Georgiu‡, Alessio Merlo*, Paolo Tonella§
*DIBRIS, University of Genoa, Italy
¶Security & Trust Unit, FBK-ICT, Trento, Italy
†University of Verona, Italy
‡Consorzio Nazionale per l'Informatica - CINI, Rome, Italy
§Software Institute, Università della Svizzera Italiana, Lugano, Switzerland

***Abstract*—The degree of code coverage reached by a test suite is an important indicator of the thoroughness of testing. Most coverage tools for Android apps work at the bytecode level and provide no information to developers about which source code lines have not yet been exercised by any test case. In this paper, we present COSMO, the first fully automated Android app instrumenter that operates at the source code level in a completely transparent way, making it fully compatible with existing system level testing technologies and Android test generators. The experiments that we have conducted on a large benchmark of Android apps show that COSMO can successfully instrument most apps without altering their execution traces, introducing a small, acceptable runtime overhead.**

***Index Terms*—code coverage, instrumentation, android testing**

I. Introduction

The complexity of mobile applications (hereafter, apps) keeps growing, as apps provide always more advanced services (e.g., in the banking or health care domains) to the users. In order to ensure adequate testing of the software, monitoring the level of code coverage is regarded as a best, highly recommended practice. It is used by developers and testers to understand the degree of exploration of the software [1], to create test cases [2] and to compare or assess the adequacy of test suites [3]. Code coverage has become an essential adequacy criterion also in the mobile world [4] [5] [6], but measuring code coverage for mobile apps is far from trivial.

In the absence of the source code, code coverage can be measured by instrumenting the *smali* bytecode of the app, as done for instance by ACVTool [7]. However, the assessment of the proportion of covered smali code is less significant for a developer than the evaluation of the covered *source code* lines. In fact, high coverage of smali code does not always mean a correspondingly high coverage of the source code. Moreover, understanding how to cover the smali code portions not yet covered by the current test suite is way more difficult for developers than reasoning directly on the uncovered *source* code. On the other hand, existing solutions to instrument directly the Java source code of an Android app are severely limited. Generic tools for Java, such as JaCoCo [8], cannot be applied to Android easily, while Android-specific solutions [9], [10] work only for unit level test cases written in Junit and cease to work at the system level, when performing GUI oriented end-to-end testing of an Android app – a scenario that is prevalent in industrial Android testing and that is supported by existing Android test generators [6], [11], [12].

To overcome these limitations, we propose COSMO (COde coverage via jacoco inStruMentatiOn), a tool that enables automatic instrumentation of Android apps. COSMO can automatically modify the source code of the Android app. In addition, COSMO can operate as a black-box tool, instrumenting an app starting from the compiled app. COSMO is the only Android coverage tool that can instrument the source code of recent (i.e., Gradle $>$ 2.13) apps (based on Java or Kotlin), transparently from the user and supporting fine grained analysis of the uncovered code portions directly on the source code.

The paper provides the following contributions:

- An approach to instrument apps at the level of classes, methods and lines of code, starting from source code or from the APK.
- An implementation of the instrumentation approach in COSMO, which can be easily integrated into any testing or dynamic analysis framework. In fact, COSMO is based on JaCoCo's code coverage measurement functionalities.
- An empirical assessment that shows the reliability and flexibility of our approach:
  - We successfully instrumented 703 apps. We checked that after instrumentation apps can be successfully executed and that their behavior remains unchanged.
  - We measured the time overhead introduced by COSMO, in the context of automated or manual testing. Our assessment of original and instrumented app executions using Espresso test cases shows that there is no noticeable run-time overhead for real apps (the mean execution time increase is 5.9%).
  - COSMO is based on JaCoCo, a popular code coverage library for Java. Thus, code coverage statistics reported by COSMO are available for visualization, reporting and analysis in the widely used JaCoCo format.
- We released COSMO as an open source tool to support the Android testing and analysis community. The source code is available at https://github.com/H2SO4T/COSMO.

II. State of the art

Coverage of Android apps can be achieved by means of two different types of instrumentation: black-box vs white-box

instrumentation. The former operates on the app bytecode; the latter on its source code.

Black-box approach. There are several black-box tools for measuring code coverage. However, they are not capable of measuring fine-grained source code level coverage. At the time of writing, ACVTool [7] represents the state of the art for black-box Android coverage tools and outperforms pre-existing tools, such as ELLA [13], InsDal [14], and CovDroid [15]. ACVTool instruments Dalvik bytecode in its smali representation by inserting probes to measure code coverage at the levels of classes, methods, and instructions. However, ACVTool calculates code coverage for the bytecode, but it cannot map such coverage back to source code lines, making it less interpretable for developers. Also, ACVTool is not compatible with *multi-dex*, which is the standard app model since Android version 5[1].

White-box approach. For generic Java projects, coverage of the source code can be determined using well-established techniques and tools, such as JaCoCo (or its predecessor Emma [16]). However, instrumentation via JaCoCo is not trivial and the available online documentation and resources are often outdated, incomplete, and not applicable (e.g., modifications that depends on Gradle or Android versions) to modern Android apps. Tools for white-box code coverage measurement are included and maintained by Google in the Android SDK [17]. Such tools can be used via Android Studio [18], the standard IDE for developing apps on the Android ecosystem. Android Studio tools include support for coverage libraries such as JaCoCo. However, Android Studio tools can compute coverage only for test cases written within the IDE. In addition, there exist several plugins compatible with Android Studio that help developers collect code coverage starting from test cases written for JUnit [9], [10]. The main limitation of these plugins is the impossibility to collect code coverage when testing tools different from JUnit are used, such as automated testing via Monkey[2], Sapienz [6], Stoat [11] or TimeMachine [12]. COSMO instead generates instrumented apks that are agnostic with respect to the testing approach being used.

III. Architecture

COSMO enables developers to measure the degree to which an Android app code is executed during testing and uses JaCoCo to generate coverage reports. COSMO is divided into two main submodules: 1) *COSMO from source*, 2) *COSMO from apk*.

COSMO from source automatically inserts all the dependencies needed to instrument the source code for coverage monitoring. It modifies Java, Gradle and Manifest source files. The code injected by COSMO is then inserted into the apk if the app is compiled with debugging enabled. In this way the developer does not need to waste time to remove portions of unneeded code when preparing the production release.

[1]https://developer.android.com/studio/build/multidex
[2]https://developer.android.com/studio/test/monkey

COSMO from apk takes a compiled app as input. The app is converted to Java bytecode and instrumented using JaCoCo, and it is eventually aligned and signed. Figure 1 illustrates the workflow of COSMO that consists in 3 phases: an *offline phase*, a *run-time phase* and the *report generation phase*. In the next section, we provide further details about each phase.

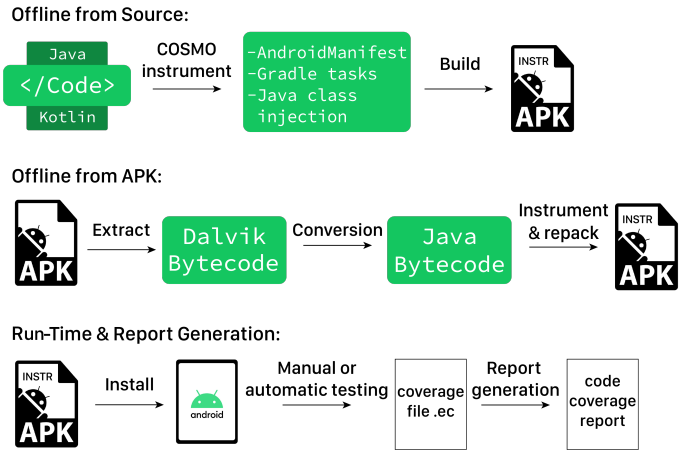

Fig. 1. COSMO workflow.

IV. Source code instrumentation

COSMO from source allows developers to automatically instrument Gradle-based [19] apps and to generate code coverage reports available for visualization, reporting and analysis in the widely used JaCoCo format. In Android, Gradle is the standard build toolkit [20]. It automates the building process of app resources and source code, packaging them into apk files that the developer can test, deploy, sign, and distribute.

A. Offline Phase

In order to instrument the source code of an app, COSMO executes the following steps:

- **Manifest Modification**: COSMO searches for the `AndroidManifest.xml` of the app. Once found, COSMO declares a broadcast receiver named `EndCoverageBroadcast` as an app component. `EndCoverageBroadcast` enables the app to receive intents named `intent.END_COVERAGE`.
- **Generation of Instrumentation Class**: COSMO creates a file called `EndCoverageBroadcast.java` in the project directory. This file implements the `EndCoverageBroadcast` broadcast receiver, previously added to the `AndroidManifest.xml`. When triggered by a developer, the intent marked as `intent.END_COVERAGE` collects the code coverage of the app under test. The code coverage file has extension `.ec`.
- **Gradle Tasks Modification**: COSMO adds the missing dependencies to the `build.gradle` task. Moreover, COSMO inserts a new task,

`jacoco-instrumenter-coverage.gradle`. This task generates the final coverage report starting from the `*.ec` files generated during the testing phase of the app.

- **JaCoCo Properties**: as final step COSMO generates a file named `jacoco-agent.properties` in the `resources` folder. It specifies the properties used by JaCoCo.

B. Run-Time Phase

Once the offline phase is completed, the developer can build and install the app with debugging enabled. The app can now be explored manually, or by using automated testing tools. Once the exploration is completed, the developer can collect the code coverage by sending the intent `intent.END_COVERAGE` to the app. The intent can be sent manually through the command line at the end of the test phase or automatically. Automating the sending of the intent during the test phase could be the best way to prevent the loss of code coverage data in the event of an app crash. The intent will trigger the generation of a `coverage.ec` file. This file is located in the `/sdcard/Android/data/<APP_PACKAGE>/files/` directory.

C. Report Generation Phase

To generate the coverage report, developers can run the task `jacocoInstrumenterReport` (i.e., `./gradlew jacocoInstrumenterReport`). Coverage reports can be in HTML, CSV or XML format and are located in `build/reports/jacoco/jacocoInstrumenterReport/` under the project directory.

D. Usage

COSMO from source is implemented in Python. Launching `cli.py <APP_PATH>` the developer can select the app folder to instrument, and the tool will complete the procedure. Then, the developer can build, install, launch and test the app (manually or automatically). In order to collect coverage data, the developer must send the following intent: `adb shell am broadcast -p <APP_PACKAGE> -a intent.END_COVERAGE`. Then, the developer can collect the generated `.ec` files from the Android device via `adb -P 5037 -s <udid> pull <PATH_TO_FILE>/coverage.ec <DESTINATION_PATH>`. At last, by running the command `./gradlew jacocoInstrumenterReport` the developer will generate the final code coverage report.

V. Bytecode instrumentation

COSMO from apk is meant to automatically instrument apps, starting from their compiled form. To the best of our knowledge, there is only one mature, freely available apk instrumentation tool: ACVTool [7]. Differently from ACVTool, which requires developers to grant the instrumented app additional permissions (e.g., file-system write permission, to write the trace file there), no invasive additional permission is required by COSMO, thus minimizing the intrusiveness of our monitoring facility.

Enabling coverage monitoring consists of three phases: offline phase run-time phase and report generation.

A. Offline Instrumentation

Offline instrumentation is automatically applied as follows:

- First of all, the Dalvik bytecode is extracted from the apk and converted to Java bytecode using `dex2jar`[3];
- Java bytecode is instrumented using JaCoCo offline instrumentation. This component adds an array of probes and opcodes in the Java bytecode, to flip probes when code is executed and branches are taken at runtime;
- A patched version of the JaCoCo agent is added. The original version of the agent reads its configuration from a configuration (property) file or from system properties. The patch consists of making the agent able to read the configuration also from Android properties, using the class `android.os.SystemProperties`;
- Java bytecode is converted back to Dalvik and then added to the original apk, that is eventually aligned and signed. This is achived using tools and utilities distributed with the Android SDK.

B. Run-Time Phase & Report Generation

Once the apk is instrumented, it can be installed on a device or emulator and run to trace code coverage. However, before running the app, an Android property should be set, to specify where to save the trace file. This can be achieved with the command line utility `setprop`, by running the command `setprop jacoco.destfile <FILE>`. The trace file should be located in a portion of the file system where the app is granted write permission. For instance the SD, in case the app is granted the corresponding permission. Alternatively, the app private file system can be used, i.e. `/data/data/<APP_PACKAGE>/coverage.ec`.

When manual exploration is performed or test cases are executed, the trace file can be downloaded from the Android file system, for instance using the command `adb pull`.

The trace file is in standard JaCoCo format, and can be converted to a detailed HTML report using the JaCoCo offline tool `jacococli`.

It is important to notice that, in case the app is compiled with source code information, source files and line numbers are preserved in the apk. These references are exploited and included in the coverage report. Differently, ACVTool does not offer this feature, because it works at the Smali code level, missing completely the references to the source code.

VI. Evaluation

To assess the practical applicability of our tool and the associated costs, we have performed an extensive empirical evaluation of COSMO with respect to the following research questions:

[3]https://github.com/pxb1988/dex2jar/wiki

- **RQ1, Instrumentation success rate:** Does the code instrumented by *COSMO* compile?
- **RQ2, App health:** Does the code instrumented by *COSMO* execute and produce the same output as the original code?
- **RQ3, Runtime overhead:** What is the impact in terms of time and memory overhead of the instrumentation made by COSMO?

To answer these research questions we have selected and built the 1350 most starred F-Droid apps available on GitHub. We successfully compiled 830 apps, and 792/830 worked properly. To answer **RQ1**, we evaluated *COSMO from source* on this set of 792 apps. *COSMO from apk* was evaluated on a subset of 475 of the 792 previously selected apps. The subset contains only apps written in Java since the JaCoCo offline instrumenter is not compatible with the Kotlin programming language, used in the excluded apps. Moreover, we removed the `multi-dex` apps as `dex2jar` currently does not support it [21]. To answer **RQ2**, we calculate the percentage of the instrumented apps that can run on an emulator. We wrote a script to install, launch, and interact automatically with the instrumented apps on an Android emulator. The script produces a sequence of actions (hereafter, *execution trace*) on the original apps and replicates the same actions on the instrumented apps a second time. During the script execution, we collect the current activity, the md5 of the activity layout, and the widgets (i.e., their hierarchy on the activity layout and their identifiers) with which we interact. At last, the script compares the execution traces to check whether there are discrepancies. If the instrumented app can run for 10 seconds without crashing and generates the same execution trace of the original app, we define it as healthy. To answer **RQ3**, we used test cases written by hand; Android Studio Profiler [22] to measure the memory usage and the execution time, averaging the values over five runs per app in order to account for random fluctuations of these measurements. For test case execution, we chose the Espresso testing framework because it introduces the smallest footprint during android app execution [23].

The analysis was carried out on a single machine with a quad-core 3.40 GHz processor and 16GB of RAM.

A. Experimental Results

Table I summarizes the main statistics related to instrumentation and app healthiness. Of the 830 apps compiled from the source code, 792 executed without crashing (column "Working"). Hence, these apps are included in the instrumentation test. Considering *COSMO from source*, results show that after the instrumentation, the compiled apps are 703/792 and that all of them work (i.e., execute with no crash after the instrumentation). We investigated the build failures, discovering that they are mainly related to three causes. 63 apps do not support `multi-dex`, and due to instrumentation, the unique `dex` file generated goes beyond the 64K methods allowed, and the build fails. A solution would be to enable `multi-dex` in the Gradle script for these apps. Another build error is related to 15 apps based on old Java versions (e.g., Java 1.6) that are not compatible with our Java files. At last, 11 apps download corrupted resources during the build phase.

Considering *COSMO from apk*, results show that after the instrumentation, the compiled apps are 413/475. 50 apps fail the instrumentation due to the Android tool `dx` [24] that can not convert the `jar` file into `dex` bytecode (`SimException`). Moreover, 12 apps already have some sort of instrumentation (e.g., instrumentation via Android Studio tools), and due to this COSMO raises the exception `IllegalStateException`. During the test phase 296/413 apps worked. All non working apps raise `RuntimeException` and crash due to certain app resources or classes that can not be found. Upon investigation of the issues, we suspect that they could be due to faults in `dex2jar`. We are working to properly identify and fix the bugs.

Then, we compared the execution traces of the working apps. To do so, we used the same script applied during the validation of *COSMO from source*. The script produces an execution trace on the original apps, collecting the current activity and the activity layout's md5. Moreover, it collects the identifiers of the widgets the apps interact with and their hierarchy on the activity layout, and it replicates the execution trace on the instrumented apps a second time. The execution trace comparison made by the script on all apps found no discrepancies. COSMO does not introduce disturbing components in the workflow of the apps.

Table I shows also the instrumentation time. On average *COSMO from source* takes less than one second, while *COSMO from apk* needs around 12 seconds.

> RQ1 & RQ2: *COSMO from source* (resp. *from apk*) successfully instrumented 88.7% (resp. 86.9%) of the apps, and 100% (resp. 71.6%) of them execute without errors. Both versions of COSMO generate apks that produce the same execution trace as the original apks if executed on the same test scenario.

At last, we evaluate the impact of the COSMO instrumentation on the app performance. To measure the runtime overhead, we randomly chose ten apps, and we profiled their activity when running test cases using Espresso. To accommodate fluctuations we repeated each test case five times. As shown in Table II, the runtime overhead causes, on average, a 5.9% execution time increase when running the Espresso test cases. Moreover, memory usage is, on average, 3.7% higher than with the original apps.

> RQ3: The instrumentation performed by COSMO introduces a small execution time and memory usage increase (on average, resp. 5.9% and 3.7%).

B. Qualitative analysis

Let us consider an example of an Android app and the use of COSMO in support to coverage testing. An interesting exam-

Meth.	Compilable	Working	Same Trace	Instr. Time
Original	830/1350	792/830	-	-
Cosmo from source	703/792	703/703	703/703	1 sec.
Cosmo from apk	413/475	296/413	296/296	12 sec.

TABLE I
AN OVERVIEW OF THE RESULTS OBTAINED BY COSMO

App	Time (sec)	Time'	TDelta (perc.)	Mem (Mb)	Mem'	MDelta (perc.)
Antennapod	44.6	45	1 %	166	176	6%
WifiAnalyzer	34	37	9%	139	139	0%
AmazeFileManager	20	21.3	7%	150	157	5%
Image-To-PDF	6.5	6.9	6%	80	84	5%
busybox	7.9	8.7	9%	34	34	0%
gpstest	9.6	10.6	10%	73	78	6%
vanilla	8.7	9.3	6%	58	59	1%
YalpStore	8.4	8.7	3%	55	58	5%
KISS	8.2	8.6	5%	76	78	3%
AppOpsX	7.3	7.5	3%	51	54	6%

TABLE II
COSMO RUNTIME OVERHEAD: INCREASED EXECUTION TIME AND MEMORY USAGE

ple is Antennapod [25], a modern Android app characterized by many activities and services. We may want to write an Espresso test that interacts with the preference activity and then collects code coverage. Figure 2 shows the portion of the coverage report generated for the `PreferenceActivity` class when executing a basic interaction scenario. The test consists of a naive exploration of the settings, in which we click only on a few of the voices accessible, without exploring related sub-preferences. The report contains the methods available in the activity and their coverage. Figure 3 shows detailed coverage information for some of the methods of class `PreferenceActivity`. We can see that our first test did not exercise some portions of the app's settings (lines 62 to 75). This code block is linked to package `fragment.preferences`. Figure 4 shows that the code coverage of this package is poor. Hence, we can infer that a second test should exercise preferences related to network, storage, or import/export in order to increase coverage.

Hence, we wrote a second test that interacts with the previously missed preferences and almost all accessible voices in `fragment.preferences`. Figure 5 shows the updated report. Even without studying the report in detail, we can immediately appreciate the increase in code coverage (method `getPreferencesScreen()` increased its coverage from 17% to 65%). The overall code coverage of `PreferenceActivity` increased from 56% to 79% and `fragment.preferences` jumped from 3% to 33%. Figure 6 and Figure 7 show that the new test case was able to test a large portion of the previously uncovered code.

VII. ADOPTION SCENARIO

COSMO is potentially useful in several adoption scenarios, which involve both developers and researchers:

- COSMO can be a useful tool to assist developers during the testing phase of their apps. COSMO automatically instruments Android apps. By examining the code coverage report, developers can write more thorough and more effective tests. In this way, developers can test their apps in more depth.
- Developers can also use COSMO to compare alternative automated test generation tools on their own app (for instance Sapienz [6], Stoat [11] or Timemachine [12]), to find the one that suits it better by achieving higher coverage. Actuakky, we integrated very easily and seamlessly our tool with the Monkey and Sapienz Android test generators.
- Researchers can use COSMO when developing novel test case generation approaches, to guide automated test generation and app exploration toward execution scenarios that increase coverage, or just to measure the achieved coverage, on case study apps that are available either as

PreferenceActivity

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
getPreferenceScreen(int)		17%		6%	8	9	14	18	0	1
onOptionsItemSelected(MenuItem)		0%		0%	3	3	6	6	1	1
onSearchResultClicked(SearchPreferenceResult)		0%		n/a	1	1	3	3	1	1
getTitleOfPage(int)		90%		88%	1	9	1	10	0	1
onCreate(Bundle)		100%		75%	1	3	0	14	0	1
openScreen(int)		100%		n/a	0	1	0	4	0	1
onCreateOptionsMenu(Menu)		100%		n/a	0	1	0	2	0	1
PreferenceActivity()		100%		n/a	0	1	0	1	0	1
Total	84 of 191	56%	21 of 33	36%	14	28	24	58	2	8

Fig. 2. `PreferenceActivity`: summary code coverage achieved by the first test case

```
31.  */
32.  public class PreferenceActivity extends AppCompatActivity implements SearchPreferenceResultListener {
33.      private static final String FRAGMENT_TAG = "tag_preferences";
34.
35.      @Override
36.      protected void onCreate(Bundle savedInstanceState) {
37.          setTheme(UserPreferences.getTheme());
38.          super.onCreate(savedInstanceState);
39.
40.          ActionBar ab = getSupportActionBar();
41.          if (ab != null) {
42.              ab.setDisplayHomeAsUpEnabled(true);
43.          }
44.
45.          FrameLayout root = new FrameLayout(this);
46.          root.setId(R.id.content);
47.          root.setLayoutParams(new FrameLayout.LayoutParams(ViewGroup.LayoutParams.MATCH_PARENT,
48.                  ViewGroup.LayoutParams.MATCH_PARENT));
49.          setContentView(root);
50.
51.          if (getSupportFragmentManager().findFragmentByTag(FRAGMENT_TAG) == null) {
52.              getSupportFragmentManager().beginTransaction()
53.                      .replace(R.id.content, new MainPreferencesFragment(), FRAGMENT_TAG)
54.                      .commit();
55.          }
56.      }
57.
58.      private PreferenceFragmentCompat getPreferenceScreen(int screen) {
59.          PreferenceFragmentCompat prefFragment = null;
60.
61.          if (screen == R.xml.preferences_user_interface) {
62.              prefFragment = new UserInterfacePreferencesFragment();
63.          } else if (screen == R.xml.preferences_network) {
64.              prefFragment = new NetworkPreferencesFragment();
65.          } else if (screen == R.xml.preferences_storage) {
66.              prefFragment = new StoragePreferencesFragment();
67.          } else if (screen == R.xml.preferences_import_export) {
68.              prefFragment = new ImportExportPreferencesFragment();
69.          } else if (screen == R.xml.preferences_autodownload) {
70.              prefFragment = new AutoDownloadPreferencesFragment();
71.          } else if (screen == R.xml.preferences_gpodder) {
72.              prefFragment = new GpodderPreferencesFragment();
73.          } else if (screen == R.xml.preferences_playback) {
74.              prefFragment = new PlaybackPreferencesFragment();
75.          } else if (screen == R.xml.preferences_notifications) {
76.              prefFragment = new NotificationPreferencesFragment();
77.          }
78.          return prefFragment;
79.      }
```

Fig. 3. `PreferenceActivity`: detailed code coverage achieved by the first test case

de.danoeh.antennapod.fragment.preferences

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ImportExportPreferencesFragment		0%		0%	48	48	174	174	33	33	1	1
NetworkPreferencesFragment		0%		0%	27	27	109	109	18	18	1	1
AutoDownloadPreferencesFragment		0%		0%	41	41	106	106	15	15	1	1
PlaybackStatisticsFragment		0%		0%	30	30	88	88	21	21	1	1
PlaybackPreferencesFragment		0%		0%	19	19	62	62	12	12	1	1
GpodderPreferencesFragment		0%		0%	20	20	66	66	14	14	1	1
UserInterfacePreferencesFragment		40%		25%	18	28	54	100	9	18	0	1
DownloadStatisticsFragment		0%		0%	12	12	34	34	11	11	1	1
StoragePreferencesFragment		0%		0%	13	13	35	35	10	10	1	1
StatisticsFragment		0%		0%	10	10	26	26	6	6	1	1
MainPreferencesFragment		65%		50%	12	18	23	72	10	16	0	1
NotificationPreferencesFragment		0%		n/a	4	4	10	10	4	4	1	1
StatisticsFragment.StatisticsPagerAdapter		0%		0%	4	4	6	6	3	3	1	1
PlaybackStatisticsFragment.new ConfirmationDialog() {...}		0%		n/a	2	2	4	4	2	2	1	1
GpodderPreferencesFragment.new AuthenticationDialog() {...}		0%		n/a	2	2	3	3	2	2	1	1
Total	3.193 of 3.536	9%	177 of 184	3%	262	278	800	895	170	185	13	15

Fig. 4. `fragment.preferences`: summary code coverage achieved by the first test case

PreferenceActivity

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
getPreferenceScreen(int)		65%		56%	4	9	6	18	0	1
onSearchResultClicked(SearchPreferenceResult)		0%		n/a	1	1	3	3	1	1
onOptionsItemSelected(MenuItem)		72%		50%	2	3	2	6	0	1
getTitleOfPage(int)		90%		88%	1	9	1	10	0	1
onCreate(Bundle)		100%		50%	2	3	0	14	0	1
openScreen(int)		100%		n/a	0	1	0	4	0	1
onCreateOptionsMenu(Menu)		100%		n/a	0	1	0	2	0	1
PreferenceActivity()		100%		n/a	0	1	0	1	0	1
Total	39 of 191	79%	12 of 33	63%	10	28	12	58	1	8

Fig. 5. `PreferenceActivity`: summary code coverage achieved by the second test case

```
31. public class PreferenceActivity extends AppCompatActivity implements SearchPreferenceResultListener {
32.     private static final String FRAGMENT_TAG = "tag_preferences";
33.
34.     @Override
35.     protected void onCreate(Bundle savedInstanceState) {
36.         setTheme(UserPreferences.getTheme());
37.         super.onCreate(savedInstanceState);
38.
39.         ActionBar ab = getSupportActionBar();
40.         if (ab != null) {
41.             ab.setDisplayHomeAsUpEnabled(true);
42.         }
43.
44.         FrameLayout root = new FrameLayout(this);
45.         root.setId(R.id.content);
46.         root.setLayoutParams(new FrameLayout.LayoutParams(ViewGroup.LayoutParams.MATCH_PARENT,
47.                 ViewGroup.LayoutParams.MATCH_PARENT));
48.         setContentView(root);
49.
50.         if (getSupportFragmentManager().findFragmentByTag(FRAGMENT_TAG) == null) {
51.             getSupportFragmentManager().beginTransaction()
52.                     .replace(R.id.content, new MainPreferencesFragment(), FRAGMENT_TAG)
53.                     .commit();
54.         }
55.     }
56.
57.     private PreferenceFragmentCompat getPreferenceScreen(int screen) {
58.         PreferenceFragmentCompat prefFragment = null;
59.
60.         if (screen == R.xml.preferences_user_interface) {
61.             prefFragment = new UserInterfacePreferencesFragment();
62.         } else if (screen == R.xml.preferences_network) {
63.             prefFragment = new NetworkPreferencesFragment();
64.         } else if (screen == R.xml.preferences_storage) {
65.             prefFragment = new StoragePreferencesFragment();
66.         } else if (screen == R.xml.preferences_import_export) {
67.             prefFragment = new ImportExportPreferencesFragment();
68.         } else if (screen == R.xml.preferences_autodownload) {
69.             prefFragment = new AutoDownloadPreferencesFragment();
70.         } else if (screen == R.xml.preferences_gpodder) {
71.             prefFragment = new GpodderPreferencesFragment();
72.         } else if (screen == R.xml.preferences_playback) {
73.             prefFragment = new PlaybackPreferencesFragment();
74.         } else if (screen == R.xml.preferences_notifications) {
75.             prefFragment = new NotificationPreferencesFragment();
76.         }
77.         return prefFragment;
78.     }
79.
```

Fig. 6. `PreferenceActivity`: detailed code coverage achieved by the second test case

de.danoeh.antennapod.fragment.preferences

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ImportExportPreferencesFragment		27%		13%	35	48	123	174	20	33	0	1
PlaybackStatisticsFragment		0%		0%	30	30	88	88	21	21	1	1
PlaybackPreferencesFragment		0%		0%	19	19	62	62	12	12	1	1
GpodderPreferencesFragment		0%		0%	20	20	66	66	14	14	1	1
UserInterfacePreferencesFragment		37%		25%	20	28	61	100	12	18	0	1
AutoDownloadPreferencesFragment		46%		34%	25	41	56	106	5	15	0	1
NetworkPreferencesFragment		73%		41%	11	27	34	109	4	18	0	1
DownloadStatisticsFragment		0%		0%	12	12	34	34	11	11	1	1
StatisticsFragment		0%		0%	10	10	26	26	6	6	1	1
MainPreferencesFragment		71%		50%	10	18	19	72	8	16	0	1
NotificationPreferencesFragment		0%		n/a	4	4	10	10	4	4	1	1
StatisticsFragment.StatisticsPagerAdapter		0%		0%	4	4	6	6	3	3	1	1
PlaybackStatisticsFragment.new ConfirmationDialog() {...}		0%		n/a	2	2	4	4	2	2	1	1
GpodderPreferencesFragment.new AuthenticationDialog() {...}		0%		n/a	2	2	3	3	2	2	1	1
StoragePreferencesFragment		93%		50%	4	13	4	35	1	10	0	1
Total	2.355 of 3.536	33%	145 of 184	21%	208	278	596	895	125	185	9	15

Fig. 7. `fragment.preferences`: summary code coverage achieved by the second test case

source code or as apk.

VIII. Conclusion

In this paper, we presented COSMO – a tool for measuring Android code coverage at the source code level. When instrumenting from source code (resp. apk), COSMO was able to instrument 88.7% (resp. 86.9%) and execute 100% (resp. 71.6%) of the considered benchmark apps, showing that COSMO is practical and reliable.

COSMO can help both researchers who are building testing, program analysis, and security assessment tools for Android, as well as developers, who need a reliable instrumenter capable of producing easy-to-understand coverage information. Among the possible future developments, we plan to use COSMO to build a testing tool that exploits code coverage information to explore the app thoroughly.

References

[1] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[2] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *The Computer Journal*, vol. 52, no. 5, pp. 589–597, 2009.

[3] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, "Guidelines for coverage-based comparisons of non-adequate test suites," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 4, pp. 1–33, 2015.

[4] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?(e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 429–440.

[5] C.-Y. Huang, C.-H. Chiu, C.-H. Lin, and H.-W. Tzeng, "Code coverage measurement for android dynamic analysis tools," in *2015 IEEE International Conference on Mobile Services*. IEEE, 2015, pp. 209–216.

[6] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 94–105.

[7] A. Pilgun, O. Gadyatskaya, S. Dashevskyi, Y. Zhauniarovich, and A. Kushniarou, "An effective android code coverage tool," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2189–2191.

[8] Jacoco code coverage library. Accessed 10-16-2020. [Online]. Available: https://www.eclemma.org/jacoco/

[9] Gradle android junit jacoco plugin. Accessed 10-16-2020. [Online]. Available: https://github.com/vanniktech/gradle-android-junit-jacoco-plugin

[10] Jacoco android gradle plugin. Accessed 10-16-2020. [Online]. Available: https://github.com/arturdm/jacoco-android-gradle-plugin

[11] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pp. 245–256.

[12] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 481–492.

[13] Ella. Accessed 10-16-2020. [Online]. Available: https://github.com/saswatanand/ella

[14] J. Liu, T. Wu, X. Deng, J. Yan, and J. Zhang, "Insdal: A safe and extensible instrumentation tool on dalvik byte-code for android applications," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 502–506.

[15] C.-C. Yeh and S.-K. Huang, "Covdroid: A black-box testing coverage system for android," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 3. IEEE, 2015, pp. 447–452.

[16] Emma: a free java code coverage tool. Accessed 11-21-2020. [Online]. Available: http://emma.sourceforge.net

[17] Android test. Accessed 10-16-2020. [Online]. Available: https://developer.android.com/studio/test

[18] Android studio. Accessed 10-16-2020. [Online]. Available: https://developer.android.com/studio

[19] Gradle. Accessed 10-16-2020. [Online]. Available: https://gradle.org

[20] Android gradle build. Accessed 10-16-2020. [Online]. Available: https://developer.android.com/studio/build

[21] dex2jar. Accessed 10-16-2020. [Online]. Available: https://github.com/pxb1988/dex2jar

[22] Android studio profiler. Accessed 10-16-2020. [Online]. Available: https://developer.android.com/studio/profile/android-profiler

[23] Espresso. [Online]. Available: https://developer.android.com/training/testing/espresso

[24] build-tools. Accessed 13-21-2020. [Online]. Available: https://developer.android.com/studio/releases/build-tools

[25] Antennapod. Accessed 11-20-2020. [Online]. Available: https://antennapod.org