IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 18, 2017, accepted February 8, 2018, date of publication February 21, 2018, date of current version March 19, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2808340

A Comparison of Android Reverse Engineering Tools via Program Behaviors Validation Based on Intermediate Languages Transformation

YAUHEN LEANIDAVICH ARNATOVICH[1], LIPO WANG[1], NGOC MINH NGO[2], AND CHARLIE SOH[1]

[1]School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore 639798
[2]Global Outreach & Extended Education, Arizona State University, Ho Chi Minh City, Vietnam

Corresponding author: Lipo Wang (elpwang@ntu.edu.sg)

ABSTRACT In Android, performing a program analysis directly on an executable source is usually inconvenient. Therefore, a reverse engineering technique has been adapted to enable a user to perform a program analysis on a textual form of the executable source which is represented by an intermediate language (IL). For Android, Smali, Jasmin, and Jimple ILs have been introduced to represent applications executable *Dalvik* bytecode in a human-readable form. To use these ILs, we downloaded three of the most popular Android reversing tools, including *Apktool*, *dex2jar*, and *Soot*, which perform transformation of the executable source into Smali, Jasmin, and Jimple ILs, respectively. However, the main concern here is that inaccurate transformation of the executable source may severely degrade the program analysis performance, and obscure the results. To the best of our knowledge, it is still unknown which tool most accurately performs a transformation of the executable source so that the re-assembled Android applications can be executed, and their original behaviors remain intact. Therefore, in this paper, we conduct an experiment to identify the tool which most accurately performs the transformation. We designed a statistical event-based comparative scheme, and conducted a comprehensive empirical study on a set of 1,300 Android applications. Using the designed scheme, we compare *Apktool*, *dex2jar*, and *Soot* via random-event-based and statistical tests to determine the tool which allows the re-assembled applications to be executed, and evaluate how closely they preserve their original behaviors. Our experimental results show that *Apktool*, using Smali IL, perform the most accurate transformation of the executable source since the applications, which are assembled from Smali, exhibit their behaviours closest to the original ones.

INDEX TERMS Event-based testing, intermediate languages, program behaviours, reliability, reverse engineering, statistics.

I. INTRODUCTION

Nowadays, Android is no doubt the most popular smartphone operating system. Android dominated the market with 85.0% share in 2017Q1.[1] In fact, due to the high market share, many Android applications (usually called ''apps'') are being developed and published without performing an adequate program analysis. In fact, the program analysis helps an app developer to automate the process of analysing the behaviours of the program regarding the correctness, robustness, and safety before it reaches the end-user. Unfortunately, after publishing the app, the source code is usually not publicly available, and performing a program analysis directly on an executable source may be inconvenient which may be hindered by various code optimization, and obfuscation techniques. Therefore, a transformation of the Android apps executables (i.e., *Dalvik* bytecode) into intermediate languages (ILs) is desired [1]. The IL is the lowest-level human-readable programming language which is automatically generated by reversing tools using transformation of the executable source into its corresponding textual representation [2].

In this paper, we use three of the most popular Android apps reversing tools including *Apktool*,[2] *dex2jar*,[3] and *Soot*,[4] which perform transformation of the executable source into

[1]http://idc.com/smartphone-os-market-share

[2]http://github.io/apktool
[3]http://github.com/dex2jar
[4]http://github/soot

Smali (*Dalvik Assembler*), Jasmin (*Java Assembler*), and Jimple (*Java Simplified*) ILs, respectively. Other than these, another tools have also been developed for reversing Android apps executables. For example, *Dexpler*[5] [3] which is built on the top of *Dedexer*,[6] and *Androguard*[7] which is built to support transformation from *Dalvik* bytecode into *Java*-like code for existing frameworks *WALA*[8] [4] and *Soot* [5]. However, an inaccurate transformation of the executable source may decrease a program analysis performance, and obscure the results if the analysis was performed using the IL with inadequate quality and reliability [6]–[8]. In fact, a ''good'' IL must be capable of representing the source code without loss of information [9], and independent of any particular source or target language.[9]

The recent research works suggest that Smali [10]–[13], Jasmin,[10] [14]–[16], and Jimple [3], [17]–[20] ILs represent a source code so that they all can be used to carry out various program analyses. For example, in static program analysis, the ILs can be used to perform mutation and metamorphic testing, clones, ''bad smells'', data and privacy leaks detection, ''points-to'' analysis, and others. In dynamic program analysis, the ILs can be used for code instrumentation to inject the user-defined probes into the code. The instrumentation technique is usually used for an automatic collection of the interesting to the test data, e.g., collecting symbolic constraints while performing concolic testing (dynamic symbolic execution).

In fact, a program analysis is fundamentally based on *Control*- and/or *Data*-flow graphs. However, extracting the flow graphs from an executable source is not a trivial task. For example, the executable source lacks many useful properties of high-level languages, and thus introduces various challenges to the program analysis tools. Furthermore, various code optimization and obfuscation techniques can further obscure the program flows. For example, an inaccurate transformation of the executable source may not adequately reflect the original program behaviours in the respective program structures, so the corresponding *Control*- and/or *Data*-flow graphs could be imprecise.

In this paper, we are inspired to conduct a study to identify the tool which performs the most accurate transformation of the Android apps executables into its respective IL. We downloaded 1,300 Android apps from *Google Play* (by 50 top free apps from 26 different categories), and processed them as follows. With the aid of *Apktool*, *dex2jar*, and *Soot* tools, we disassemble the downloaded apps to obtain Smali, Jasmin, and Jimple ILs, respectively, and assemble the apps from the obtained ILs without making any modifications to the ILs content. We install the downloaded (original), and re-assembled apps on Android emulators, and perform the functional GUI testing using an event-based Monkey test. We analyze the obtained Monkey test results (the number of successfully injected events) by performing *one-sample* and *two-sample* statistical *t*-tests. As a result, we identify the tool which performs the most accurate transformation. Our experimental results show that the apps which are assembled from Smali behave closest to the original ones. Therefore, we suggest *Apktool* as a tool which performs the most accurate transformation of the Android apps executables.

In this work, we made the following contributions:

1) We develop the statistical event-based comparative scheme using the parametric statistical tests, and functional GUI testing to evaluate an accuracy of the Android ILs transformation via an automated program behaviours validation.
2) We apply our scheme on 1,300 closed-source Android apps, and compare the program behaviours of the Android apps which are assembled from Smali, Jasmin, and Jimple ILs with original ones.
3) We identify a reversing tool which most accurately performs a transformation of the Android apps executables into its respective IL.

The rest of the paper is organized by the following sections. Section II introduces the *Dalvik* virtual machine, *Dalvik* executable, and shows differences between Java virtual machine and Java executable (bytecode). Section III provides details of our experiment design. Section IV presents our experimental results. Section V discusses related work. Section VI concludes our paper.

II. Dalvik VIRTUAL MACHINE AND Dalvik EXECUTABLE

In this section, we describe the special features of *Dalvik* VM and stress its differences from standard *Java* VM. We also show that *Java* bytecode structure is different from *Dalvik* bytecode (executable), even most Android apps are being developed in *Java*.

A. Dalvik VIRTUAL MACHINE

The *Dalvik* VM is not a *Java* VM. The *Dalvik* is a *register-based* VM in which Android apps are executed. Since both apps and system services on the Android OS are, generally, implemented in *Java*, the *Dalvik* VM has been developed so that an Android device can run multiple VMs efficiently. Every Android app runs in its own process, with its own instance of the *Dalvik* VM and it is referred to as sandboxing apps. When an Android device is started, a single virtual machine process, called *Zygote* is created, which preloads and pre-initializes core library classes (*.libc*). Once the *Zygote* has been initialized, it will reside and listen for socket requests coming from the runtime process, which indicates that it should generate new VM instances based on the existing *Zygote* VM instance. Thus, by spawning new VM processes from the *Zygote*, the boot time of *Dalvik* VM is highly minimized. All other *Java* programs or services are originated from this process, and run as their own process or threads in their own address space.

[5] http://abartel.net/dexpler/
[6] http://sourceforge.net/dedexer
[7] http://code.google.com/androguard
[8] http://sourceforge.net/wala
[9] http://cs.lmu.edu/ir
[10] http://sourceforge.net/jasmin

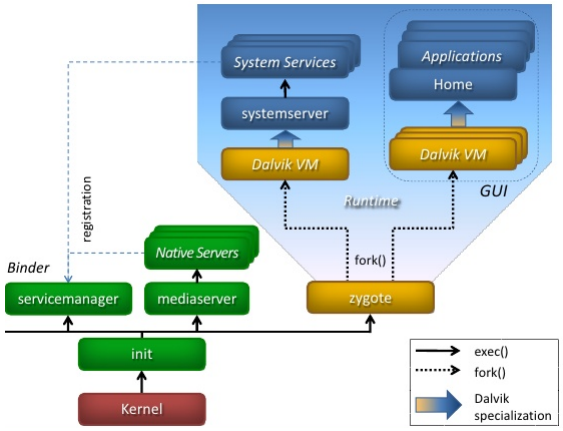

FIGURE 1. Dalvik virtual machine: creation and usage.

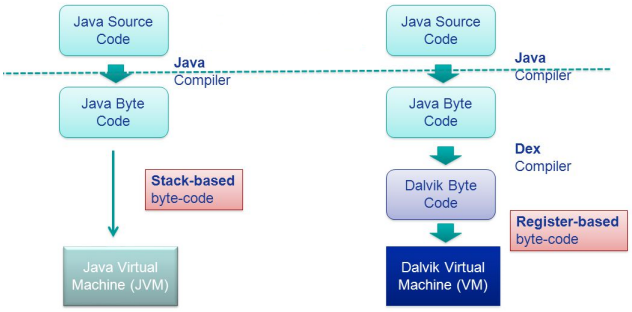

FIGURE 2. Difference in process of transformation Java source code (.java) into Java bytecode (.class), and Dalvik executable (.dex).

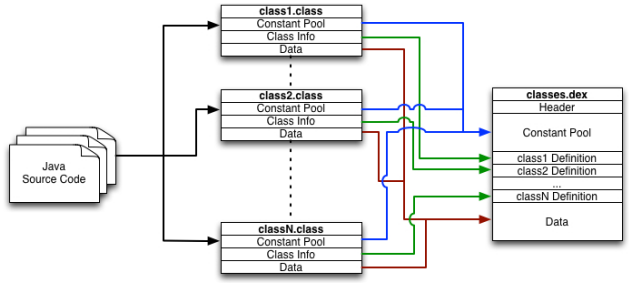

FIGURE 3. Difference in internal structure of Java bytecode (.class), and Dalvik executable (.dex).

The *Dalvik* VM relies on the Linux kernel for underlying functionality such as threading and low-level memory management. The core library classes (*.libc*) that are shared across the *Dalvik* VM instances are, commonly, only read by apps. When the (*.libc*) classes are needed, the memory from the shared *Zygote* VM process is simply copied to the forked child process of the app's *Dalvik* VM. Such behaviour allows it to maximize the total amount of shared memory while still restricting apps from overlapping with each other and providing security across apps, and sandboxing individual processes. In Fig. 1, we show the overall process how the *Dalvik* VM is created and used in the Android system.

Traditionally, the *Java* VM (*stack-based* architecture) design is the instance of the VM which has an entire copy of the core library class files and any associated heap, thus memory is not shared across multiple instances of *Java* VM. Since the *Dalvik* VM is *register-based* architecture, it requires, on average, 47% less executed VM instructions than the *stack-based*. However, the *register-based* code is, on average, 25% larger than the corresponding *stack-based* code, and this increases the cost of fetching VM instructions per unit of time. Notwithstanding, the overall performance of the *register-based* VM takes, on average, 32.3% less time than *stack-based* VM to execute standard benchmarks [21]–[23].

With the above-mentioned assumptions, *Dalvik* VM was developed to meet the following criteria [21], [22]:

1) Limited CPU speed.
2) Limited RAM amount.
3) No swap space.
4) Battery powered.
5) Diverse set of Android devices.
6) Sandboxed application runtime.

B. Dalvik EXECUTABLE

The *Dalvik* executable (*.dex*) file format is designed to meet the requirements of systems that are constrained in terms of memory (RAM) and processor (CPU) speed. Since Android apps are, usually, written in *Java*, *Java* classes (*.java*) are compiled by a *Java* compiler (*javac* tool) that have been further transformed into the single file *classes.dex* by a standard *dex* compiler (*dx* tool which is, by default, included into Android SDK). After applying the *dx* tool, the *classes.dex* file has had only distant relationship with the *Java* bytecode (*.class*). In Fig. 2, we show the difference in the process of translation original *Java* source code into the *Java* bytecode and *Dalvik* executable.

The *Dalvik* executable file design is primarily driven by sharing of data between running processes. The main difference between the *Java* bytecode (*.class*) and *Dalvik* bytecode (*.dex*) is that all the classes of the Android app (*.class*) are packed into one single file *classes.dex*. Note that this is not simply packing, all the classes in the same *Dalvik* bytecode (*.dex*) file share the same *field*, *method*, *tables* and other resources. In particular, *Constant Pool* stores all literal constant values used within the class such as *string constants* used in code as well as *field*, *variable*, *class*, *interface*, and *method* names [21], [23]. In Fig. 3, we show the difference in the internal structure of the *Java* bytecode (*.class* files) and *Dalvik* executable (*.dex*) file.

III. EXPERIMENT DESIGN

In our experiment, we use four Android emulators with preinstalled Android 6.0 (Marshmallow, API Level 23), *Apktool*, *dex2jar*, and *Soot* Android apps reversing tools, Monkey tool which is an automated GUI exerciser,[11] and, by default, included in Android SDK, *one-sample* and *two-sample t*-test

[11] http://developer.android.com/monkey

TABLE 1. Android emulator device specifications.

Parameter	Value
Device	Galaxy Nexus
Target	Android 6.0 (API Level 23)
CPU/ABI	Intel Atom (x86_64)
Hardware Keyboard	Present
RAM	2GB
VM heap	128MB
Front/Back cameras	Emulated
Internal storage	500MB
SD card	200MB
Internal storage content:	
audio/video files	.mp3, .mp4, .avi
pictures	.jpeg
office documents	.doc, .xls, .ppt, .pdf

statistics from MINITAB.[12] The experiment was conducted on a 4-core CPU @ 3.10GHz with 32GB RAM machine running Windows 10 (64-bit).

A. EXPERIMENTAL ENVIRONMENT

To communicate with Android emulators, we use Android Debug Bridge (*adb*).[13] The *adb* is a versatile Command Line tool from Android SDK, which permits communication with a running Android instance [24]. The detailed Android emulator device specifications are shown in Table 1. To prepare the testing environment, we performed the following steps:

1) We installed the apps on the emulators with the aid of *adb*. Each emulator is for original apps, and assembled from Smali, Jasmin, and Jimple ILs.
2) For the first time, we manually ran the apps to provide the required information (if any), e.g., *login/password*, *email*, *server port*, and other normal means of access. Such manual efforts are required to mitigate a threat to the validity of our results since, by nature, Monkey is not able to automatically generate such user inputs. However, we could not provide the information for *two-step authentication* (e.g., one-time password, verification code), since it is usually required during the app runtime, and cannot be stored by the app for the future use.
3) On every emulator, we uploaded a gallery of the most popular audio/video files, pictures and office documents (see Table 1), in case if the testing apps require them. So, the testing process will not be interrupted due to the app exception "file not found", or similar.

B. TOOLS

1) Apktool: Dalvik DISASSEMBLER

To conduct experiment on Smali, we chose the tool named *Apktool*. The next two commands are used to disassemble and assemble Android package (*.apk*), respectively:

1) *apktool d* `<apk_name>`
2) *apktool b* `<directory_with_decompiled_apk>`

When an Android app is assembled, the tool automatically creates `build/` and `dist/` sub-directories in the root `directory_with_decompiled_apk/`. The `dist/` sub-directory contains the fully assembled an Android package.

2) Dex2jar: JAVA DISASSEMBLER

The *dex2jar* is another tool which provides Jasmin. The *dex2jar* consists of four basic components:

1) *dex-reader* is designed to read the *Dalvik* executable (*.dex*) file.
2) *dex-translator* is designed to convert *Dalvik* executable instructions to a dex-ir format.
3) *dex-ir* is the format used by dex-translator, and designed to represent the *Dalvik* executable instructions.
4) *dex-tools* are developed to work with *Java* bytecode (*.class*) files.

The *dex2jar* tool uses its internal algorithms to manipulate *classes.dex* file. To run disassembling/assembling process, the following code can be used:

1) *d2j-dex2jar –f –o* `<jar_name1> <apk_name>`
2) *d2j-jar2jasmin –f –o* `<directory1> <jar_name1>`
3) *d2j-jasmin2jar –f –o* `<jar_name2> <directory1>`
4) *d2j-jar2dex –f –o* `classes.dex <jar_name2>`

The first 2 instructions are used to reverse *classes.dex* file from an Android package into Jasmin. The last 2 instructions are used to re-assemble the obtained Jasmin code into *classes.dex*. For all these commands, the first parameter serves as an output, and the second one serves as an input for file/folder. As *dex2jar* assembles only the *classes.dex* file from Jasmin code, we need to directly replace it in an Android package. Note that Android package has the same structure as a standard (*.zip*) archive, thus, we can replace *classes.dex* in Android package without corrupting its internal structure. For that reason, we renamed the extension of Android package from *.apk* to *.zip*, and ran *7-Zip*[14] archiver as follows:

7z a `<zip_name> classes.dex`

Once the replacement process is finished, we renamed the file extension from *.zip* to *.apk*.

3) SOOT: JAVA OPTIMIZATION FRAMEWORK

The *Soot* is a language manipulation and optimization framework for Java. *Soot* is capable of generating four ILs for Java code: *.baf*, *.jimple*, *.shimple* and *.grimp* [19], [20]. In this work, we focus on Jimple since only transformation from *Dalvik* bytecode into Jimple is currently supported. *Soot* automatically checks an appropriate API level specified in *AndroidManifest.xml* file, and uses the corresponding Android framework (i.e., *android.jar*) for reversing and assembling. We run *Soot* with specified arguments as follows:

–src-prec, `<[jimple|apk]>`

[12] http://minitab.com/products/minitab
[13] http://developer.android.com/adb
[14] http://7-zip.org

–*process-dir,* `<apk_folder>`
–*android-jars,* `<android_api_jar_folder>`

Since *Soot* assembles only *classes.dex* file from Jimple, it requires replacing the *classes.dex* file directly in the Android package. Thus, similarly as discussed in Section III-B.2, we replaced the assembled *classes.dex* file in each Android package without corrupting its internal structure.

4) MONKEY: AUTOMATED GUI EXERCISER

In our experiment, we use Monkey tool from Android SDK to automatically generate, and inject pseudo-random user and system events into the testing Android apps. In fact, Monkey does not know anything about the software which is being tested, and thus it simply performs random actions at random positions on the app GUI. For example, it randomly generates clicks, touches, or gestures, as well as a number of system-level events which are formed using a uniform probability distribution or *Infinite Monkey Theorem* [25]. However, in practice, Monkey shows high performance results in terms of the achieved code coverage, and failures detection rate. According to Choudhary *et al.* [26] Monkey appears most effective and efficient automated GUI-testing tool comparing to other relevant ones. In particular, the authors conducted a comprehensive study using 68 Android apps which were obtained from various app categories, and compared Monkey with other state-of-the-art tools.

To further justify a choice of Monkey test, we list the following critical quality risks which Monkey test addresses as suggested by Nyman [27]:

1) Input validation.
2) Functionality of GUI.
3) Transformation completeness and reliability.

In particular, after reversing the apps, their original behaviours may be changed during the transformation process causing them to crash or hang. So, to expose any of the above-mentioned risks, we perform the functional GUI-testing of the re-assembled apps. However, it is practically impossible, within a reasonable time, to manually perform tests involving human user on a large set of 1,300 apps. Thus, we decided to perform our testing by (1) choosing Monkey as the best option for an automated GUI-testing which is suggested by Choudhary *et al.* [26] and by (2) choosing Monkey as a sufficient test to reveal the above-mentioned quality risks as suggested by Nyman [27].

We use Monkey from the Android SDK. Default Monkey generally looks for the following conditions:

1) If Monkey is restricted to run in one or more specific packages, it disallows any attempts to navigate to any other packages, and blocks them.
2) If a particular app crashes or receives any sort of unhandled exception, Monkey will stop and report the exception.
3) If a particular app does not respond, Monkey will stop and report the exception.

To automate Monkey tests, we created a batch script file (*.bat*). For each app, our batch script includes the following command:

adb shell monkey –p `<package>` *–v* `<events_count>`

In our experiment, `<package>` parameter is used to identify a single app package accessible to Monkey without any dependencies from other packages. The `<events_count>` parameter value was manually set to be 10,000 pseudo-random events (i.e., maximum number of events to be injected).

C. DATA TRANSFORMATION

In our experiment, we use parametric statistics. One challenge to use parametric analytic techniques is the requirement that the population from where samples are taken must be *normally* or *approximately normally distributed*. In fact, for a number of non-normal distributions, there is a possibility to transform the experimental data by applying a mathematical transformation. After transformation, the data may be more similar to a normal distribution. For instance, a positively skewed distribution with a long positive tail (e.g., *log-normal*) may be transformed into a near normal distribution by taking the logarithm of the data values. Logarithmic transformation is also a convenient to transform a highly skewed variable into *normal* or *approximately normal distribution* [28].

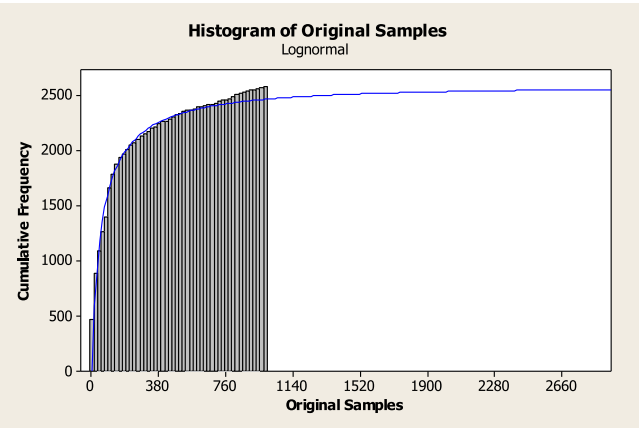

FIGURE 4. Monkey tests results *before* log-transformation: log-normal distribution of data values.

Fig. 4 shows the cumulative histogram of data distribution of the results obtained from Monkey tests for original apps, and the apps which are assembled from Smali, Jasmin, and Jimple. Analogously, Fig. 5 shows the histogram of logarithmic transformation of Monkey tests results. It is important to note that, in our experiment, *Monkey tests results* are referred to as a *number of successfully injected events* for each app tested.

From Fig. 4, we can see that the distribution of Monkey tests results is *log-normal*. From Fig. 5, we can see that the *log-transformed* data values of Monkey tests results are *approximately normally distributed*. Therefore, we are able to use parametric tests on the log-transformed data values of

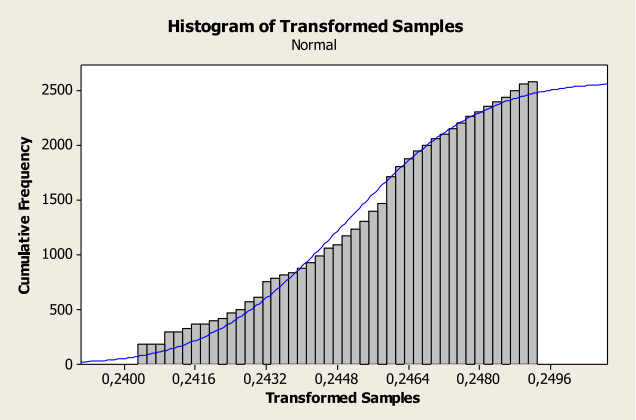

FIGURE 5. Monkey tests results *after* log-transformation: approximately normally distributed data values.

Monkey tests results. Importantly, we performed a logarithmic transformation of Monkey tests results in order to use parametric statistical tests since they have the following valuable advantages compared to non-parametric ones according to [29]–[31]:

1) Easier to use and interpret.
2) More efficient and accurate than their non-parametric counterpart.

D. EXPERIMENTAL SCHEME

For our experiment, we downloaded 1,300 Android apps from *Google Play*, and disassembled them using *Apktool*, *dex2jar*, and *Soot* to obtain Smali, Jasmin, and Jimple ILs, respectively. Afterwards, without making any modifications to the generated ILs content, we assembled Android apps using the generated ILs as a source. Next, we installed the re-assembled apps on Android emulators, ran Monkey tests with the long sequences of pseudo-random UI and system events, and performed parametric statistical tests, i.e., *one-sample* and *two-sample* *t*-tests. We use Monkey tests results (the number of successfully injected events before the app hangs or crashes) as input for our statistical tests. We apply statistics to identify if there are any differences in program behaviours between the re-assembled and original apps. *One-sample* *t*-test helps to conclude whether a particular Android app has passed Monkey test. *Two-sample* *t*-test helps to identify whether the apps, which are assembled from Smali, Jasmin, and Jimple ILs have introduced any notable differences in their behaviours comparing to the original ones, e.g., crashes or hangs which are not observed in the original apps for the same Monkey tests.

In our experiment, every Monkey test contains 5 trials (replications) for every app category. Every trial contains the same set of 50 apps from a particular category, which are assembled from Smali, Jasmin, Jimple, and original ones. Note that Monkey generates a sequence of pseudo-random events based on a manually specified *seed* value, i.e., for the same *seed* value, Monkey generates the same sequence of pseudo-random events. So, to fairly compare the program behaviours, within a trial, we run Monkey with the same *seed* value for the same set of 50 apps from a particular category. Therefore, performing 5 trials, we are able to statistically analyse, and compare the behaviours of the apps which are assembled from ILs, and the original ones in terms of the number of successfully injected events. When Monkey test is finished, we perform *one-sample* *t*-test to conclude whether Monkey test has passed successfully for a particular app, and next perform *two-sample* *t*-test to identify how close a particular re-assembled app preserves its original behaviours.

For our experiment, we designed and applied the following statistical event-based comparative scheme:

1) Disassemble Android apps using *Apktool*, *dex2jar*, and *Soot* tools to obtain Smali, Jasmin, and Jimple ILs, respectively.
2) Assemble Android apps from the unmodified source of the obtained ILs using corresponding *Apktool*, *dex2jar*, and *Soot* tools.
3) Install Android apps on four Android emulators. Each emulator for original apps, and assembled from Smali, Jasmin, and Jimple ILs. Note that, to be installed on an Android device, the Android apps must be signed with a developer private key (it may be from a self-generated key pair). Optionally, the Android packages can be aligned using the default Android *zipalign* tool for a more efficient RAM access during the app execution.
4) Run Monkey tests for original apps, and those which are assembled from Smali, Jasmin, and Jimple ILs. After every Monkey test, we wipe the Android emulators reverting them into the initial state. For every trial, we set a unique *seed* value for Monkey.
5) Perform parametric *one-sample* and *two-sample* statistical *t*-tests.

IV. EMPIRICAL EVALUATION

In our experiment, we apply statistics since we cannot solely rely on the raw results from Monkey tests since an app may crash or hang for any other reason which is not originated from Monkey test itself. Therefore, to identify a tool which performs the most accurate transformation of the Android executables, we carry out parametric *one-sample* and *two-sample* *t*-tests on the log-transformed data of Monkey tests results.

A. ONE-SAMPLE PARAMETRIC T-TEST

We perform *one-sample* *t*-test to identify whether a particular app has *passed* or *failed* Monkey test. For that purpose, we choose the following criteria to verify [32]–[34]:

Hypothesis:

$$H_0 : \mu = \mu_0$$
$$H_1 : \mu < \mu_0$$

Test Statistic:

$$t_0 = \frac{\bar{y} - \mu_0}{S/\sqrt{n}}$$

where $\bar{y}$ is the sample mean; μ_0 is the true mean value; S is the sample standard deviation; n is the sample size. The statement $H_0 : \mu = \mu_0$ is called the **null hypothesis** and $H_1 : \mu < \mu_0$ is called the **alternative hypothesis**. The alternative hypothesis specified here is called a **one-sided alternative hypothesis** because it would be true only if $\mu < \mu_0$. To test a hypothesis, we compute test statistic, and then reject or fail to reject the null hypothesis H_0 based on the computed value of the test statistic. Also, we specify the set of values for the test statistic that leads to rejection of H_0. This set of values is called the **critical region** or **rejection region** for the test. Two kinds of errors may be used when testing hypotheses. If the null hypothesis is rejected when it is true, a type I error has occurred. If the null hypothesis is not rejected when it is false, a type II error has been made. The probabilities of these two errors are given special symbols:

$$\alpha = P(\text{type I error}) = P(\text{reject } H_0 | H_0 \text{ is true})$$
$$\beta = P(\text{type II error}) = P(\text{fail to reject } H_0 | H_0 \text{ is false})$$

The general procedure in hypothesis testing is to specify a value of the probability of type I error α, often called the **significance level** of the test, and then design the test procedure so that the probability of type II error β has a suitably small value.

In our experiment, if H_0 is true, we conclude that the testing app has *passed* Monkey test successfully. If H_0 is not true, we accept alternative hypothesis H_1, i.e., the testing app has *failed* Monkey test. Note that we calculated μ_0 based on the log-transformed data values obtained from Monkey tests. In practice, the value of the mean μ_0 specified in the null hypothesis is usually determined in one of three ways. It may result from past evidence, knowledge, or experimentation. It may also be the result of some theory or model describing the situation under study, or result of contractual specifications.

Parameters $\bar{y}$ and S are evaluated according to [32]–[34] as follows:

$$\bar{y} = \frac{\sum_{i=1}^{n} y_i}{n}$$
$$S = \left(\frac{\sum_{i=1}^{n} (y_i - \bar{y})^2}{n-1} \right)^{1/2}$$

where y_i represents a sample, $i = \overline{1, n}$.

To interpret *one-sample* t-test results, we use the P-value approach with the following assumptions [32]–[34]:

1) If P-value is less than 5% level of significance (α), we would reject the null hypothesis H_0 in favour of alternative hypothesis H_1.
2) If P-value is greater than 5% level of significance, we would not reject the null hypothesis H_0 because there is no evidence that the treatment median is less than μ_0.

The P-value approach has been adopted widely in practice. It is used to state that the null hypothesis was or was not rejected at a specified α-value (**level of significance**). This is often called **fixed significance level testing**. P-value is the probability that the test statistic will take on a value that is at least as extreme as the observed value of the statistic when the null hypothesis H_0 is true. Thus, P-value conveys much information about the weight of evidence against H_0, and so we can draw a conclusion at any specified level of significance (α). More formally, we define P-value as the smallest level of significance that would lead to rejection of the null hypothesis H_0. It is common to call the test statistic (and the data) significant when the null hypothesis H_0 is rejected. Therefore, we may think of P-value as the smallest level at which the data are significant. Once P-value is known, we can determine how significant the data are without the data analyst formally imposing a pre-selected level of significance.

In Table 2, for each category, we show the number of app which failed Monkey test. To explain the values in Table 2, let us assume that we have an original *Google Chrome* app from *Communication* category. After performing Monkey test, and applying log-normal transformation to Monkey test results, we obtain data as shown in Table 3. Next, after performing *one-sample* t-test on the log-transformed data values of Monkey test results, we obtain another results as shown in Table 4. Afterwards, using the P-value approach, from Table 4 we conclude that *Google Chrome* app has failed Monkey test since the obtained P-value ($P = 0.007$) is less than the required level of significance ($\alpha = 0.05$). Therefore, we increment the value in the cell (in Table 2, column "*Original*" and row "*Communication*") by adding "1" to the current number of apps which are failed Monkey test. Analogously, we test every app from 26 categories for Smali, Jasmin, Jimple, and original ones, and calculate the number of apps failing Monkey test.

After performing *one-sample* t-test for all Smali, Jasmin, Jimple, and original apps, in Table 2, we calculate the total number of failures for apps which are assembled from Smali, Jasmin, and Jimple, and original ones. Using these values, we are able to compute a percentage, analytic values, to determine if there are any notable differences in program behaviours between the re-assembled apps and original ones in terms of the number of successfully injected events. We compute the percentage for each tool as follows:

$$P_{Tool} = \left(1 - \frac{Tool\_failures - Original\_failures}{Total\_apps}\right) \times 100\%$$

where *Tool_failures* is the total number of failures for the apps which are assembled from Smali, Jasmin, and Jimple; *Original_failures* is the total number of failures for the original apps, i.e., apps without any transformation of their executables; *Total_apps* is the total number of apps tested. Note that since we use 1,300 Android apps in our study,

TABLE 2. Number of apps failing Monkey test for 26 categories tested.

No.	Category	Tool name–Intermediate language			Original
		Apktool–Smali	*Dex2jar–Jasmin*	*Soot–Jimple*	
1	Books Reference	4	15	16	3
2	Business	5	12	18	3
3	Comics	5	17	15	4
4	Communication	7	25	14	6
5	Education	7	15	27	5
6	Entertainment	5	30	21	4
7	Finance	3	34	36	2
8	Health Fitness	8	16	16	8
9	Libraries Demo	6	15	15	5
10	Lifestyle	3	27	17	2
11	Live Wallpaper	6	17	20	4
12	Media Video	7	9	11	7
13	Medical	8	12	10	5
14	Music Audio	4	22	10	3
15	News Magazines	10	18	15	7
16	Personalization	5	22	12	3
17	Photography	4	24	21	2
18	Productivity	7	25	14	4
19	Shopping	11	28	29	9
20	Social	6	26	30	3
21	Sports	3	16	10	3
22	Tools	5	22	26	4
23	Transportation	4	20	20	3
24	Travel Local	7	16	14	5
25	Weather	8	24	16	7
26	Widgets	9	19	17	8
Total number of failures		***157***	***526***	***470***	***119***
Original program behaviours preservation, %		***~97***	***~69***	***~73***	**N/A**

the latter parameter is constant, i.e., $Total\_apps = 1,300$. In particular, if we compute the percentage, for example, for Smali, we define the values of the parameters as follows:

Tool_failures = 157 (see Table 2, intersection of row "Total number of failures" and column *"Apktool–Smali"*).
Original_failures = 119 (see Table 2, intersection of row "Total number of failures" and column "Original").

Therefore, we compute the percentage for *Apktool* as follows: $P_{Apktool} = (1 - \frac{157-119}{1,300}) \times 100\% \approx 97\%$. Analogously, we compute the percentage for *dex2jar*, and *Soot*. For *dex2jar*, $P_{Dex2jar} = (1 - \frac{526-119}{1,300}) \times 100\% \approx 69\%$, and for *Soot*, $P_{Soot} = (1 - \frac{470-119}{1,300}) \times 100\% \approx 73\%$.

To verify the above-mentioned computations, in Fig. 6, we graphically interpret the results from Table 2. There are four areas in Fig. 6, which are referred to as SMALI, JASMIN, JIMPLE, and ORIGINAL. Vertical axis represents 100% STACKED FAILING AREA for every category. The horizontal axis represents APPLICATION CATEGORIES.

TABLE 3. Statistic of original *Google Chrome* app for 5 replications (Monkey tests).

Replication	Total injected events (maximum 10,000)	Log-transformed values of "Total injected events"
1	65	0.105762
2	345	0.106431
3	403	0.106494
4	308	0.106386
5	345	0.106431

TABLE 4. One-sample t-test results for original *Google Chrome* app.

Replications	T-value	P-value	Significance level
5	–4.12	0.007	0.05

To calculate the areas values, for example, for *Book Reference*, we obtained a number of apps which have failed Monkey test from Table 2. That is, 4 apps for SMALI, 15 for JASMIN, 16 for JIMPLE, and 3 for ORIGINAL. By summing

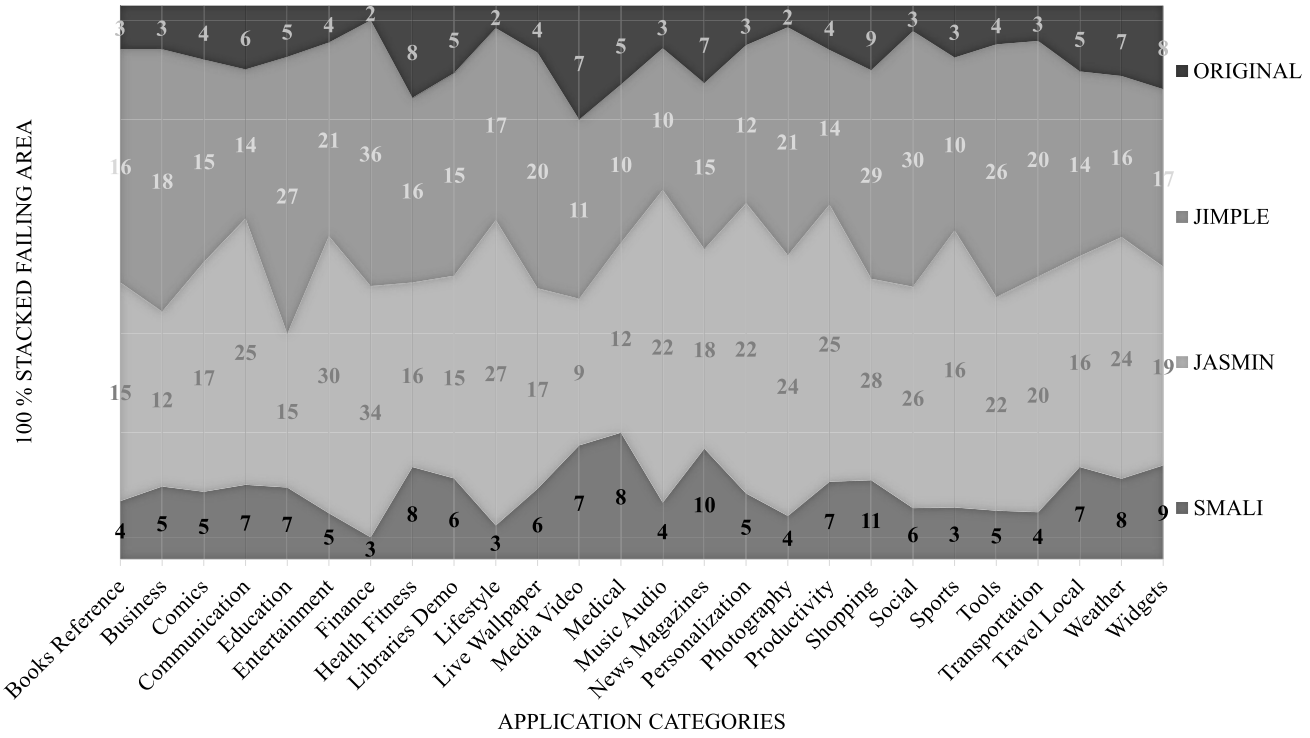

FIGURE 6. Number of apps failing Monkey tests for 26 categories tested.

all these values, it gives us 38 apps in total. Since Fig. 6 represents 100% STACKED FAILING AREA, we assume that 38 apps are 100%, and distribute the failing apps accordingly in their respective areas. Analogously, for other categories, we obtain failing areas for SMALI, JASMIN, JIMPLE, and ORIGINAL. As a result, in Fig. 6, we obtain the visual representation of differences in failing areas between SMALI, JASMIN, JIMPLE, and ORIGINAL. Comparing the failing areas, we can see that SMALI failing area is the closest to ORIGINAL, from which we hypothetically conclude that the apps, which are assembled from Smali, most accurately preserve their original behaviours in terms of the number of successfully injected events, comparing to those which are assembled from Jasmin, and Jimple. As a result, we identify a candidate tool which performs the most accurate transformation of the Android apps executables, i.e., *Apktool*. To experimentally verify our hypothetical conclusions, we perform *two-sample t*-test.

B. TWO-SAMPLE PARAMETRIC T-TEST

We perform *two-sample t*-test to identify whether the apps, which are assembled from Smali, Jasmin, and Jimple ILs have introduced any notable differences in their behaviours comparing to the original ones. In this section, we experimentally verify our hypothesis about the candidate tool which has been identified in Section IV-A, and hypothetical conclusions about Smali, Jasmin, and Jimple ILs are indeed valid.

For that purpose, we choose the following criteria to verify [32]–[34]:

Hypothesis:

$$H_0: \mu_1 = \mu_2$$
$$H_1: \mu_1 \neq \mu_2$$

Assumption:

$$\sigma_1^2 \neq \sigma_2^2$$

Test Statistic:

$$t_0 = \frac{\bar{y}_1 - \bar{y}_2}{\left(\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}\right)^{1/2}}$$

where $\bar{y}_1$ and $\bar{y}_2$ are the sample means of the first and second samples, respectively; S_1^2 and S_2^2 are the sample variances of the first and second samples, respectively; n_1 and n_2 are the sizes of the first and second samples, respectively; σ_1^2 and σ_2^2 are unknown and unequal variances of the first and second samples, respectively.

In our experiment, if H_0 is true, we conclude that a particular tool *performs* the most accurate transformation of the Android apps executables into its respective IL, and the apps which are assembled from such IL, *exhibit* their behaviours closest to the original ones. If H_0 is not true, we accept alternative hypothesis H_1 which implies that a particular tool *does not perform* the most accurate transformation of the Android apps executables into its respective IL, and the apps which are assembled from such IL, *do not exhibit* their behaviours closest to the original ones.

TABLE 5. Two-sample t-test results for Apktool, Dex2jar, and Soot.

Tool	Total data values	T-value	P-value	Significance level
Apktool	26	–1.89	0.093	0.05
Dex2jar	26	–3.99	0.001	0.05
Soot	26	–3.45	0.002	0.05

Parameters $\bar{y}_1$, $\bar{y}_2$ and S_1^2, S_2^2 are evaluated according to [32]–[34] as follows:

$$\bar{y}_k = \frac{\sum_{i=1}^{n_k} y_{i_k}}{n_k}$$

$$S_k^2 = \frac{\sum_{i=1}^{n_k} (y_{i_k} - \bar{y}_k)^2}{n_k - 1}$$

where y_{i_k} represents a sample, $i_k = \overline{1, n_k}$, $k = \{1, 2\}$.

To interpret the obtained results, we use the *P*-value approach with the following assumptions [32]–[34]:

1) If *P*-value is less than 5% level of significance, the null hypothesis H_0 would be rejected in favour of alternative hypothesis H_1.
2) If *P*-value is greater than 5% level of significance, the null hypothesis H_0 would not be rejected.

For *two-sample t*-test, we use data values for *Apktool–Smali*, *Dex2jar–Jasmin*, *Soot–Jimple*, and Original which we obtained from *one-sample t*-test (see Table 2). However, we found that *one-sample t*-test results are *log-normally* distributed. Therefore, to use parametric test, we need to have data values which are *normally distributed* or *approximately normally distributed*. So, similarly as described in Section III-C, we first apply the logarithmic transformation method to obtain *approximately normally distributed* data values. Afterwards, using the log-transformed data values, we perform parametric paired *two-sample t*-test on *Apktool–Smali*, *Dex2jar–Jasmin*, and *Soot–Jimple* with Original.

In Table 5, we show the results of *two-sample t*-tests for *Apktool–Smali*, *Dex2jar–Jasmin*, and *Soot–Jimple*. Comparing the obtained *P*-values for each tool with the significance level, we can see that the obtained *P*-value ($P = 0.093$) for *Apktool* is greater than required level of significance ($\alpha = 0.05$). Therefore, we conclude that *Apktool* performs the most accurate transformation of the apps executables since the apps, which are assembled from Smali, are indeed most closely preserve their original behaviours in terms of the number of successfully injected events.

V. RELATED WORK

Multiple techniques can be used to facilitate researches, and practitioners in the challenging task of program comprehension. One of these techniques is reverse engineering [35]–[37], the process of extracting and consolidating the high-level design information from an executable source. A reverse engineering helps to analyse the executable source of a software system or program in order to identify its components and inter-relationships, and generate corresponding reports or intermediate representations of the source in the textual or graphical forms, usually at a higher level of abstraction [38]–[40].

Multiple studies have been conducted in the field, which perform comparison and evaluation of various reverse engineering tools and approaches. Cutting and Noppen [41] introduce the benchmark, namely RED-BM (Reverse Engineering to Design BenchMark), which consists of a large set of Java sources for reversing, and a set of measures to evaluate and compare tools performance and effectiveness in a systematic manner. The benchmark facilitates the analysis of the reverse engineering tools based on their ability to reconstruct class diagrams. Also, it provides a set of measures which facilitate the comparison and ranking of the tools. To evaluate the performance of the benchmark, the authors used 12 commercial and open source reverse engineering tools. The performance results vary from 8.82% to 100% demonstrating the ability of the benchmark to differentiate between tools.

Fülöp [42] and Fülöp *et al.* [43]perform the evaluation of reverse engineering tools using the proposed benchmark, namely BEFRIEND (BEnchmark For Reverse engInEering tools workiNg on source coDe), the benchmark which supports different types of the reversing tools, programming languages, software systems, and also enables the users to define their own evaluation criteria. The authors suggest that using their benchmark, the reverse engineering tools results can be evaluated and compared effectively and efficiently. For example, with BEFRIEND, the results of the tools from different domains recognizing arbitrary characteristics of source code, and can be subjectively evaluated and compared with each other. The results could be from tools such as ''design pattern'' or ''bad code smell'' miners, ''duplicated code'' detectors, ''coding standard violation'' checkers, and others. Using this benchmark as a fundamental basis, the authors plan to create a universal, and generally applicable benchmark that can facilitate the evaluation, and comparison of various reverse engineering tools.

Lamprier *et al.* [44] propose CARE, a uniform platform for comprehensive evaluation of behaviour miners (reversing tools). The CARE is approach-independent, and only requires a set of artificial data (programs and execution traces) as input to produce a behaviour model such as Finite State Automata (FSA) which uses states to represent the execution flow of the program. The authors designed CARE platform with an ultimate goal to (1) provide a benchmark mechanism for reverse engineering tools comparison, (2) allow comparison of the tools based on a type of programs and behavioural patterns, and (3) help users in choosing the most accurate reversing tool for their objectives.

Gueheneuc *et al.* [45] present a comprehensive comparative framework for design recovery tools. The framework is built upon the authors' experience, and also extends existing frameworks. The authors evaluated the framework on two design recovery tools, namely Ptidej and LiCoR.

The obtained results show that the comparative framework is well-defined, comprehensive, and universal. However, further validation is required to verify whether the framework enables an objective comparison of tools.

Arcelli *et al.* [46] perform a comparison of reverse engineering tools based on the design pattern decomposition. The authors suggest that in reverse engineering, the recognition of design patterns provides additional information related to the rationale behind the implemented system design. In particular, the design patterns do not only provide information about how the architecture has been built, but also why it has been built in a such specific way. In their experiment, the authors formalize the design patterns according to the drawbacks they may represent. In turn, such formalization leads to the identification of the sub-patterns which are recurring fundamental elements of the design patterns. So, using the sub-patterns, the authors analyse the role of sub-patterns by evaluating two reverse engineering tools FUJABA and SPQR. During the evaluation process, the main focus was on how sub-patterns are exploited by the system to define and detect the implemented design patterns. In order to emphasize the similarities and differences between the tools, in the case-study, the authors used a Composite Design Pattern.

Arnatovich *et al.* [47] evaluate reversing capabilities of three reverse engineering tools based on the quality assessment of the untransformed apps' semantics preservation in the generated intermediate representations. For their evaluation, the authors use Smali, Jasmin, and Jimple intermediate representations. They obtained a set of 520 Android apps, re-assembled them with reversing tools using the intermediate representations, and applied randomized event-based testing with statistical analysis of the test results. Based on the obtained results, the authors suggest that Smali most accurately preserves the untransformed apps' semantics.

Bellay and Gall [48] performed an evaluation of four commercial reverse engineering tools. They used Refine/C, Imagix4D, SNiFF+, and Rigi tools which analyse C-code. The main focus was on the tools capabilities to generate graphical reports such as *Call*-graphs, *Control*- and *Data*-flow graphs. The authors investigated the capabilities of the tools by applying them to a real-world embedded software system. In their experiment, they used various assessment criteria for tools evaluation, e.g., the capabilities of parsing techniques, capabilities of generation of intermediate representations via analysis of properties of textual and graphical reports, capabilities of browsing and editing source code facilities, and general capabilities such as supported platforms, extensibility, and searching features.

Armstrong and Trudeau [49] evaluated five reverse engineering tools SNiFF+, Rigi, CIA, Dali, and Bookshelf. The main focus was on the tools capabilities to extract an architectural design from the source code. In particular, they focused on the abstraction and visualization of the software system components and their inter-relationships. Their study reports that Dali has the same visualization capabilities as Rigi since it uses Rigi's graph editor, SNiFF+ has limited visualization capabilities, and the Bookshelf and CIA tools have a performance issue while displaying large graphs so they should not be used for medium-to-large sized systems.

Koskinen and Lehmonen [50] identified a necessity to analyse systematically, and in detail the information retrieval capabilities provided by the reverse engineering tools. For that purpose, the authors compare ten the most popular reverse engineering tools Eclipse Java Development Tools (JDT), Wind River Workbench (WRW), Understand, Imagix4D, Creole, Javadoc, Javasrc, Source Navigator, Doxygen, and HyperSoft. They evaluate the reverse engineering tools capabilities in terms of four aspects including the *data structures*, *visualization* and *information request specification* mechanisms, and *navigation features*. The authors report that, for *data structures*, 80% of the tools can generate *Call*-graph, *Class* diagram is provided by 70% of the tools, and only 30% of the tools are able to do an automated re-documentation. For *visualization* mechanisms, 90% of the tools are able to view the executable source in a readable form, while 80% of the tools mainly focus on *Call*-graph visualizations. For *information request specification* mechanisms, 80% of the tools implement a basic filtering, and expansion of the *Call*-graph contents. Majority of the tools provide traditional text search mechanisms, and regular expressions are supported only by 40% of the tools. And, for *navigation features*, navigation capabilities among methods, any kinds of data elements, and browsing history are supported by 80–90% of the tools.

VI. CONCLUSION

In this paper, we developed a novel, statistical event-based comparative approach to examine an accuracy of the Android ILs transformation via an automated program behaviours validation. For our evaluation, we use Monkey to automatically generate the sequences of GUI pseudo-random user and system events. We apply *one-sample* and *two-sample t*-tests to empirically identify a reversing tool which performs the most accurate transformation of the Android apps executables into its respective IL.

We evaluated our approach performance on 1,300 real-world Android apps which were obtained from 26 distinct categories on *Google Play*. We used *Apktool*, *dex2jar*, and *Soot* reversing tools to perform transformation of the Android apps executables into their respective ILs, i.e., Smali, Jasmin, and Jimple. We identify the program behaviours differences in the apps, which are assembled from Smali, Jasmin, Jimple, and original ones, based on the number of successfully injected events. We found that *Apktool* performs the most accurate transformation of the Android apps executables since the apps, which are assembled from Smali, most closely preserve their original behaviours. Our results show that the apps which are re-assembled by *Apktool*, using Smali IL, preserve ∼97%, while *dex2jar*, and *Soot* using Jasmin, and Jimple, preserve ∼69%, and ∼73% of their original behaviours, respectively.

REFERENCES

[1] T. Systä and K. Väänänen-Vainio-Mattila, "On empirical studies to analyze the usefulness and usability of reverse engineering tools," in *Proc. IEEE Int. Workshop Softw. Technol. Eng. Pract. (STEP)*, Budapest, Hungary, Sep. 2005. [Online]. Available: http://post.queensu.ca/ zouy/files/preproc-step-2005.pdf

[2] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong, "Reverse engineering: A roadmap," in *Proc. Conf. Future Softw. Eng. (ICSE)*, New York, NY, USA, 2000, pp. 47–60.

[3] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: Converting Android dalvik bytecode to jimple for static analysis with soot," in *Proc. ACM SIGPLAN Int. Workshop State Art Java Program Anal. (SOAP)*, New York, NY, USA, 2012, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/2259051.2259056

[4] J. Dolby, "Program analysis for mobile: How and why to run WALA on your phone," in *Proc. 3rd Int. Workshop Mobile Develop. Lifecycle (MobileDeLi)*, New York, NY, USA, 2015, pp. 47–48. [Online]. Available: http://doi.acm.org/10.1145/2846661.2846673

[5] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot framework for Java program analysis: A retrospective," in *Proc. Cetus Users Compiler Infrastruct. Workshop*, Galveston, TX, USA, Oct. 2011, pp. 1–43.

[6] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proc. 10th ACM Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, 2003, pp. 290–299. [Online]. Available: http://doi.acm.org/10.1145/948109.948149

[7] T. Kistler and M. Franz, "A tree-based alternative to Java byte-codes," *Int. J. Parallel Programm.*, vol. 27, no. 1, pp. 21–33, 1999. [Online]. Available: http://dx.doi.org/10.1023/A:1018740018601

[8] T. Dullien and S. Porst, "REIL: A platform-independent intermediate representation of disassembled code for static code analysis," in *Proc. CanSecWest*, 2009, pp. 1–7.

[9] D. Chisnall, "The challenge of cross-language interoperability," *Commun. ACM*, vol. 56, no. 12, pp. 50–56, Dec. 2013. [Online]. Available: http://doi.acm.org/10.1145/2534706.2534719

[10] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A. D. Schmidt, and S. Albayrak, "Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications," in *Proc. 6th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Oct. 2011, pp. 66–72.

[11] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth, "Slicing droids: Program slicing for smali code," in *Proc. 28th Annu. ACM Symp. Appl. Comput. (SAC)*, New York, NY, USA, 2013, pp. 1844–1851. [Online]. Available: http://doi.acm.org/10.1145/2480362.2480706

[12] R. Johnson, Z. Wang, C. Gagnon, and A. Stavrou, "Analysis of Android applications' permissions," in *Proc. IEEE 6th Int. Conf. Softw. Secur. Rel. Companion (SERE-C)*, Jun. 2012, pp. 45–46.

[13] V. Rastogi, Y. Chen, and X. Jiang, "DroidChameleon: Evaluating Android anti-malware against transformation attacks," in *Proc. 8th ACM SIGSAC Symp. Inf., Comput. Commun. Secur. (ASIA CCS)*, New York, NY, USA, 2013, pp. 329–334. [Online]. Available: http://doi.acm.org/10.1145/2484313.2484355

[14] J. Meyer, T. Downing, and A. Shulmann, *Java Virtual Machine*. Sebastopol, CA, USA: O'Reilly & Associates, Apr. 1997.

[15] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification*. Boston, MA, USA: Pearson, 2014.

[16] J. Gosling, "Java intermediate bytecodes: ACM SIGPLAN workshop on intermediate representations (IR'95)," *ACM SIGPLAN Notice*, vol. 30, no. 3, pp. 111–118, Mar. 1995.

[17] R. Vallee-Rai, "The jimple framework," School Comput. Sci., Sable Res. Group, McGill Univ., Montreal, QC, Canada, Sable Tech. Rep. 1, Feb. 1998. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.52.2166&rep=rep1&type=pdf

[18] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing Java bytecode using the Soot framework: Is it feasible?" in *Compiler Construction*. Berlin, Germany: Springer, Jun. 2000, pp. 18–34. [Online]. Available: http://dx.doi.org/10.1007/3-540-46423-9_2

[19] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *Proc. Conf. Adv. Stud. Collaborative Res. (CASCON)*, 1999, pp. 214–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008

[20] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying Java bytecode for analyses and transformations," Sable Res. Group, McGill Univ., Montreal, QC, Canada, Sable Tech. Rep. 1998-4, Jul. 1998.

[21] D. Bornstein, "Dalvik VM internals," in *Proc. Google I/O Develop. Conf.*, vol. 23. 2008, pp. 17–30.

[22] D. Ehringer, "The Dalvik virtual machine architecture," Tech. Rep., Mar. 2010, vol. 4, no. 8. [Online]. Available: http://show.docjava.com/posterous/file/2012/12/10222640-The_Dalvik_Virtual_Machine.pdf

[23] J. Huang, "Understanding the Dalvik virtual machine," Google Technol. User Groups, Taipei, Taiwan, Tech. Rep., 2012. [Online]. Available: https://www.slideshare.net/jserv/understanding-the-dalvik-virtual-machine

[24] M. Gargenta, *Learning Android*. Sebastopol, CA, USA: O'Reilly Media, 2011.

[25] S. Christey, *The Infinite Monkey Protocol Suite (IMPS)*, document RFC 2795, 2000.

[26] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for Android: Are we there yet?" in *Proc. 30th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, Nov. 2015, pp. 429–440.

[27] N. Nyman, "Using monkey test tools," *Softw. Test. Quality Eng.*, vol. 29, no. 2, pp. 18–21, 2000.

[28] K. Benoit, "Linear regression models with logarithmic transformations," London School Econ., Methodol. Inst., London, U.K., Tech. Rep., 2011. [Online]. Available: https://pdfs.semanticscholar.org/169c/c9bbbd77cb7cec23481f6ecb2ce071e4e94e.pdf

[29] D. Hull, "Using statistical testing in the evaluation of retrieval experiments," in *Proc. 16th Annu. Int. ACM SIGIR Conf. Res. Develop. Inf. Retr. (SIGIR)*, New York, NY, USA, 1993, pp. 329–338. [Online]. Available: http://doi.acm.org/10.1145/160688.160758

[30] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*. Boca Raton, FL, USA: CRC Press, 2003.

[31] C. Davis, *SPSS for Applied Sciences: Basic Statistical Testing*. Melbourne, VIC, Australia: CSIRO, 2013.

[32] D. C. Montgomery, *Design and Analysis of Experiments*. Hoboken, NJ, USA: Wiley, 2017.

[33] R. L. Mason, R. F. Gunst, and J. L. Hess, *Statistical Design and Analysis of Experiments: With Applications to Engineering and Science*, vol. 474. Hoboken, NJ, USA: Wiley, 2003.

[34] C. Davis, *Statistical Testing in Practice With StatsDirect*. Tamarac, FL, USA Llumina Press, 2010.

[35] A. Mishra, "Reverse engineering: The promising technology," in *Proc. Covenant Univ. Conf. (SEIS)*, 2010, pp. 1–25.

[36] W. Wang, *Reverse Engineering: Technology of Reinvention*. Boca Raton, FL, USA: CRC Press, 2010.

[37] E. Eilam, *Reversing: Secrets of Reverse Engineering*. Hoboken, NJ, USA: Wiley, 2011.

[38] C. Raibulet, F. A. Fontana, and M. Zanoni, "Model-driven reverse engineering approaches: A systematic literature review," *IEEE Access*, vol. 5, pp. 14516–14542, 2017.

[39] H. Washizaki, Y.-G. Guéhéneuc, and F. Khomh, "A taxonomy for program metamodels in program reverse engineering," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Oct. 2016, pp. 44–55.

[40] E. J. Chikofsky and J. H. Cross, II, "Reverse engineering and design recovery: A taxonomy," *IEEE Softw.*, vol. 7, no. 1, pp. 13–17, Jan. 1990.

[41] D. Cutting and J. Noppen, "An extensible benchmark and tooling for comparing reverse engineering approaches," *Int. J. Adv. Softw.*, vol. 8, nos. 1–2, pp. 115–124, 2015.

[42] L. J. Fülöp, "Evaluating and improving reverse engineering tools," Ph.D. dissertation, Univ. Szeged, Szeged, Hungary, 2011.

[43] L. J. Fülöp, P. Hegedus, R. Ferenc, and T. Gyimóthy, "Towards a benchmark for evaluating reverse engineering tools," in *Proc. 15th Work. Conf. Reverse Eng.*, Oct. 2008, pp. 335–336.

[44] S. Lamprier, N. Baskiotis, T. Ziadi, and L. M. Hillah, "CARE: A platform for reliable comparison and analysis of reverse-engineering techniques," in *Proc. 18th Int. Conf. Eng. Complex Comput. Syst.*, Jul. 2013, pp. 252–255.

[45] Y. G. Gueheneuc, K. Mens, and R. Wuyts, "A comparative framework for design recovery tools," in *Proc. Conf. Softw. Maintenance Reeng. (CSMR)*, Mar. 2006, pp. 1–10.

[46] F. Arcelli, S. Masiero, C. Raibulet, and F. Tisato, "A comparison of reverse engineering tools based on design pattern decomposition," in *Proc. Austral. Softw. Eng. Conf.*, Mar. 2005, pp. 262–269.

[47] Y. Arnatovich, H. B. K. Tan, S. Ding, K. Liu, and L. K. Shar, "Empirical comparison of intermediate representations for Android applications," in *Proc. SEKE*, 2014, pp. 205–210.
[48] B. Bellay and H. Gall, "A comparison of four reverse engineering tools," in *Proc. 4th Work. Conf. Reverse Eng.*, Oct. 1997, pp. 2–11.
[49] M. N. Armstrong and C. Trudeau, "Evaluating architectural extractors," in *Proc. 5th Work. Conf. Reverse Eng.*, Oct. 1998, pp. 30–39.
[50] J. Koskinen and T. Lehmonen, "Analysis of ten reverse engineering tools," in *Advanced Techniques in Computing Sciences and Software Engineering*. Dordrecht: The Netherlands, Springer, 2010, pp. 389–394. [Online]. Available: https://doi.org/10.1007/978-90-481-3660-5_67

YAUHEN LEANIDAVICH ARNATOVICH received the bachelor's and master's degrees from the Belarusian State University of Informatics and Radioelectronics, Belarus. He is currently pursuing the Ph.D. degree with Nanyang Technological University, Singapore. His research interests include program analysis with a focus on how an automated software testing helps to improve the quality of the mobile apps. He is a (co-)author of six papers, of which two are in journals, and four on the international conferences as full papers. His current research interests include vulnerabilities and faults detection using formal and statistical approaches with their applications to the mobile security. He is currently involved in a project concerning automatic generation of the tests with high coverage, vulnerabilities, and faults detection rate via an automated exercising of the Android apps GUI components, and user input generation via dynamic symbolic execution.

LIPO WANG received the bachelor's degree from the National University of Defense Technology, China, and the Ph.D. from Louisiana State University, Baton Rouge, LA, USA. He has (co-)authored 300 papers, of which 100 are in journals. He holds a U.S. patent in neural networks and a patent in systems. He has co-authored two monographs and (co-)edited 15 books. His research interests include intelligent techniques with applications to optimization, communications, image/video processing, biomedical engineering, and data mining. He was a member of the Board of Governors of the International Neural Network Society (for two terms), the IEEE Computational Intelligence Society (for two terms), and the IEEE Biometrics Council. He served as the CIS Vice President for Technical Activities. He was the President of the Asia-Pacific Neural Network Assembly (APNNA). He received the APNNA Excellent Service Award. He served as the Chair of Emergent Technologies Technical Committee and the Chair of Education Committee of the IEEE Engineering in Medicine and Biology Society (EMBS). He was the founding Chair of the EMBS Singapore Chapter and CIS Singapore Chapter. He serves/served as chair/committee members for over 200 international conferences. He was/will be a keynote/panel speaker for 30 international conferences. He is/was an Associate Editor/Editorial Board Member of 30 international journals, including three IEEE TRANSACTIONS, and a Guest Editor for 10 journal special issues.

NGOC MINH NGO received the bachelor's and Ph.D. degrees from Nanyang Technological University, Singapore. She was with the National University of Singapore as a Post-Doctoral Research Fellow, where she studied different approaches to automatically locate various software failures. She was with the Singapore Institute of Management as a Lecturer. She is currently an Instructional Designer with Arizona State University, Vietnam, and has been an Associate Faculty with the Singapore Institute of Technology since 2014, where she teaches in information technology. She has (co-)authored 17 papers, of which seven are in journals, and six on the international conferences as full papers. Her research interests include software quality, and how the software design, program analysis, and automated software testing help to improve the quality of the software systems.

CHARLIE SOH received the bachelor's degree from Nanyang Technological University, Singapore, where he is currently pursuing the Ph.D. degree. He has (co-)authored five papers, of which two are in journals, and three on the international conferences as full papers. His research interests include machine learning, deep learning, and natural language processing (NLP) with a focus on their applications to the mobile security for Android. His current research interests include the versatile representations of Android apps, which can be leveraged for various mobile security tasks. He is currently involved in a project concerning the identification of potential insider threats using NLP techniques.