Empirical Comparison of Intermediate Representations for Android Applications

Yauhen Leanidavich Arnatovich, Hee Beng Kuan Tan, Sun Ding, Kaping Liu
INFINITUS, Infocomm Centre of Excellence, School of Electrical and Electronic Engineering, Nanyang Technological University, 50 Nanyang Avenue, Singapore 639798 {yauhen001, ibktan, ding0037, kpliu}@ntu.edu.sg

Lwin Khin Shar
Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg lwinkhin.shar@uni.lu

Abstract—In Android-based mobile computing, since the original Java source code is irretrievable from Dalvik bytecode, intermediate representations (IRs) were developed to represent Dalvik bytecode in readable form. To date, SMALI, JASMIN, and JIMPLE are all used as Android application IRs by mobile developers, testers and researchers. Here, we compare these three IRs via randomized event-based testing (Monkey testing) to determine that which most accurately preserves the original program behaviors in terms of the number of successfully injected events. As such program behaviors are critical to mobile security, the choice of IR is crucial during software security testing. In our experiment, we developed an event-based comparative scheme, and conducted a comprehensive empirical study. Statistical comparison of the three IRs’ program behaviors shows that SMALI behaves closest to the original applications and hence is the most suitable for software security testing as the most accurate alternative to the original Java source code (which is usually not publicly available).

Keywords—intermediate representation; program behaviors; event-based testing; Android computing; SMALI; JASMIN; JIMPLE

I. INTRODUCTION

Android operating system usage is widespread, and many Android applications are being developed every day. All these applications are originally written in the Java programming language. To be executed on an Android-powered mobile device, Java source code must be compiled to Java bytecode, and then, transformed into Dalvik bytecode.

It is a fact that the Dalvik Virtual Machine (DVM) was especially developed for the Android mobile platform considering the limitations of mobile devices. In sharp contrast, the JVM is a register-based environment whereas the Java Virtual Machine (JVM) is stack-based. As the JVM and the DVM have different internals, there is no way to retrieve an original Java source code from Dalvik bytecode. Therefore, third-party tools can be used to obtain appropriate intermediate representations (IRs) such as SMALI [1], JASMIN [2], and JIMPLE [3] from Dalvik bytecode.

The latest Android research suggests that SMALI [4][5][6][7][8][9], JASMIN [10], and JIMPLE [11][12][13][14][15] can be used for Android software security testing purposes. But, the point at issue is that it is still unknown which IR most accurately preserves the original program behaviors. For example, carrying out software security testing on the least accurate IR could mislead test results by reflecting the original program behaviors mistakenly, i.e. it could change the original program internals and consequently generate new, change or eliminate existing vulnerabilities of the original application. In sharp contrast, if we used the most accurate IR for software security testing, such an IR could mostly retain the number of existing security risks of the original application, and provide the most accurate testing results so that the IR would be the most suitable for testing purposes. Consequently, the most accurate IR could guarantee an ability to detect a similar number of security risks to that of an original Java source code. Therefore, it is necessary to investigate which IR is the most accurate and loses the least of the original program behaviors during disassembling.

Inspired by this interest, our motivation, in this paper, is to find which IR most accurately preserves the original program behaviors in terms of the number of successfully injected events. This research is intended to help developers, testers and researchers to most accurately detect mobile security risks of Android applications without having an original Java source code.

Our main contributions of this study are the following:

- We developed an event-based comparative scheme to find which IR most accurately preserves the original program behaviors in terms of the number of successfully injected events. We run 520 applications from Google Play on our scheme and find that SMALI behaves closest to the original applications.
- Using an automated event-injected testing approach and statistics, we compare program behaviors of SMALI, JASMIN, and JIMPLE to original ones. Our results represent the benefits of the use of SMALI against JASMIN and JIMPLE for software security testing.

These research results can guide mobile developers, testers and researchers to use an appropriate IR in order to most accurately carry out software security testing of Android applications without having an original Java source code.

The rest of the paper is organized as follows: Section II covers the Dalvik Virtual Machine and the Java Virtual Machine features, and shows the differences between them. Section III describes our experiment design in detail. Section IV represents our experimental results. Section V discusses related work. Section VI concludes our paper.
II. FEATURES OF THE DALVIK VIRTUAL MACHINE AND THE DALVIK EXECUTABLE FILE FORMAT

In this section, we describe the special features of the DVM as a virtual machine especially developed for Android-based mobile computing, and stress the differences between the DVM and the JVM. Our goal, in this section, is to show that the Java bytecode structure has nothing similar to Dalvik bytecode, and consequently it would not be true to directly apply existing software security testing techniques to Dalvik bytecode.

A. The Dalvik VM Design Overview

The Dalvik VM is not a Java VM. Dalvik is the name of the Virtual Machine in which Android applications are to run. The Dalvik VM is a register-based environment, and it runs classes compiled by a Java compiler javac that have been further transformed into a single class (.classes.dex) by the Android SDK default dx tool. After applying dx tool, the Dalvik Executable (.dex) has only a distant relationship with Java bytecode (.class). The Dalvik VM executes the Dalvik Executable (.dex) file, which is optimized for minimal memory consumption.

As both applications and system services of the Android OS are developed in the Java language, the Dalvik VM has been written so that a device can run multiple VMs efficiently. Every Android application runs in its own process, with its own instance of the Dalvik VM, and it is referred to as sandboxing applications. When an Android-powered device is started, a single virtual machine process called Zygote is created, which preloads and pre-initializes core library classes. 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 Zygote VM instance. Thus, by spawning new VM processes from the Zygote, the startup time of another 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. Since every application runs in its own process within its own virtual machine, it takes advantage of not only efficient running of multiple VMs, but also fast creation of new VMs. The Dalvik VM relies on the Linux kernel for underlying functionality such as threading and low-level memory management.

The core library classes (.lib) that are shared across the Dalvik VM instances are commonly only read by applications. When the (.lib) classes are needed, the memory from the shared Zygote VM process is simply copied to the forked child process of the application’s Dalvik VM. Such behavior allows it to maximize the total amount of shared memory while still restricting applications from overlapping with each other and providing security across application and sandboxing individual processes [16][17][18].

B. The Dalvik Executable File Format (.dex)

The Dalvik Executable (.dex) format is designed to meet the requirements of systems that are constrained in terms of memory and processor speed. The .dex design is primarily driven by the sharing of data between running processes. The main difference between Java bytecode (.class) and Dalvik bytecode (.dex) is that all the classes of the Android application are packed into one file. 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 the Dalvik VM, classes from the same Dalvik Executable (.dex) file are loaded by the same class loader instance.

The .dex file format uses mechanisms for conserving the RAM of mobile device running Android OS. A 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. Instead of storing these values in the class, they always can be found by their indices in the Constant Pool. In the case of the .class file, each class has its own Constant Pool. But, the .dex file contains many classes in one file named classes.dex. All of these classes share the same type-specific Constant Pool in order to avoid the duplication of constants in the .dex file [16][18].

III. EXPERIMENT DESIGN

For our experiment, we use an Android-powered physical device running Android 4.1.2, android-apktool and dex2jar assembler/disassembler tools, Soot Java optimization framework, the Dumb Monkey test, and a 1-Sample Sign test statistic. Our experiment was conducted on a Core i5-2400 @ 3.10GHz with 8GB of RAM machine running Windows 8.

A. Experimental Design and Scheme

In our experiment, we identify which reassembled Android applications (SMALI, JASMIN, or JIMPLE) most accurately preserves the original program behaviors in terms of the number of successfully injected events. For that purpose, we downloaded 520 top free applications from Google Play (by 20 top free applications from 26 different categories). All downloaded applications are compatible with our testing Android-powered physical device. Since we have a data set of applications from Google Play, we consider them to be benign applications (not malware), and consequently not obfuscated or encrypted. Hence, we processed the 520 Android applications as follows: disassemble, assemble, sign, align, install, and run the Dumb Monkey test. Next, we reassembled applications with the aid of android-apktool, dex2jar, and Soot tools in order to obtain SMALI, JASMIN, and JIMPLE IRs, respectively. Also, we did not modify any IR code during reassembling. We applied the Dumb Monkey test to 520 Android applications. Every Dumb Monkey test generates a sequence of pseudo-random events based on a specified seed value. In our experiment, every Dumb Monkey test contains 30 trials for every category. Every trial contains the same set of 20 applications (from a particular category) assembled from SMALI, JASMIN, JIMPLE, and original applications. We use the constant seed value within a trial (to test the same set of 20 SMALI, JASMIN, JIMPLE, and original applications) to generate the same sequence of pseudo-random events, and be able to statistically compare the program behaviors of the IRs to the original ones in terms of the number of successfully injected events. Every trial has a unique seed value (unique sequence of pseudo-random events), which changes 30 times (since we have 30 trials) within a category.

The Dumb Monkey test has been chosen as our aim is to find which IR most accurately preserves the original program behaviors in terms of the number of successfully injected events. When the Dumb Monkey test has been done for every category, we apply a non-parametric 1-Sample Sign test statistic to be able
to conclude how close a particular IR reflects the behaviors of the original applications.

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

- Disassemble Android applications by using android-apktool, dex2jar, and Soot tools. Android-apktool, dex2jar, and Soot disassemble Android package (.apk) into SMALI (Dalvik Assembler), JASMIN (Java Assembler), and JIMPLE code (Java Simplified), respectively (see Section III C).

- Assemble Android applications by using android-apktool, dex2jar, and Soot tools. All Android applications were assembled without any code modifications.

- Sign Android applications with private keys. This is a mandatory requirement for any Android application if a particular application should be installed whether on an Android emulator or an Android-powered physical device. We applied a standard Java package to run the signing process.

- Align Android packages by using default Android zipalign tool. The archive alignment tool provides important optimization of Android packages (.apk). This tool aligns the Android packages on 4-byte boundaries and reduces the amount of RAM consumed when running the application. Notwithstanding, this tool is not a mandatory requirement to install applications, but highly recommended by Android developers.

- Install applications on an Android-powered physical device. We use a testing physical device running Android 4.1.2.

- Apply the Dumb Monkey test to all installed applications (to every 20 top free applications from 26 different categories). The Dumb Monkey test meets our requirements, and is sufficient to reveal the difference in behaviors between the IRs and the original applications. We run the Dumb Monkey test with the same sequence of pseudo-random events for SMALI, JASMIN, JIMPLE, and original applications. After every Dumb Monkey test, we stop all running applications to return them to the initial state and repeat the Monkey test again.

- Apply non-parametric statistical test. To identify whether a particular application failed the Dumb Monkey test, we use a 1-Sample Sign test, and consequently we are able to reveal the difference in behaviors between applications assembled from SMALI, JASMIN, JIMPLE, and the original applications. In particular, we statistically compare program behaviors of the IRs to the original ones.

B. Android Experimental Environment

In our experiment, we use the Android Debug Bridge (adb), and an Android-powered physical device. The Android Debug Bridge is a versatile Command Line tool that permits communication with an Android emulator instance or an Android-powered physical device [19][20].

To install and run the Dumb Monkey test on the Android-powered physical device, we use the cmd tool. Specifications of the Android-powered physical device on which we conducted our experiment are indicated below in Table I.

We configured the Android-powered physical device as follows:

- We install applications on the Android-powered physical device with the aid of adb tool.

- At the first time, we manually run the applications to provide the required information (login/password, phone number, and other normal means of access), for example, applications with a sign-in screen (Skype, OneDrive (formerly SkyDrive), Facebook, Facebook Messenger, Spotify, and others), and applications with a required phone number (Viber, WhatsApp, WeChat, and others). However, we could not provide the information for two-step authentication (one-time password, verification code), which requires human interaction. Also, we keep all applications running and they can be seen as “running services” or “cached background services” on the Android-powered physical device to allow the Monkey test to carry out testing.

- We have a gallery of the most popular audio/video files (.mp3, .mp4, .avi), pictures (.jpeg), and office documents (.doc, .xls, .ppt, .pdf) for the applications in case they use the gallery during the testing.

The Android-powered physical device was connected to the Internet, Bluetooth was turned on, and the device was on a mobile carrier service during the experiment.

C. Android Tools

In our experiment, we use some well-known existing reverse engineering tools for Android applications.

1) Android-apktool: Dalvik Disassembler

To conduct the experiment on SMALI, we chose the disassembler/assembler named android-apktool. Android applications typically use resources provided by the Android OS. The android-apktool needs these framework files to decode and build Android applications correctly. To install the framework, we run the following cmd code:

1. apptool if <framework_name>

The parameter <framework_name> specifies the path to the framework apk-file. Next, two commands are used to disassemble and assemble Android packages, respectively:

1. apptool d <apk_name>
2. apptool b <directory with decompiled apk name>

When an Android application is assembled, this tool automatically creates build/ and dist/ directories. The dist/ directory contains the fully assembled Android package (.apk).

<table>
<thead>
<tr>
<th>TABLE I. ANDROID-POWERED PHYSICAL DEVICE SPECIFICATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android version</td>
</tr>
<tr>
<td>RAM</td>
</tr>
<tr>
<td>Internal storage</td>
</tr>
<tr>
<td>External storage</td>
</tr>
<tr>
<td>Camera</td>
</tr>
<tr>
<td>Display</td>
</tr>
</tbody>
</table>
D. The Dumb Monkey Test

To process the large number of Android applications, we use an automated Monkey test tool [22][23]. We applied the Dumb Monkey test tool [24] bundled with the standard Android SDK. We created a batch script file to launch the Dumb Monkey test automatically. Our batch script contains the following command:

```
adb shell monkey -p <package_name> - v <pseudo_event_count>
```

We specified two parameters <package_name> with a particular Android (.apk) package. In our experiment, this parameter is used to identify a single package accessible to Monkey without any dependencies on other packages. The <pseudo_event_count> parameter is set to 2000 pseudo-random events.

IV. STATISTICAL ANALYSIS RESULTS

We applied the event-based comparative scheme, discussed in Section III, to 520 Android applications to find which IR most accurately preserves the original program behaviors in terms of the number of successfully injected events. We applied a non-parametric 1-Sample Sign test statistic to the Dumb Monkey test results to conclude if a particular application failed the Monkey test. From the 1-Sample Sign test results, we count the number of applications failing the Dumb Monkey test for every category, and, based on the obtained numbers, graphically show the difference in program behaviors between SMALI, JASMIN, JIMPLE, and original applications. We perform statistical analysis with the aid of MINITAB statistical software. To be able to apply the 1-Sample Sign test, we have performed a normality test on the data (Monkey test results) and found that the data are non-normally distributed and non-symmetric.

We run 30 trials for 20 Android applications from every category. After the Dumb Monkey test, the result contains 600 values (each value represents the number of successfully injected events) for a particular category. To compare the IRs to the original applications appropriately, the same sequence of pseudo-random events was applied to the applications assembled from SMALI, JASMIN, JIMPLE, and original applications.

We chose the following hypotheses testing criteria to verify whether the Dumb Monkey test was passed:

\[ H_0 : \mu = \mu_0 \]
\[ H_1 : \mu < \mu_0 \]

where \( H_0 \) is null hypothesis and \( H_1 \) is an alternative hypothesis. If \( H_0 \) is true, a particular application passed the Dumb Monkey test. If \( H_0 \) is not true, we accept an alternative hypothesis \( H_1 \). To carry out the 1-Sample Sign test, we set \( \mu_0 \) to 2000 pseudo-random events (the maximum number of pseudo-random events to be injected). To interpret the 1-Sample Sign test results, we use a P-value approach with a 0.05 level of significance. After performing the 1-Sample Sign test, we graphically show the final results to find the most accurate IR.

Figs. 1-3 clearly show the difference between SMALI, JASMIN, JIMPLE, and original applications, and help visually identify the IR, which most accurately preserves the original program behaviors. The dashed lines in Figs. 1-3 plot the number of applications which failed the Dumb Monkey test for SMALI, JIMPLE, and JASMIN applications, respectively, against the application categories on the horizontal x-axis. The dotted lines represent the number of the original applications which failed the Dumb Monkey test.

Comparing Figs. 1-3, it can be seen that the lines in Fig. 1 are the closest to each other, from which it is concluded that SMALI most accurately preserves the original program behaviors in terms of the number of successfully injected events. Thus, the applications, assembled from SMALI, behave closest to their respective original applications.
As JASMIN and JIMPLE both perform very similarly to each other and it is difficult to identify the difference in behaviors between them, we provide statistics in Table II to show the difference in accuracy of program behaviors between the IRs and the original applications.

### V. RELATED WORK

Previous studies were based on source code semantic analysis applying static or dynamic tools. The previous approaches were connected with extracting duplicated parts of the code, finding dependencies within the code written in different languages, identifying functionality from executable source which is poorly documented, or detecting merging and splitting of files and functions in procedural code.

Marcus and Maletic [27] propose an approach to scan the original code and try to identify the parts of the source code of similar high-level implementations. This approach uses an information retrieval technique to carry out a static analysis, and determine semantic similarities between source code documents and executable source code.

Ying et al. [28] propose a technique to predict source code changes by mining change history because it is difficult to find the entity dependency between source codes written in different languages. The new approach is to help developers identify relevant source code during a modification task. The proposed solution applies data mining techniques to determine change patterns from the change history of the code.

Eisenbarth et al. [29] present a semiautomatic technique that reconstructs the mapping for features triggered by the user and exhibits observable behaviors. This technique allows for the distinction between general and specific computational units with respect to a given set of features. This technique combines dynamic and static analyses.

Godfrey and Zou [30] propose an extended detection of merging and splitting of files and functions in procedural code. The improved approach shows how reasoning about how call relationships have changed and where merges and splits have occurred. Thus, it helps to recover some information about the context of the design change.

We have proposed another approach based on statistical comparison of the program behaviors. In sharp contrast to existing approaches, we are focusing on the program behaviors, but not any changes of source code. We identify the differences in the program behaviors between the IRs and the original applications based on the injection of the same sequences of pseudo-random events.

### VI. CONCLUSION

In this work, we examined three different IRs on 520 Android applications. We compared SMALI, JASMIN, and JIMPLE IRs to the original Android applications in terms of the number of successfully injected events via an event-based comparative scheme to determine the difference in their behaviors. We found that SMALI most accurately preserves and provides the closest reflection of the original program behaviors. We suggest that SMALI can be used by developers, testers and researchers as the most accurate alternative to an original Java source code for software security testing.

ACKNOWLEDGMENT

Lwin Khin Shar was supported by the National Research Fund, Luxembourg (grant FNR/P10/03).

REFERENCES


