InsDal: A Safe and Extensible Instrumentation Tool on Dalvik Byte-Code for Android Applications*

Jierui Liu[1,3], Tianyong Wu[1,3], Xi Deng[2,3], Jun Yan[†1,2,3], Jian Zhang[1,3]
[1]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[2]Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences
[3]University of Chinese Academy of Sciences
{liujr, wuty, yanjun, zj}@ios.ac.cn, dengxi215@mails.ucas.ac.cn

***Abstract*—Program instrumentation is a widely used technique in dynamic analysis and testing, which makes use of probe code inserted to the target program to monitor its behaviors, or log runtime information for off-line analysis. There are a number of automatic tools for instrumentation on the source or byte code of Java programs. However, few works address this issue on the register-based Dalvik byte-code of ever-increasing Android apps. This paper presents a lightweight tool, *InsDal*, for inserting instructions to specific points of the Dalvik byte-code according to the requirements of users. It carefully manages the registers to protect the behavior of original code from illegal manipulation, and optimizes the inserted code to avoid memory waste and unnecessary overhead. This tool is easy to use and has been applied to several scenarios (*e.g.* energy analysis, code coverage analysis). A demo video of our tool can be found at the website: https://www.youtube.com/watch?v=Fpw-aygZ3kE.**

I. Introduction

With the development of mobile Internet, Android smartphones are becoming more and more popular. The quality of Android apps attracts much attention from both industrial and academic communities. Naturally, program analysis and testing, two widely adopted techniques for software quality assurance, are hot topics in Android-related research in recent years.

There are two types of analysis approaches, static and dynamic, of which the latter one is more useful in some specific scenario where the values of variables may not be easily predicted statically (for example, implicit invocations). To perform dynamic analysis, we need to acquire the runtime information of target software. Program instrumentation, which only modifies the code of the target program instead of runtime environment, plays an important role in dynamic analysis for recording execution information of the program.

A lot of mature tools have been developed for instrumentation on source and byte code of Java programs [5]. Although the program analysis technique of Android apps is a hot issue, to the best of our knowledge, there is no research work that focuses on directly instrumenting the Dalvik byte-code. Many of the existing works make use of the Java source or byte code instrumentation tools to analyze the Android Apps, which limits the scope of approaches. We observe that most of apps are commercial and the source code is not available publicly. Thus, we believe it is a promising work to develop an automatic tool that can analyze the Android package file (.apk), find the proper places to insert minimum number of instructions to the DEX file, and repackage the modified files into an executable app.

There are three issues that need to be considered in designing this tool. First of all, as is well-known, the Dalvik virtual machine (DVM) is a register-based one and it is necessary to manage the registers carefully to avoid illegal manipulation to the behavior of original code, which may crash the app during execution. Second, for a relatively precise measurement of the target app, the overhead of the extra code should be reduced to a low level. At last, as a practical tool, it should be extensible, *i.e.*, it processes the instrumentation according to a set of user written rules for different application scenarios.

This paper is organized as follows. It first introduces the background knowledge of Dalvik byte code in Section II and presents the system design and implementation of our tool `InsDal` in Section III. Section IV shows the experimental results for evaluating `InsDal`, while Section V lists several application scenarios. The last two sections introduce the related works and conclude our work.

II. Dalvik and Smali Byte-Code

The source code of an Android app is compiled to register-based Dalvik byte-code, of which smali is an easily readable format. There are two kinds of registers in smali, the local registers and the parameter registers (used for storing parameters of the method). In the beginning of a method, the integer following the keyword `.locals` represents the number of local registers that can be used in this method, and the keyword `.registers` represents the total number of all registers (both local and parameter registers). There are two naming schemes for registers: the `v`-naming scheme for all registers and the `p`-naming scheme only for parameter registers.

Fig. 1 shows an example of a method `methodA` of class `me.test` in smali format. In the method, there are 15 local registers and one parameter register. It invokes another method `methodB` by the instruction `invoke-direct` with two

*This work is supported in part by the National Key Basic Research (973) Program of China (Grant No. 2014CB340701) and National Natural Science Foundation of China (Grant No. 61672505, 61361136002, 91418206).
†Corresponding author

operands, the parameter register p0 storing the reference of the object which the method methodA belongs to (*i.e.*, this in Java program), and local register v0.

```
.method public methodA()V
  .locals 15

  ......
  const/4 v0, 0x0
  invoke-direct {p0, v0}, Lme/test;->methodB(I)V
  ......

  return-void
.end method
```

Fig. 1. Smali code snippet

There are two important properties about the registers. The first one is that the parameters of the method are always stored at the last registers after all local registers. Recall the example smali code, the parameter register p0 will be stored in the 16th register, *i.e.* register p0 is the same as register v15. The second property of registers is that some instructions are only allowed to use a specific set of registers. For example, the invoke-direct instruction can only use the first 16 v-naming registers (v0 to v15). These two properties will bring challenges to the instrumentation. For instance, if we want to add a local register for instrumentation in the smali code in Fig. 1, then the parameter register p0 is changed to the 17th register (v16) according to the first property, which will violate the second property. Thus, it is essential for an instrumentation tool to carefully manage the registers to avoid illegal manipulation to the behavior of original code.

III. InsDal

In this section, we introduce the system architecture and techniques used in our instrumentation tool InsDal.

A. System Overview

The tool InsDal takes smali format byte-code as input, which can be easily extracted from the Android Package file (apk file). It provides a general interface to be easily extended for different user requirements of the instrumentation. Fig. 2 shows the high-level overview of InsDal, which mainly consists of three modules as follows.

- **Disassembling and Repackaging.** This module is used for obtaining the target Dalvik byte-code (disassembling) and rebuilding the modified files to a new Android app (repackaging). Specifically, this module works on top of Apktool [1] that is a reverse engineering tool for Android apps.
- **Analysis.** This module is used to analyze and determine the instrumented code according to user requirements, such as which instructions should be instrumented, the location where they are inserted, and how many local registers are added in each method.
- **Instrumentation.** This module is the most important one in our tool, which consists of three submodules. Register management submodule is used to find the safe registers used for our instrumented code and avoid to distort the original code logic. Target instruction insertion submodule is used to instrument the target instructions according to the analysis results. Efficiency optimization submodule aims to optimize our instrumentation to avoid unnecessary computational resource (*e.g.* memory) waste.

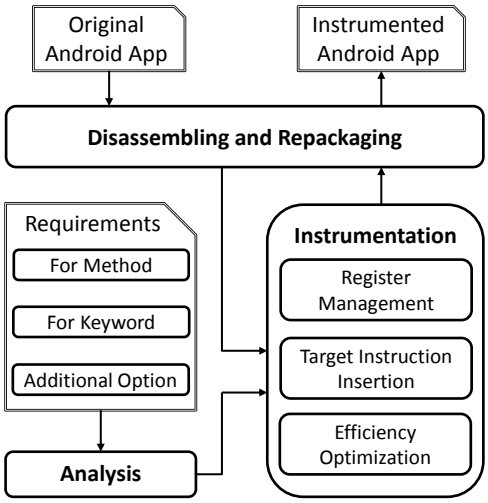

Fig. 2. High-level Overview of InsDal

B. Analysis Module

This module determines the inserted instructions according to user requirements. Up to now, InsDal mainly supports two instrumentation requirements: instrument for user-developed methods and instrument for given keywords. Besides, our tool also supports additional options in these two requirements.

1) Instrument for User-Developed Methods: This requirement aims to track the invocations of each user-developed method in the app. Our tool will instrument a log instruction in the beginning of each method in user-written classes. Since the log instruction requires two parameters, we will introduce two extra local registers in each method.

2) Instrument for Given Keywords: In this requirement, the user targets at some specific instructions involving given keywords. We will perform a simple string matching algorithm to find all such instructions and instrument a log instruction before them. Note that when an instruction contains multiple keywords, our tool will insert multiple log instructions for each keyword. For each method, we also introduce two extra local registers, and reuse them for multiple inserted log instructions.

3) Additional Option: Except for supporting the base instrumentation requirements, our tool also supports some additional options according to different purposes. For example, the tool supports logging the values of parameters of the methods when they are executed. This option needs to introduce one extra local register for each method.

C. Instrumentation Module

This module is the core component in our tool which consists of three submodules, register management, target instruction insertion and efficiency optimization.

1) Register Management: Section II discusses the challenges of register usage mechanism in smali. To avoid register conflict, we use the new registers that are not used in the original byte-code to serve the instrumented instructions, and manage the registers in a method by the following steps.

(a) **Calculate the number of parameter registers.** Since `.locals` statement does not show the number of parameter registers, we must infer the number of parameter registers from the declaration of the method.
(b) **Rewrite the register number.** If we want to introduce extra local registers, we need to modify the number that is declared in the register usage statement (`.locals` or `.register`).
(c) **Translate parameter registers to v-naming registers.** We copy the values of parameter registers to its corresponding `v`-naming registers to avoid the violation against the register properties due to the introduced local registers.
(d) **Update parameter registers.** After we translate the parameter registers, we need to update all of their occurrences to the `v`-naming registers.

Recall the code snippet in Fig. 1 again. Suppose we want to introduce 3 extra local registers. Fig. 3 shows the instrumented code after the register rearrangement. Through step (a), we know that this method uses only one parameter register. Then there are totally 19 registers that will be used in this method and we rewrite statement `.locals 15` as `.registers 19` (same as `.locals 18`). The extra `move` instruction translates the register `p0` to its corresponding `v`-naming register `v15`. Finally, we update all the occurrence of the register `p0` to `v15`. After these steps, the registers `v16` to `v18` are valid local registers for our instrumented instructions.

```
.method public methodA()V
  .registers 19 # 15 + 1 + 3 #

  move-object/16 v15, p0

  ......
  const/4 v0, 0x0
  invoke-direct {v15, v0}, Lme/test;->methodB(I)V
  ......

  return-void
.end method
```

Fig. 3. Smali code snippet after register rearrangement

2) Target Instruction Insertion: Based on the works that have been done in register management submodule, we can instrument the target instructions in this module according to the results from analysis module. Note that Android apps often contain some compatible and third-party classes, which are not our instrumentation target. To avoid instrumenting such classes, we obtain the package name of the app from the `AndroidManifest` configuration file and only process the smali files in the app's package directory.

3) Efficiency Optimization: Instrumentation will cause extra overhead on original apps. To make our tool more practical and avoid unnecessary computational resource (*e.g.* memory) waste, in this module, we optimize our instrumented code. Without such work, the modified app may work correctly, but may cause a heavy burden for the device.

For example, we instrument a global `StringBuilder` used for generating various target `String` object for every log instructions to reduce the overhead introduced by frequent creation and destruction of the String object. This `StringBuilder` variable will be inserted in the main Activity that is the first running Activity in an Android app and will be found in every Android app, and it is inserted as a `public static` field of the main Activity.

D. Usage

In this section, we will introduce how to use `InsDal` to instrument an Android app. This tool is developed in Java language and on top of the disassembling tool `Apktool`, so it can be easily deployed on different operating systems. Our tool is packaged in a `Jar` file and can be used as the following command in a terminal.

```
java –jar InsDal.jar [options]
```

Giving different `options` will drive our tool to run for different functionalities. And the specific options are shown as follows.

```
-f [arg]  Set the target APK file
-h        Show the help information
-m        Add logs for methods
-V        Print the parameters' values of the
          instrumented methods
-k [arg]  Add logs for keywords
-v        Print the arguments' values of the
          operations with keyword
-i        Output all instrumentation information
          into a file
-p        Preserve the decompiled files
-t [arg]  Set some specific words in tag string
```

The options `m` and `k` are used for specifying the instrumentation requirements, which can be set at same time. The option `V` is an additional option on instrumentation for user-developed methods to log parameter values and it works only when the option `m` is specified. Meanwhile, the option `v` is an additional option on instrumentation for given keywords and it works only when the option `k` is specified.

IV. Evaluation

To evaluate the effectiveness of our tool, we conduct an experiment on 10 real-world apps, 3 of which are from an open-source repository F-Droid and the rest ones are downloaded from WandouJia (a Chinese Android market). The statistic information of these apps is shown in Table I, including the app name, size of apk file (KB) and DEX file (KB), and the number of classes (#C) and methods (#M).

TABLE I
STATISTIC INFORMATION OF EXPERIMENTAL APPS

Apps	APK size	DEX size	#C	#M
Andless	300.1	112.2	69	450
Bluechat	62.5	38.3	47	154
GetBackGPS	249.5	104.3	69	459
Antivirus	15177	14176	6570	51404
Baidu	38659	14586	8086	51546
CheDengWo	3949	1240	1210	8799
FontMaster	5400	932	1407	9603
QiCaiScan	1176	1192	936	4881
WebPCSuite	3380	2764	2614	19309
XiaoMiWiFi	859	488	604	3343

A. Instrumentation Overhead

To evaluate the instrumentation overhead, we apply our tool on the experimental apps for method instrumentation with and without recording the parameters' values of each invocation of the methods. The detailed results are shown in Table II. The second column gives the original size of the DEX file (KB). The third and fourth columns show the size of DEX file after instrumentation (KB) and its increase percentage (%) under the 'without recording values' option, while the following two columns give the results under the 'with recording values' option. The last column gives the instrumentation time (second).

TABLE II
OVERHEAD OF INSTRUMENTATION FOR USER-DEVELOPED METHOD

Apps	DEX_o	*without value*		*with value*		T
		DEX_i	O_{dex}	DEX_i	O_{dex}	
Andless	112.2	139.4	24.2	169.9	50.6	0.45
Bluechat	38.3	52.1	36.0	67.5	76.2	0.23
GetBackGPS	104.3	137.1	31.3	163.3	56.3	0.31
Antivirus	14176	14763	4.2	15275	7.8	1.83
Baidu	14586	18749	28.5	22802	56.3	8.77
CheDengWo	1240	1370	10.5	1552	25.2	0.90
FontMaster	932	1015	8.9	1118	20.0	0.69
QiCaiScan	1192	1301	9.1	1413	18.5	1.02
WebPCSuite	2764	2864	3.6	2972	7.5	0.71
XiaoMiWiFi	488	618	26.6	774	58.6	0.92

From the table, we know that our instrumentation brings an acceptable space overhead, which depends on the number of user-written methods, within a few seconds. Since Android apps are always driven by user events and the instrumented instructions are not time-consuming (*e.g.* no loops, no recursions), the time caused by our extra codes in a callback method is much smaller than the interval between user-events. According to our experiments on this issue, there is no significant difference in execution time between the apps with and without our instrumentation.

B. Case Study

In this section, we present a case study to show how our tool instruments an Android app. We instrument for user-developed method on an open-source app `Bluechat` that is a light-weight P2P chat app. First, we put `Bluechat.apk` file into our tool's folder and execute a command as follows.

```
java -jar InsDal.jar -f Bluechat.apk -m -p
```

The above option `f` tells `InsDal` that the target file is `Bluechat.apk` and option `m` tells `InsDal` that we want to instrument logs for methods. The option `p` means the disassembled smali files will be preserved and we are able to check the instrumented instructions. Now we open the class `MainActivity` of the app and a byte-code snippet that contains the instrumented instructions is shown in Fig. 4. The code in dash line is the instructions for printing log.

```
.method public constructor <init>()V
   # InsDal Modified --> local register: (0)#
   .registers 4

   move-object/16 v0, p0

   .prologue

   const-string v1, "M_InsDal"
   const-string v2, "Lcom/alexkang/bluechat/MainActivity;--><init>()V"
   invoke-static/range {v1 .. v2}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I

   .line 24
   invoke-direct {v0}, Landroid/app/Activity;-><init>()V

   return-void
.end method
```

Fig. 4. Code snippet of Bluechat

After the instrumentation process, a new instrumented apk file `Bluechat_ins.apk` is generated. Now we can install this modified app and observe the printed log information. To eliminate the useless logs generated by system, we set a filter so that only instrumented logs will be displayed. Then we run the app and useful logs are printed as in Fig. 5. From the printed logs, we can know that there are 7 methods invoked when the app is launched.

```
D/M_InsDal: Lcom/alexkang/bluechat/MainActivity;--><clinit>()V
D/M_InsDal: Lcom/alexkang/bluechat/MainActivity;--><init>()V
D/M_InsDal: Lcom/alexkang/bluechat/MainActivity;-->onCreate(Landroid/os/Bundle;)V
D/M_InsDal: Lcom/alexkang/bluechat/MainActivity$1;--><init>(Lcom/alexkang/bluechat/MainActivity;)V
D/M_InsDal: Lcom/alexkang/bluechat/MainActivity$2;--><init>(Lcom/alexkang/bluechat/MainActivity;)V
D/M_InsDal: Lcom/alexkang/bluechat/MainActivity;-->onResume()V
D/M_InsDal: Lcom/alexkang/bluechat/MainActivity;-->onCreateOptionsMenu(Landroid/view/Menu;)Z
```

Fig. 5. Logs for Bluechat

V. APPLICATION SCENARIO

In this section, we explain that our tool can effectively assist program analysis for Android apps in various application scenarios. According to the analysis purposes, we categorize the application scenarios into two kinds, code coverage measurement and execution trace recording. We will introduce the details of these two kinds of practical application scenarios of our tool in the following subsections.

A. Code Coverage Measurement

Code coverage is a factor that indicates whether the code snippets (statements, methods, classes, etc.) are executed under a given set of inputs. It is often used to measure the quality of test cases in the software testing process. Since the methods can be recognized in both source and byte code, we can

employ our tool to insert the instructions into each method for method-level code coverage analysis. We have applied our tool in two different application scenarios, which will be introduced as follows.

`AppTag` [10] is a target directed automatic test sequence generation tool for Android apps with consideration of the view status information as well as the back stack. They evaluate their tool with some commercial apps, whose source code is not available. Thus the existing Java instrumentation tools are not applicable in this scenario. To evaluate the quality of test cases generated by `AppTag`, Yan *et al.* make use of `InsDal` to calculate method-level coverage. After the test cases are executed, they compute the method coverage through an off-line analysis on the log file.

Battery shortage is a headache problem in mobile device and the bad quality of Android apps is one of the main reasons. Energy analysis can help us to understand the energy consumption for an app in detail. Lu *et al.* [6] proposed an approach to estimate the average energy consumption for each method. They also evaluate their approach with some commercial apps, and leverage `InsDal` to instrument the Android apps for recording the invocation times of each method. Then they execute the instrumented apps under some event sequences and measure the energy consumption for each event sequence by an energy profiling software. Finally they propose an algorithm based on linear regression to estimate the energy consumption of each method. Their experimental results show that only about 2% instructions are inserted into the original apps by our tool `InsDal`.

B. Execution Trace Recording

Execution trace recording is a widely adopted technique for program analysis, especially for dynamic analysis. In general, it is performed by inserting probes into the target programs to record the execution information about the entities (like the values of some variables, the invocation of specific APIs) that we are interested in. Currently our tool supports the byte-code instrumentation for the code snippets involving the given keywords in the app, which can be used to monitor the user-defined features of the runtime information of specific program elements. These features can be used for further analysis with the help of debugging and mining techniques.

We have applied our tool to assist the bug confirmation for a static analyzer that can detect the resource leaks in Android apps. Resource leak is a common bug, which is caused by missing release operations of the resources that require programmers to explicitly release them. Wu *et al.* [8] implemented a flow-sensitive tool `Relda2` to detect resource leaks in Android apps. Since the resource leaks can not be easily observed from the GUI behavior, they leverage `InsDal` to eliminate the false positives in bug reports. Specifically, they randomly execute the instrumented apps, trace all the executed resource operations at runtime, and figure out the mismatch of resource operations. With the help of `InsDal`, 67 out of 121 reported leaks have been confirmed.

VI. Related Work

A number of approaches have been proposed to perform program instrumentation on Android apps for assisting program analysis with different purposes. Arzt *et al.* [2] propose an approach to instrument Java and Android programs with the help of some tools for Java programs, including AspectJ, Tracematches, and Soot, which either require the source code or translate the Dalvik byte-code to the Jimple byte-code. Li [4] leverages code instrumentation technique on top of Soot to resolve two challenges, inter-component communication and reflection, for precise static analysis of Android apps.

Hay *et al.* [3] make use of the instrumentation method to recover the custom fields (variables) of Intent. Xu *et al.* [9] propose a framework based on the instrumentation technique to profile the execution information, including Binder transactions and system calls. However, these works do not systematically discuss the instrumentation problem (the challenges) of Dalvik byte-code. In addition, the above works only focus on some specific targets, *i.e.*, they do not provide a general framework that can be easily extended for different user requirements as we highlight in this paper.

VII. Conclusion and Future Work

This paper describes a safe and extensible instrumentation tool `InsDal` on Dalvik byte-code for Android applications. It mainly supports two instrumentation requirements: instrument for user-developed methods and given keywords. `InsDal` has been applied to several analyses, such as energy analysis, code coverage analysis and resource leak analysis.

There are some future works to improve the functionality and performance of `InsDal`. On one hand, we will support more application scenarios, such as the branch coverage measurement and regular expression based keywords. On the other hand, we will adopt some program analysis techniques, such as Wu *et al.*'s work [7], to reduce the number of instrumentation points.

References

[1] Apktool - for reverse engineering. http://ibotpeaches.github.io/Apktool/.

[2] S. Arzt, S. Rasthofer, and E. Bodden. Instrumenting Android and Java applications as easy as abc. In *RV*, pages 364–381, 2013.

[3] R. Hay, O. Tripp, and M. Pistoia. Dynamic detection of inter-application communication vulnerabilities in Android. In *ISSTA*, pages 118–128, 2015.

[4] L. Li. Boosting static analysis of Android apps through code instrumentation. In *ICSE*, pages 819–822, 2016.

[5] N. Li, X. Meng, J. Offutt, and L. Deng. Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools. In *ISSRE*, pages 380–389, 2013.

[6] Q. Lu, T. Wu, J. Yan, J. Yan, F. Ma, and F. Zhang. Lightweight method-level energy consumption estimation for Android applications. In *TASE*, pages 144–151, 2016.

[7] R. Wu, X. Xiao, S. Cheung, H. Zhang, and C. Zhang. Casper: an efficient approach to call trace collection. In *POPL*, pages 678–690, 2016.

[8] T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang. Light-weight, inter-procedural and callback-aware resource leak detection for Android apps. *TSE*, 42(11):1054–1076, 2016.

[9] H. Xu, Y. Zhou, C. Gao, Y. Kang, and M. R. Lyu. SpyAware: Investigating the privacy leakage signatures in app execution traces. In *ISSRE*, pages 348–358, 2015.

[10] J. Yan, T. Wu, J. Yan, and J. Zhang. Target directed event sequence generation for Android applications. *CoRR abs/1607.03258*, 2016.