Whole Program Analysis of Android Apps: An Anatomy of Break Points in Call Graph

A thesis submitted for the degree
Bachelor of Advanced Computing (Honours)

24 pt Honours project, S2/S1 2023–2024

By:
Zezhou Wang

Supervisors:
Dr. Xiaoyu Sun

Australian National University

School of Computing
College of Engineering, Computing and Cybernetics (CECC)
The Australian National University

May 2024

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the University Academic Misconduct Rules;
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or Wattle site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

May, Zezhou Wang

Acknowledgements

The author, Zezhou Wang, gratefully acknowledge the support of the supervisor Dr. Xiaoyu Sun and the fellow student, Yitong Wang.

Abstract

Software is a program installed on electronic devices, designed to perform specific tasks, which is an indispensable component of today's information technology era. Android is a widely used mobile device operating system, with its importance being self-evident. With the rapid development and widespread usage of the Android platform, it is facing unprecedented threats to software security. Facing the continuously newly emerged malware and security vulnerabilities, program analysis is a crucial step in ensuring software quality. Program analysis is a technique for understanding and evaluating various perspectives of software performance, which is of vital importance in the field of Android software analysis. The Call Graph (CG) is a significant tool in the process of program analysis. The Missing method invocation relationships in the call graph will lead to the Break Points (BPs) in the call graph, which negatively impact the program analysis process. The thesis proposed a novel framework, BPFinder, for detecting BPs in the call graph of Android apps, based on the classic program analysis framework Soot. And the thesis evaluated BPFinder on the DroidBench benchmark suite and 22,000 Android apps developed in 2020 and later obtained from Google Play. BPFinder achieved a 91.21% accuracy and a 88.89% F1-score rate on the DroidBench and summarized six root causes of BPs in CG, also effectively detected the BPs in the real-world Android apps. The experiment as well as the proposed BPFinder framework provide ideas for the development of Android program analysis and the improvement of the analysis framework, which are of practical significance for Android software analysis.

Keywords: Android Application, Software Security, Program Analysis, Call Graph, Break Point

Table of Contents

Chapter 1

Background

In this chapter, the research scope and fundamental concepts for readers to understand the work in this thesis are provided. The topic of the thesis is the whole program analysis of Android apps, the technical terms in the thesis are restricted to Android software and Java programming language unless otherwise specified.

1.1 Program Analysis

Program analysis refers to the process of understanding and evaluating the overall performance of Android applications by specific techniques and methodologies. Currently, program analysis plays an important role in many aspects, including software development, testing, and security study.

1.2 Call Graph and Break Point

Call graph is proposed in the 1970s for the first time for the need of C++ program analysis (Ryder, 1979), which describes the calling relationships between different methods within a program. After decades of development, call graph has become a significant tool in the context of inspecting and analyzing Android applications.

Break Point (BP) refers to the missing calling relationships between any relevant methods or the specific method that is either entirely missing in the CG or lacking active method body. The occurrence of break points may indicate the potential vulnerabilities within a program. Therefore, the break point in the CG is worthy of analysis as well as attention.

1.3 Static Analysis, Dynamic Analysis, and Hybrid Approach

Static and dynamic analysis are basic approaches of program analysis. Static analysis detects potential issues by analyzing the program's source code, bytecode, and intermediate representation (IR), is a process of analyzing the program itself without dynamically executing it. Conversely, dynamic analysis analyzes the triggered program's behaviors by executing the program. Dynamic analysis identifies the vulnerabilities within a program by monitoring the process of executing as well as collecting targeted outputs and is able to find run-time behaviors of a program which cannot be detected by static analysis.

Hybrid approach combines different analysis approaches, commonly composed of static and dynamic analysis (Artho and Biere, 2005). Different analysis approaches can work together and compensate for shortcomings, potentially achieving improved analysis results (Artho and Biere, 2005).

1.4 Soot Framework

Soot (Vallée-Rai et al., 2010) is a well-known program analysis framework for Java bytecode analysis and optimization. Many works have chosen to build their own frameworks or tools on top of Soot.

Chapter 2

Introduction

In the information era, mobile devices are indispensable for daily life. Android is a widely used mobile device operating system and software development platform, with its significance being self-evident. According to the Enterprise Apps Today website (EnterpriseAppsToday, 2024), as of early 2024, Android holds a substantial lead with a market share of approximately 71% across many different regions and groups around the world. However, with the rapid development and wide spread of Android, it is facing unprecedented threats. For example, different versions of Android software simultaneously present across various types of mobile devices, making the compatibility of Android and its component an increasingly serious problem (Cai et al., 2019). Besides, the open-source nature of the Android system attracts many malware developers. Nowadays, improving software quality and protecting users' privacy security are urgent needs. Program analysis is an essential step for ensuring software quality and security facing the continuously newly emerged malware and security vulnerabilities.

In this process, call graph is a powerful tool that can provide extra information for detection. Many works on Android vulnerability detection are based on the call graph (CG) (Mumtaz and El-Alfy, 2017). The break points in the call graph can lead to deviations between the constructed CG and the actual situations, thereby becoming the key factor that influence the experiment results. Therefore, the comprehensive research on break points can facilitate developers to refine the program analysis and construction process of the CG. To the best of the available knowledge, none of the previous research proposed a single tool to systemically detect break points in the program CG and explain the reasons of CG's unsoundness. The thesis is aimed to conduct the thorough research on the break points and proposed a novel framework towards both academia and industry.

The thesis proposed the BPFinder, based on Soot framework (Vallée-Rai et al., 2010), to systematically detect the break points in the call graph and made the following contributions:

- The thesis proposed an open-source tool based on Java, BPFinder, targeting at solving current gaps in the CG's unsoundness in the perspective of detecting break points in the call graph.
- To the best of the available knowledge, BPFinder is the first single tool that systemically detects break points in the CG.
- BPFinder innovatively employed the jimple-based static analysis approach and instrumentation-based dynamic analysis approach to conduct program analysis on detecting break points.
- Based on experiments of BPFinder, the thesis summarized the root causes for break points and investigated the break points in the real-world Android applications.

The rest of the thesis is organized by following chapters. Chapter 2: Related Work provides related research about the construction and application of the CG, some root causes of CG's unsoundness, and the analysis approach. Chapter 3: Motivation demonstrates the motivation for developing BPFinder and experiments. Chapter 4: Methodology elucidates the workflow of BPFinder and the algorithms involved. Chapter 5: Evaluation showcases the detailed experimental results in the form of two research questions (RQs). Chapter 6: Discussion explains the experiment results, inspiration for future work, and limitations of this work. And eventually, Chapter 7: Conclusion summarizes the work of this thesis.

Chapter 3

Motivation

In the literature, Reif et al. conducted the systematic evaluation of unsoundness of Android CG construction algorithms (Reif et al., 2018). In their research, Reif discovered that the generated call graphs are always missing some important call edges due to the different Java and SDK versions, deficiency of construction algorithms, and some special Android APIs. This overall summarization only provides limited information on the detailed reasons for the unsoundness of the CG and how to optimize the construction process of it. After inspiring by the aforementioned work, the thesis decided to delve into the CG of Android apps. And the concept of the unsoundness has been refined into break points (BPs) in the CG in the context of this thesis.

Consider the scenario of an application within the DroidBench benchmark suite (Arzt et al., 2014b). The SinkInNativeCode benchmark app is known to be a representative case of malicious implementation of Android native code (native interface) mechanism. Listing 3.1 and 3.2 below showcase the code snippets from SinkInNativeCode.apk, containing the complete usage of the native code mechanism. In Listing 3.1, the native function *cFuncSendSMS()* is firstly defined (line 7) and is invoked later (line 19). The unique IMEI information is obtained (line 18) and passed as the parameter to *cFuncSendSMS()* function (line 19). Listing 3.2 shows the code snippet in the *cFuncSendSMS()* function within the ndkmod.cpp file, which is compiled of C/C++. In this C++ function, *CallVoidMethod()* function is invoked to send the sensitive information IMEI to the unknown device by *sendTextMessage()* function (line 6). Since that the actual data leakage happens within the native function, many traditional static analysis frameworks improperly recognized this case as false negative, such as FlowDroid (Arzt et al., 2014a).

```
1  public class ActMain extends Activity {
2          static {
3                  System.loadLibrary("ndkmod");
4          }
5
6          // --------------------
7          public native boolean cFuncSendSMS(String message); // Source in
               C code
8          // --------------------
9
10         @Override
11         public void onCreate(Bundle savedInstanceState) {
12                 super.onCreate(savedInstanceState);
13                 ......
14                 button.setOnClickListener(new OnClickListener() {
15                         @Override
16                         public void onClick(View arg0) {
17                                 TelephonyManager telephonyManager = (
                                       TelephonyManager) getSystemService(
                                       Context.TELEPHONY_SERVICE);
18                                 String strIMEI = telephonyManager.
                                       getDeviceId();
19                                 Toast.makeText(ActMain.this, "ok:" +
                                       cFuncSendSMS(strIMEI), Toast.
                                       LENGTH_LONG).show();
20                         }
21                 });
22         }
23 }
```

Listing 3.1: **Code excerpt of src.mod.ndk.ActMain.java from DroidBench's SinkInNativeCode.apk.** Reminder: although the analysis in the thesis operates on the bytecode level of Android apps, Java (or C/C++) code snippets are provided as examples to enhance readability.

```
1  // Construct the target phone number
2          jstring targetPhoneNo = env->NewStringUTF("+4923243234423");
3          ......
4          if (sendTextMessage == NULL)
5                  return false;
6          env->CallVoidMethod(smsManager, sendTextMessage, targetPhoneNo,
               NULL, message, NULL, NULL);
```

Listing 3.2: **Code snippet excerpt of ndkmod.cpp from DroidBench's SinkInNativeCode.apk.**

The methods within the native interface are actually executed methods but will not be captured by the Android CG algorithms, thereby caused the break point in the CG. This is one cause of break points in the CG. The special circumstances which lead to break points in the Android CG should be comprehensively understood, evaluated, and monitored.

Chapter 4

Related Work

4.1 Call Graph Construction Algorithms

Call graph has various construction algorithms in the literature. Among them, Reachability Analysis (RA) (Nikolić and Spoto, 2014), Class Hierarchy Analysis (CHA) (Dean et al., 1995), and Rapid Type Analysis (RTA) (Bacon and Sweeney, 1996) are widely accepted algorithms for call graph construction. RA is the most fundamental construction algorithm, which only count the signatures of methods. For object-oriented programming language like Java, RA is not accurate since it ignores the class hierarchy of methods (Ismail, 2009). CHA takes hierarchy into consideration and includes all the methods from instantiated sub-classes of current methods, improving the accuracy significantly. RTA detects the possible implemented virtual functions in the context of all the instantiated classes, which is an efficient and direct CG construction approach. Additionally, numerous novel approaches have been proposed for CG construction, such as the cocktail call graph construction proposed by Cai and Zhang (2023), the abstract syntax tree (AST)-based approach proposed by Bedadala et al. (2020), and the scalable propagation-based approach proposed by Tip and Palsberg (2000).

In the thesis, SPARK pointer analysis algorithm (Whaley, 2007) is employed for constructing the CG. SPARK algorithm achieves the data flow-sensitive and program context-sensitive analysis by using pointers that point to the same memory space. Generally, SPARK algorithm has outstanding construction precision.

4.2 The Applications of Call Graph

There are wide applications of CG in Android program analysis and malware detection. The structural similarities of CGs are employed to distinguish potential obfuscated malwares from benign software (Ikram et al., 2019). Graph isomorphism can be implemented

to measure the similarities of the program (Bai et al., 2019). Machine learning techniques can also be utilized to extract syntactic and semantic information from CGs. In the literature, Graph Neural Networks (GNNs) and Convolutional Neural Networks (CNNs) have been successfully applied in this field (Feng et al., 2020; Vinayaka and Jaidhar, 2021). The static analysis can also be implemented, Samhi et al. (2022) proposed the merged call graphs within their JuCify framework, facilitating the static analysis to reach more functions/methods during analysis.

4.3 Root Cause 1: Native Code

Some kinds of BPs are already investigated in the previous research. Native code is a basic reason which allowing developers to compile in other programming languages such as C/C++ and use in the Android development environment (Lin et al., 2011). According to a recent research (Ruggia et al., 2023), native code and functions are always employed to conceal malicious behaviors and the analysis of native code needs extra algorithms. The literature has developed some approaches for detecting native code, like binary scanning approach (Fourtounis et al., 2020) and aforementioned merged CG approach.

4.4 Root Cause 2: Reflection Call

Reflection call is another cause of the CG's unsoundness (Reif et al., 2018). Reflection is of vital significance in private API calling, maintaining compatibility of different Java versions, supporting external library and frameworks, and protecting developer's intellectual property (Sun et al., 2021). Like native code, reflection can be exploited to hide the malicious behaviors or even transfer the malicious code (Sun et al., 2021). Many previous frameworks can detect reflection calls within Android applications. For instance, the static analysis framework, FlowDroid (Arzt et al., 2014a), can partially detect reflections calls; the DroidRA framework (Li et al., 2016; Sun et al., 2021) implemented the instrumentation-based approach to detect reflection calls. Besides, there are also other reasons that might attribute to the BPs in the CG, such as Dynamic Code Loading (DCL) (Qu et al., 2017).

4.5 Analysis Approach

According to the systematic literature review of Android static analysis (Li et al., 2017), despite that the static analysis is less accurate than deep neural network-based analysis approach, it is effective for analysis and suitable for large-scale experiment. And the dynamic analysis approach is especially effective for vetting obfuscated code in the program, but much more computational expensive (Yang et al., 2018).

Therefore, the thesis combined the static and dynamic analysis to detect break points. The most similar analysis approach recently was used by Amin et al. (2019) within their

AndroShield framework. AndroShield implemented the static analysis approach to check vulnerabilities in the bytecode level and implemented the dynamic analysis approach to detect malicious run-time behaviors. The thesis employed the static analysis approach to compare the differences of jimple intermediate representation and methods in the call graphs. Meanwhile, instrumentation-based dynamic analysis is integrated to detect the behaviors triggered during the process of executing the program, like reflection calls and dynamic code loading.

4.6 The Instrumentation Technique

The literature has already adopted code instrumentation as a mature technique in the dynamic analysis (Sun et al., 2021). Instrumentation techniques can acquire the desired output information by inserting the specific code into the particular location within a program. The aforementioned DroidRA framework adopted the instrumentation-based method to reduce the complexity of composite data flow involved reflection calls. Sahin et al. (2019) implemented run-time instrumentation within their RANDR framework, enabling the record and replay of dynamic behaviors. And Li (2016) also used code instrumentation approach to enhance the static analysis process.

Chapter 5

Methodology

5.1 Overview

The main objective of this thesis is to design a new framework named BPFinder to comprehensively detect all kinds of break points in the call graph of Android applications. BPFinder combines the static and dynamic analysis: the comprehensiveness and efficiency of static analysis compensate for the low code coverage of dynamic analysis while dynamic analysis's ability to detect run-time behaviors also covers the situations that cannot be detected by static analysis. For example, the dynamically invoked methods through reflection operations from java.lang.reflect package can be detected and monitored by dynamic approach, but cannot be identified by static analysis frameworks like FlowDroid (Arzt et al., 2014a). Figure 5.1 provides the overview of the BPFinder framework. To be specific, BPFinder firstly takes the Android application package (APK) as input. The following workflow of BPFinder can be divided into two separate parts: 1) static analysis obtains all jimple statements from the input APK and compares the method calls with the call graph; 2) dynamic analysis executes the instrumented APK on Android emulators or physical devices, collecting the method invocation sequence for further comparison with the call graph.

5.2 Jimple-based Static Analysis

Jimple is an IR within the Soot framework (Vallée-Rai et al., 2010), serves as a media for understanding, analyzing, and updating Java bytecode. Jimple is a 3 Address Code (3AC), which simplifies the expression of the calling statement and clearly shows the method signature, assignment process, and the calling relationship. The use of jimple enables Soot framework to effectively analyze every method. In the static analysis component of BPFinder, jimple serves as the pivotal cornerstone to scrutinize the bytecode of Android apps.

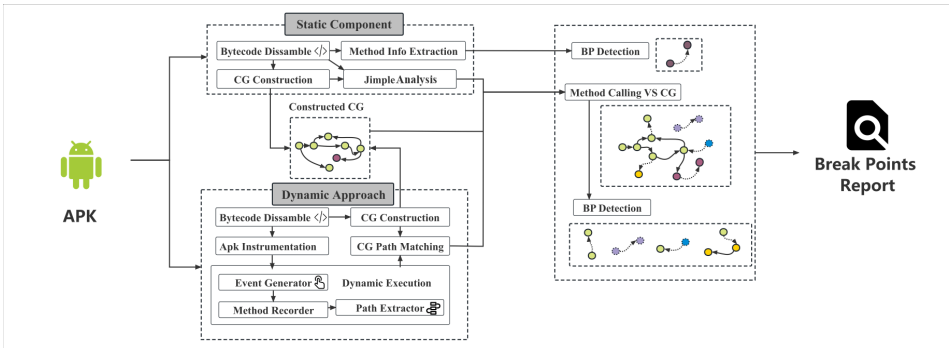

Figure 5.1: The Overview of the Proposed

BPFinder loads the call graph as the first step. The call graph contains the detailed roadmap of method invocations and dependencies within an application. The Chapter 4 provides the sample applications of the call graph.

Subsequently, BPFinder thoroughly traverses every method invocation statement from jimple IR. For each method invocation jimple statement, BPFinder systematically cross-references and compares it against the call graph, identifying the potential break points (BPs) based on code pattern recognition. This exhaustive traverse and comparison process scrutinizes the nuances of method invocations in the jimple files and the generated call graph, enabling the comprehensiveness of detection.

The pseudocode shown in Algorithm 1 outlines the crucial steps of this jimple-based static analysis, providing a structured pipeline for this process. This algorithm demonstrates the systematic approach composed of two steps: 1) call graph loading and construction; 2) jimple scanning and comparison.

Algorithm 1: Static Analysis of BPFinder

Data: The Testing APK File *apk*
Result: Break point(s) *bp*
1 $cg \leftarrow$ loadCallGraphBySoot(*apk*);
2 $bp \leftarrow$ scanJimpleAndCompare(*apk*,*cg*);
3 **return** *bp*

In the first step, BPFinder loads and constructs the call graph by SPARK algorithm. Then, the strategy and detailed implementation of how to conduct jimple scanning and comparison can be viewed in Algorithm 2. During this procedure, only the nodes in the target packages are preserved in the call graph. Utilizing the APIs from the Soot framework, BPFinder sequentially analyzes class files in the APK, converts them into the jimple representation, and obtains the caller and callee methods from the statements. Then, BPFinder checks whether each statement has the active body or whether it is a part of an edge within the call graph. If a statement does not meet these criteria, BPFinder will flag and report it as a case of BP.

Algorithm 2: scanJimpleAndCompare Step

Data: The Testing APK File *apk*, Constructed Call graph *cg*
Result: Break points *bp*

```
bp ← ∅;
Jimples ← soot.getBody().getUnits(apk);
for j ∈ Jimples do
    if j does not belong to target packages or j not containsInvokeExpr() then
        continue;
    end
    s ← j.caller; t ← j.callee;
    if t not hasActiveBody() then
        p ← noActiveBody();
    end
    else if s ∉ cg or t ∉ cg or edge(s, t) ∉ cg then
        p ← inCompleteCG();
    end
    bp.add(p);
end
return bp
```

5.3 Instrumentation-based Dynamic Analysis

Unlike the Static analysis which enables BPFinder to dissect the source code of an APK without executing it, the dynamic component of BPFinder runs instrumented APKs on emulators or physical devices to monitor and collect their run-time behaviors. In the Android environment, the execution of Java methods is relied on Java call stack, a stack-based structure of storing method invocation sequence. The instrumented APKs can invoke this stack and dynamically print the method information through Log function, which is then monitored by the process listener. Through meticulous code instrumentation of the APK, BPFinder transforms the packaged Android applications into dynamically observable program entities, laying a solid foundation for the detection of the APK's run-time behaviors.

For an APK, the specific execution process within BPFinder is as follows:

- Android Debug Bridge (adb) connects the host machine with an Android emulator or a physical device. Meanwhile, BPFinder employed Android Asset Packaging Tool (aapt) to parse APK and obtain the packages and corresponding names for the following pipeline.
- The process listener, logcat, is then initialized to trace the messages thrown by the instrumented APK. If the log contains log information of previous experiments, all the previous content will be cleared.

- Implementing the adb push command to install the instrumented APK of an application on the Android emulator/physical device.
- Maxim (Zhang, 2024), an Android Monkey Tester compatible with both emulators and physical devices, is activated for the interaction with the application. Maxim initiates adequate amount of random pseudo-user behaviors on the screen to simulate the real interaction with the application. BPFinder adopts **uiautomatormix** strategy (AndroidDevelopers, 2023b) to set the Maxim tool.
- During the process of interacting, the emulator continuously generates and throws log information containing the indicative information of different methods within the app.
- After execution, BPFinder is going to compare and match the acquired methods with the methods in the call graph. At the same time, BPFinder collects the unique method name and calculates the code coverage rate.

By capturing the run-time behaviors of the APK and comparing the call graph, the dynamic execution process can detect potential BPs, complementing the work of the static analysis component.

Chapter 6

Evaluation

Research Questions. To evaluate the effectiveness and efficiency of BPFinder, two research questions (RQs) have been designed and investigated.

RQ1: Is BPFinder effective in detecting break points in the call graph? The thesis investigates the performance of BPFinder on testing suite DroidBench (Arzt et al., 2014b).

RQ2: What are root causes of break points in the real-world applications? The thesis conducts the large-scale experiment based on the static component of BPFinder and conducts the case study to explain in detail.

The experiment environment was Soot version 4.1.1 (Vallée-Rai et al., 2010), Android Studio version 2022.3.1 (AndroidDevelopers, 2023a), Nox Android emulator version 7.0.5.8 (NoxLimited, 2023) and Android emulator 9.0 Google ARM64-V8a API 28 (AndroidDevelopers, 2024), Maxim tool version 1.0.21.0612 (Zhang, 2024). As for algorithms, **SPARK algorithm** (Whaley, 2007) was employed for CG construction and **uiautomatormix** strategy (AndroidDevelopers, 2023b) was employed for Maxim Initialization. For source code inspection of the real-world applications, JADX tool version 1.4 (Skylot, 2024) was utilized. The benchmark experiment was conducted on a 16-inch MacBook Pro equipped with the Apple M1 chip and the large-scale experiment was conducted on the High-Performance Computer (HPC) from National Computational Infrastructure (NCI, Australia).

6.1 RQ1: Benchmark Experiment

Experiment Settings. The purpose of RQ1 was to validate the effectiveness of BPFinder in the well-known DroidBench benchmark suite. For this experiment, the DroidBench 3.0 version develop repository Arzt et al. (2014b) was selected. The introduce of the DroidBench provided comprehensive and diverse APK testing benchmark

data, including source code of native code, reflection call, callback methods, etc. All experimental results were meticulously classified as true positive (TP), false positive (FP), true negative (TN), or false negative (FN) and their quantities were recorded. These metrics were further used to calculate the accuracy, precision, recall, and F1-score of BPFinder, aiming to ensure the objectivity, reliability, and stability of the evaluation. For every testing APK file, the experiment 1) applied BPFinder to detect break points statically and dynamically, 2) verified the source code of the APK to ensure the correctness of the detected results.

Root Causes of BP: After summarization, BPFinder detected 6 root causes of the BP from DroidBench benchmark suite in total. Table 6.1 demonstrates the patterns, statistics, and corresponding behaviors of these root causes. Among these 6 causes, reflection call and native code have already undergone relatively thorough research and evaluation. #1 Incomplete CG is caused by partially failed analysis of SPARK algorithm and may lead to the ignorance of some essential information during analyzing. Besides, the ignorance of abstract/interface method may also result in missing critical information about the program. Dynamic loading can also cause BPs in the CG and has the similar pattern with reflection call.

It is worth noticing that BPFinder fully utilized static and dynamic analysis and took advantage of them in detecting BPs. Static analysis is effective in detecting problems like #1 incomplete CG, #2 native code, and #3 abstract methods. Its advantage lies in its capacity of detecting BPs without executing the program. Dynamic analysis is more appropriate to be applied for detecting BPs related to run-time only behaviors such as #4 reflection call and #6 dynamic loading.

Table 6.1: BP cases and corresponding behaviors in DroidBench

No.	BP Root Causes	BPDetector		BP Explanation	Behavior	DroidBench Apks
		Static	Dynamic			
#1	Incomplete CG	✓	×	CG in Soot is incomplete after SPARK analysis		24
#2	Native Code	✓	✓	Java native interface implemented by C++	Methods with native signature	9
#3	Abstract/Interface Methods	✓	×	Abstract method in super class or interface		11
#4	Reflection Call	×	✓	Reflection operation using java.lang.reflect	java.lang.Class/newInstance java.reflect.Constructor/newInstance java.reflect.Method/invoke	18
#5	Callback Methods	×	✓	Method called in certain Android library	android.os.Parcel/writeValue android.app.Activity/setContentView android.app.Activity/onCreate android.app.Activity/onPause android.app.Activity/onStart android.content.ServiceConnection/onServiceConnected android.os.Handler/dispatchMessage	12
#6	Dynamic Loading	×	✓	Loading code dynamically during execution	java.lang.ClassLoader/loadClass dalvik.system.PathClassLoader/loadClass dalvik.system.DexClassLoader/loadClass	7

Evaluation Metrics and Results: The experiment identified TP, FP, TN, and FN by following definitions:

- True Positive (TP): BPFinder successfully reports BP(s) out of the APK which actually has BP(s)

- True Negative (TN): BPFinder doesn't report BP(s) out of the APK which doesn't have BP(s)
- False Positive (FP): BPFinder wrongly reports BP(s) out of the APK which doesn't have BP(s)
- False Negative (FN): BPFinder is failed to report BP(s) out of the APK which actually has BP(s)

And four metrics were defined as:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{6.1}$$

$$\text{Precision} = \frac{TP}{TP + FP} \tag{6.2}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{6.3}$$

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{6.4}$$

Among the 190 benchmark apps in the develop repository, 8 of them were failed to be analyzed by BPFinder, and the typical reasons for the failures includes low SDK versions and bugs within the program, which are basically inherent reasons of the APK files themselves. Therefore, in this experiment, the thesis reported the data derived from the remaining 182 apps. Overall, BPFinder detected 80 APKs whose call graph contains break points out of these 182 apps, producing the average F1-score value of 88.89% and the average accuracy value of 91.21% across 6 categories.

Table 6.2 shows the detailed efficiency of BPFinder on detecting BPs with different root causes. It is observable that BPFinder achieved 100% precision for all kinds of BPs in the call graph, indicating the remarkable ability of BPFinder in detecting BPs. For static analysis component, the F1-scores on 3 different root causes (#1, #2, and #3) reached 85.71%, 100%, and 100% respectively, indicating that BPFinder has relatively balanced performance across different categories. For dynamic analysis part, the F1-scores on 4 different root causes (#2, #4, #5, and #6) only reached 66.67%, 84.85%, 66.67%, and 100%. The reason for the decreased F1-score values may attribute to the random process of Maxim and the limited code coverage rate. Besides, in the dynamic process, the activation of native code and callback method during run-time always require complicated execution and behaviors, which is sometimes not supported by Maxim.

Table 6.2: Metrics on APKs with different root causes

Metrics	BPFinder-Static			BPFinder-Dynamic				
No.	#1	#2	#3	#2	#4	#5	#6	Sum
TP	18	9	11	2	14	12	3	64
TN	166	181	179	181	171	164	183	102
FP	-	-	-	-	-	-	-	-
FN	6	-	-	2	5	12	-	16
Failed	-	-	-	5	-	2	4	8
Accuracy(%)	96.84	100	100	98.92	97.37	93.62	100	91.21
Precision(%)	100	100	100	100	100	100	100	100
Recall(%)	75	100	100	50	73.68	50	100	80
F1-score(%)	85.71	100	100	66.67	84.85	66.67	100	88.89

To answer RQ1, the aforementioned experimental outcomes have proved the effectiveness and efficiency of BPFinder in the perspective of detecting break points in the Android call graph.

6.2 RQ2: Large-scale Experiment and Case Study

Experiment Settings. The purpose of RQ2 is to further validate the effectiveness of BPFinder and investigate the break points in the real-world Android applications. The experiment dataset is 22,000 Android apps developed since 2020 and onwards, collected from Google Play (Google LLC, 2024), with equal amount of benign software (goodware) and malicious software (malware). The benign and malicious categories were tested independently and subsequently compared for analysis. Dynamic analysis requires extensive computational resources and time, which is extremely ineffective for the large-scale experiment and real-world Android apps. Therefore, only the static component of BPFinder was evaluated. For testing APK files, the experiment 1) applied BPFinder to detect break points statically and collected the break points reports, 2) counted the number of the APKs detected whose call graph contains the break points for both categories, 3) quantified the amount of native code and abstract/interface methods case in the detected APK. Additionally, the experiment explored and explained the break points in the call graph of real-world Android apps by the case study.

Experimental Results: The results are shown in the Table 6.3 below: for benign software, BPFinder detected 272 instances of APKs containing break points, while for malicious software, BPFinder detected 1182 instances. In the 272 benign instances, there are 471 cases of native code usage and 18276 cases of abstract/interface method usage. On the other hand, within the 1182 malicious instances, there are 4415 cases of native code usage and 91240 cases of abstract/interface method usage.

Table 6.3: Results of the Large-scale Experiment

Category	Benign Software	Malicious Software
Detected APKs	272	1182
Native Code Cases	471	4415
Abstract/Interface Method Cases	18276	91240

Explanation. The experimental results indicate the genuine presence of break points in the deployed Android applications. Within the APK files of real-world Android software, whether benign or malicious, the utilization of abstract/interface methods and native code is prevalent, serving as the primary reasons for break points in the real-world situations. These results are in line with general expectations, as in the industry, the utilization of native code and C/C++ code can enhance the program execution efficiency, while the usage of abstract/interface methods is also inevitable for maintaining large-scale and overly complicated Android software systems. Besides, in the experiment, the amount of APKs containing break points in the malicious software is over four times greater than that in the benign software. This may indicate that the malicious code is more likely to cause the break points in the call graph.

Case Study: To better demonstrate the break points in the call graph of real-world Android applications, the APK of a mobile game developed by the Chinese company NetEase, **Infinite Lagrange**, is investigated and presented as the case study in the thesis. This APK was among the 22,000 APKs collected from Google Play (Google LLC, 2024). There are a significant amount of break points in this game's call graph. The Listing 6.1 and 6.2 below showcases the implementation process of the Client class, which is used for user login and related operations.

```
1  public class Client extends NeoXClient {
2      private static final int CAMERA_REQUEST_CODE = 99;
3      static final int MBB_ABORT = 5;
4      ......
5  }
6  ......
7  @Override // com.netease.neox.NeoXClient, android.app.NativeActivity,
       android.app.Activity
8      public void onCreate(Bundle bundle) {
9          ......
10         if (hookPackageManagerHelper.m_need_load_new_so) {
11             hookPackageManagerHelper.hookNativeActivityPackage(bundle);
12         } else {
13             NativeInterface.Dummy();
14             super.onCreate(bundle);
15         }
```

Listing 6.1: **Code snippet of Client class excerpt from the APK of Infinite Lagrange.**

```
1 public abstract NeoXClient extends NativeActivity{
2     public static final int BUILDINFO_BOARD = 0;
3     public static final int BUILDINFO_BOOTLOADER = 1;
4     ......
5 }
```

Listing 6.2: **Code snippet of NeoXClient class excerpt from the APK of Infinite Lagrange.**

This case indicates the extremely complex architectures as well as the aliased hierarchies in the deployed Android software systems. The definition of the Client class involves the NeoXClient class and the NativeActivity class. In Android development, NativeActivity is an integration activity class for Android developers to invoke the native code in C/C++. The Client class is established upon multiple inheritance of previous classes (line 1 in Listing 6.1 and 6.2) and the override of previous method (line 7 in Listing 6.1). All these operations contribute to making the data flow within the program increasingly complicated. Besides, as shown in the line 2 and 3 in Listing 6.1 and 6.2, different classes have different fields (and probably different value assignments), which further exacerbated the complexity of data flow analysis and method invocation tracking for the APK file. The failure of tracking data flow and method invocation process finally resulted in the missing edges or methods in the call graph, which are exactly break points.

To answer RQ2, the experiment successfully detected abundant break points in the call graph of real-world Android apps, further validated the effectiveness of BPFinder. Native code and abstract/interface methods, which meet the needs of practical applications of apps, are widely existed in the real-world Android apps.

Chapter 7

Discussion

7.1 Implications and Future Work

Call graph describes the relationships between different method invocations in the program, is a powerful tool for program inspection, analysis, and debugging. The break points in the call graph indicate the deviations between the call graph and the real data flow process within the program, is a significant factor which affect the accuracy of program analysis. The research on break points in the call graph can provide the developers with deeper and unique insights into the call graph and program structure of Android application, and even its development direction and status.

Boost Static Analyzers. The proposed BPFinder framework along with its findings can be viewed as the references to refine and boost the static analysis process. Currently, many kinds of break points pervasively exist in the call graph. Some break points are caused by the deficiencies of the CG construction algorithms, while some stem from the inherent mechanisms of Java as well as Android themselves. BPFinder detected and summarized many different root causes which lead to the break points in the call graph. Fixing these break points is the future direction to improve the CG construction algorithms and keep the CG up-to-date. Another crucial aspect is that BPFinder helps to regularize the legal usage of Android mechanisms and monitor potential malicious behaviors.

Go Beyond Android. This thesis was based on Android applications. However, nowadays, Java applications as well as some hybrid applications based on cross-platform frameworks are becoming increasingly popular (Gok and Khanna, 2013; Kaur and Kaur, 2022). The research of Hui et al. (2013) suggested that the cross-platform techniques play an important role in the overall software environment. Previous research on JavaScript in Android applications also figured out the vulnerabilities of JavaScript in the real-world Android apps (Bai et al., 2018; Song et al., 2018). The findings of BPFinder in

this thesis can not only be applied to Android but also inspire research on program analysis in other programming languages. There are much more to explore in the field of software development and security.

7.2 Limitations

Code obfuscation is a widely applied technique in the real-world Android applications for intellectual property protection and illegal reverse engineering prevention (Kovacheva, 2013). Bacci et al. (2018) demonstrated that the static analysis-based approach is particularly ineffective against the obfuscated malware based on the findings of their experiments. Since the thesis validated BPFinder with 22,000 real-world Android applications from Google Play, the code obfuscation is an unignorable factor in disrupting the static analysis process of BPFinder.

The dynamic approach was based on code instrumentation and the Maxim tester (Zhang, 2024). Like previous framework DroidRA (Li et al., 2016), the thesis expected the code instrumentation method to cover as much code within the program as possible. However, the dynamic experimental results indicated that the code coverage rate fell within the range of 30% to 40%. Despite that this rate is normal and acceptable in Android software testing (Huang et al., 2015), the dynamic analysis approach of BPFinder is powerless for the remaining part of the program.

Besides, within the reachable range of executed code, the occurrence of some break points in real-world apps are the results of combined effects of multiple unknown factors. Although the thesis summarized all the results comprehensively and meticulously, there are still some exceptions that cannot be categorized due to the limited sample, such as the *httpjob* case.

Chapter 8

Conclusion

Call graph is of significant importance in Android software engineering and is worthy of continuous attention. Focused on the break point (BP), the thesis presents a novel perspective of analyzing the call graph (CG) and further extracting meaningful information from it. The open-source Android analysis framework, BPFinder, is proposed for BP detection. BPFinder combines the static analysis and dynamic analysis together, aiming to comprehensively detect BPs in the Android CG. It leverages the jimple intermediate representation within the Soot framework as the foundation of static approach and the code instrumentation technique for the dynamic. BPFinder can effectively detect BPs in the APKs from the benchmark DroidBench, achieving the 91.21% accuracy and 88.89% F1-score. Through the large-scale experiment and a case study, the thesis further validated the performance of BPFinder, providing the significant reference for future research.

Bibliography

Amin, A.; Eldessouki, A.; Magdy, M. T.; Abdeen, N.; Hindy, H.; and Hegazy, I., 2019. Androshield: Automated android applications vulnerability detection, a hybrid static and dynamic analysis approach. *Information*, 10, 10 (2019), 326. [Cited on page 8.]

AndroidDevelopers, 2023a. Android studio. https://developer.android.com/studio. Version 2022.3.1, accessed 2023-09-01. [Cited on page 14.]

AndroidDevelopers, 2023b. Uiautomatormix strategy. https://developer.android.com/training/testing/ui-automator. Accessed: 2023-11-25. [Cited on pages 13 and 14.]

AndroidDevelopers, 2024. Android emulator. https://developer.android.com/studio/run/emulator. Accessed: 2024-03-24. [Cited on page 14.]

Artho, C. and Biere, A., 2005. Combined static and dynamic analysis. *Electronic Notes in Theoretical Computer Science*, 131 (May 2005), 3–14. doi:10.1016/j.entcs.2005.01.018. [Cited on page 2.]

Arzt, S.; Rasthofer, S.; Fritz, C.; Bodden, E.; Bartel, A.; Klein, J.; Le Traon, Y.; Octeau, D.; and McDaniel, P., 2014a. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM sigplan notices*, 49, 6 (2014), 259–269. [Cited on pages 5, 8, and 10.]

Arzt, S.; Rasthofer, S.; Fritz, C.; Bodden, E.; Bartel, A.; Klein, J.; Traon, Y. L.; Octeau, D.; and McDaniel, P., 2014b. Droidbench: A micro-benchmark suite to assess the stability of taint-analysis tools for android. https://github.com/secure-software-engineering/DroidBench/tree/develop. Accessed: 2024-05-23. [Cited on pages 5 and 14.]

Bacci, A.; Bartoli, A.; Martinelli, F.; Medvet, E.; Mercaldo, F.; Visaggio, C. A.; et al., 2018. Impact of code obfuscation on android malware detection based on static and dynamic analysis. In *Proceedings of the 2018 International Conference on Information Systems Security and Privacy (ICISSP)*, 379–385. [Cited on page 21.]

Bacon, D. F. and Sweeney, P. F., 1996. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 324–341. [Cited on page 7.]

Bai, J.; Shi, Q.; and Mu, S., 2019. A malware and variant detection method using function call graph isomorphism. *Security and Communication Networks*, 2019 (2019), 1–12. [Cited on page 8.]

Bai, J.; Wang, W.; Qin, Y.; Zhang, S.; Wang, J.; and Pan, Y., 2018. Bridgetaint: a bi-directional dynamic taint tracking method for javascript bridges in android hybrid applications. *IEEE Transactions on Information Forensics and Security*, 14, 3 (2018), 677–692. [Cited on page 20.]

Bedadala, P.; Manasa, D.; and Nair, L. S., 2020. Generation of call graph for java higher order functions. In *2020 5th International Conference on Communication and Electronics Systems (ICCES)*, 1151–1156. IEEE. [Cited on page 7.]

Cai, H.; Zhang, Z.; Li, L.; and Fu, X., 2019. A large-scale study of application incompatibilities in android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 216–227. [Cited on page 3.]

Cai, Y. and Zhang, C., 2023. A cocktail approach to practical call graph construction. *Proceedings of the ACM on Programming Languages*, 7, OOPSLA2 (2023), 1001–1033. [Cited on page 7.]

Dean, J.; Grove, D.; and Chambers, C., 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995 9*, 77–101. Springer. [Cited on page 7.]

EnterpriseAppsToday, 2024. Android statistics. https://www.enterpriseappstoday.com/stats/android-statistics.html. Accessed: 2024-05-23. [Cited on page 3.]

Feng, P.; Ma, J.; Li, T.; Ma, X.; Xi, N.; and Lu, D., 2020. Android malware detection based on call graph via graph neural network. In *2020 International Conference on Networking and Network Applications (NaNA)*, 368–374. IEEE. [Cited on page 8.]

Fourtounis, G.; Triantafyllou, L.; and Smaragdakis, Y., 2020. Identifying java calls in native code via binary scanning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 388–400. [Cited on page 8.]

Gok, N. and Khanna, N., 2013. *Building Hybrid Android Apps with Java and JavaScript: Applying Native Device APIs*. O'Reilly Media, Inc. [Cited on page 20.]

Google LLC, 2024. Google play. https://play.google.com. Accessed: 2024-02-17. [Cited on pages 17 and 18.]

Huang, C. Y.; Chiu, C. H.; Lin, C. H.; and Tzeng, H. W., 2015. Code coverage measurement for android dynamic analysis tools. In *2015 IEEE International Conference on Mobile Services*, 209–216. IEEE, New York, NY, USA. doi: 10.1109/MobServ.2015.38. [Cited on page 21.]

Hui, N. M.; Chieng, L. B.; Ting, W. Y.; Mohamed, H. H.; and Arshad, M. R. H. M., 2013. Cross-platform mobile applications for android and ios. In *6th Joint IFIP Wireless and Mobile Networking Conference (WMNC)*, 1–4. IEEE. [Cited on page 20.]

Ikram, M.; Beaume, P.; and Kâafar, M. A., 2019. Dadidroid: An obfuscation resilient tool for detecting android malware via weighted directed call graph modelling. *arXiv preprint arXiv:1905.09136*, (2019). [Cited on page 7.]

Ismail, U., 2009. Incremental call graph construction for the eclipse ide. *University of Waterloo Technical Report*, (2009). [Cited on page 7.]

Kaur, A. and Kaur, K., 2022. Systematic literature review of mobile application development and testing effort estimation. *Journal of King Saud University-Computer and Information Sciences*, 34, 2 (2022), 1–15. [Cited on page 20.]

Kovacheva, A., 2013. Efficient code obfuscation for android. In *Advances in Information Technology: 6th International Conference, IAIT 2013, Bangkok, Thailand, December 12-13, 2013. Proceedings 6*, 104–119. Springer. [Cited on page 21.]

Li, L., 2016. Boosting static security analysis of android apps through code instrumentation. (2016). [Cited on page 9.]

Li, L.; Bissyandé, T. F.; Octeau, D.; and Klein, J., 2016. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 318–329. [Cited on pages 8 and 21.]

Li, L.; Bissyandé, T. F.; Papadakis, M.; Rasthofer, S.; Bartel, A.; Octeau, D.; Klein, J.; and Traon, L., 2017. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88 (2017), 67–95. [Cited on page 8.]

Lin, C.-M.; Lin, J.-H.; Dow, C.-R.; and Wen, C.-M., 2011. Benchmark dalvik and native code for android system. In *2011 Second International Conference on Innovations in Bio-inspired Computing and Applications*, 320–323. IEEE. [Cited on page 8.]

Mumtaz, H. and El-Alfy, E.-S. M., 2017. Critical review of static taint analysis of android applications for detecting information leakages. In *2017 8th International Conference on Information Technology (ICIT)*, 446–454. IEEE. [Cited on page 3.]

Nikolić, . and Spoto, F., 2014. Reachability analysis of program variables. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 35, 4 (2014), 1–68. [Cited on page 7.]

NoxLimited, 2023. Nox android emulator. https://www.bignox.com/. Version 7.0.5.8, accessed 2023-11-01. [Cited on page 14.]

Qu, Z.; Alam, S.; Chen, Y.; Zhou, X.; Hong, W.; and Riley, R., 2017. Dydroid: Measuring dynamic code loading and its security implications in android applications. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 415–426. IEEE. [Cited on page 8.]

Reif, M.; Kübler, F.; Eichberg, M.; and Mezini, M., 2018. Systematic evaluation of the unsoundness of call graph construction algorithms for java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, 107–112. [Cited on pages 5 and 8.]

Ruggia, A.; Possemato, A.; Dambra, S.; Merlo, A.; Aonzo, S.; and Balzarotti, D., 2023. The dark side of native code on android. *Authorea Preprints*, (2023). [Cited on page 8.]

Ryder, B. G., 1979. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, , 3 (1979), 216–226. [Cited on page 1.]

Sahin, O.; Aliyeva, A.; Mathavan, H.; Coskun, A.; and Egele, M., 2019. Randr: Record and replay for android applications via targeted runtime instrumentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 128–138. IEEE. [Cited on page 9.]

Samhi, J.; Gao, J.; Daoudi, N.; Graux, P.; Hoyez, H.; Sun, X.; Allix, K.; Bissyandé, T. F.; and Klein, J., 2022. Jucify: A step towards android code unification for enhanced static analysis. In *Proceedings of the 44th International Conference on Software Engineering*, 1232–1244. [Cited on page 8.]

Skylot, 2024. Jadx: Dex to java decompiler. https://github.com/skylot/jadx/releases/tag/v1.4.0. Version 1.4.0, accessed 2024-04-16. [Cited on page 14.]

Song, W.; Huang, Q.; and Huang, J., 2018. Understanding javascript vulnerabilities in large real-world android applications. *IEEE Transactions on Dependable and Secure Computing*, 17, 5 (2018), 1063–1078. [Cited on page 20.]

Sun, X.; Li, L.; Bissyandé, T. F.; Klein, J.; Octeau, D.; and Grundy, J., 2021. Taming reflection: An essential step toward whole-program analysis of android apps.

ACM Transactions on Software Engineering and Methodology (TOSEM), 30, 3 (2021), 1–36. [Cited on pages 8 and 9.]

Tip, F. and Palsberg, J., 2000. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 281–293. [Cited on page 7.]

Vallée-Rai, R.; Co, P.; Gagnon, E.; Hendren, L.; Lam, P.; and Sundaresan, V., 2010. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, 214–224. [Cited on pages 2, 3, 10, and 14.]

Vinayaka, K. and Jaidhar, C., 2021. Android malware detection using function call graph with graph convolutional networks. In *2021 2nd International Conference on Secure Cyber Computing and Communications (ICSCCC)*, 279–287. IEEE. [Cited on page 8.]

Whaley, J., 2007. *Context-sensitive pointer analysis using binary decision diagrams*. Ph.D. thesis, Stanford University. [Cited on pages 7 and 14.]

Yang, Y.; Wei, Z.; Xu, Y.; He, H.; and Wang, W., 2018. Droidward: an effective dynamic analysis method for vetting android applications. *Cluster Computing*, 21 (2018), 265–275. [Cited on page 8.]

Zhang, Z., 2024. Maxim: A tool for generating pseudo-user behaviors. https://github.com/zhangzhao4444/Maxim. Accessed: 2024-05-23. [Cited on pages 13, 14, and 21.]