Spring 4-23-2020

A Memory Usage Comparison Between Jitana and Soot

Yuanjiu Hu

University of Nebraska - Lincoln, yuanjiu.hu@huskers.unl.edu

A MEMORY USAGE COMPARISON BETWEEN JITANA AND SOOT

by

Yuanjiu Hu

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Witawas Srisa-an

Lincoln, Nebraska

May, 2020

A MEMORY USAGE COMPARISON BETWEEN JITANA AND SOOT

Yuanjiu Hu, M.S.

University of Nebraska, 2020

Adviser: Witawas Srisa-an

There are several factors that make analyzing Android apps to address dependability and security concerns challenging. These factors include (i) resource efficiency as analysts need to be able to analyze large code-bases to look for issues that can exist in the application code and underlying platform code; (ii) scalability as today's cybercriminals deploy attacks that may involve many participating apps; and (iii) in many cases, security analysts often rely on dynamic or hybrid analysis techniques to detect and identify the sources of issues.

The underlying principle governing the design of existing program analysis engines is the main cause that prevents them from satisfying these factors. Existing designs operate like compilers, so they only analyze one app at a time using a "close-world" process that leads to poor efficiency and scalability. Recently, Tsutana et al. introduced JITANA, a Virtual Class-Loader (VCL) based approach to construct program analyses based on the "open-world" concept. This approach is able to continuously load and analyze code. As such, this approach establishes a new way to make analysis efforts proportional to the code size and provides an infrastructure to construct complex, efficient, and scalable static, dynamic, and hybrid analysis procedures to address emerging dependability and security needs.

In this thesis, we attempt to quantify the performance benefit of JITANA through the lens of memory usage. Memory is a very important system-level resource that if not expended efficiently, can result in long execution time and premature termination of a program. Existing program analysis frameworks are notorious for consuming a large amount of memory during an attempt to analyze a large software project. As such, we design an experiment to compare the

memory usage between JITANA and SOOT, a widely used program analysis and optimization framework for Java. Our evaluation consists of using 18 Android apps, with sizes ranging from 0.02 MB to 80.4 MB. Our empirical evaluations reveal that JITANA requires up to 81% less memory than SOOT to analyze an app. At the same time, it can also analyze more components including those belonging to the application and those belonging to the Android framework.

ACKNOWLEDGMENTS

I would like to express my gratitude for those who have been supporting me throughout my academic career; without them, I could not have far with this work.

First, I would like to give special recognition to **my family** for providing me with the opportunity to continue my education on a higher level. Their encouragement and love along the way tremendously inspired me to pursue my path despite difficult times.

I would like to thank my academic advisor, **Dr. Witty Srisa-an** for patiently guiding me through this project. I started this project without much specific subject knowledge, and Dr. Witty was kind enough to thoroughly explain everything I needed to know to get started. His guidance was essential to this thesis, and I cannot show my appreciation enough.

I would also like to mention **Dr. Justin Bradley** and **Dr. Vu Nguyen** for reviewing my work as a committee and giving me valuable feedback on my thesis.

Last but not least, I appreciate the **University of Nebraska - Lincoln** for fostering a great learning environment during my four semesters of study. I am remarkably satisfied with my decision to finish my university education here at UNL almost in 2018, and I believe the lessons that I learned here will come a very long way in my future endeavors.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Static program analysis has been successfully utilized to enhance software quality, dependability, and security over the last few decades. The main idea of static program analysis is to analyze software (source code, intermediate representation code, or binary code) without actually executing programs. The types of analysis that can be performed include identifying and computing software metrics, detecting software defects and security vulnerabilities, applying formal methods, and verifying safety-critical software systems; e.g., [1, 2, 3, 4, 5, 6, 7].

Because analysis is done directly on a code project without executing it, the first step to perform program analysis is loading the project. Existing static program analysis techniques take an approach similar to a compiler; that is, it first loads all code in the project to ensure completeness. It then analyzes the loaded code. This type of analysis often makes a "closed-world" assumption; that is, the analysis is done on the *complete* code; and once the analysis is done, the results *cannot change.* (We refer to this type of approach as "compiler-based".) Fundamentally, the goal is to analyze all relevant components in the project. However, achieving this goal is quite challenging in modern computing systems because the analysis requirements have changed to address emerging security and dependability issues better. Below, we highlight some of these challenges.

Challenge 1. Modern programming languages and platforms provide rich library support.

Therefore, loading just the application code alone may not be sufficient to ensure dependability and security. Recently, we have seen various instances of security vulnerabilities and software defects that affect a large number of computer systems worldwide [8,9,10]. In these examples, the root causes exist in the underlying systems or supporting libraries. *To detect these issues, we need to analyze the application code and the supporting libraries together. However, such analysis techniques have been shown to result in excessive memory consumption, making them infeasible to scale up to meet this requirement.*

Challenge 2. In addition to the high memory overhead, compiler-based analysis approaches also have issues dealing with the dynamism of modern programming languages. As an example, Android supports both Java Reflections and Dynamic Code Loading (collectively referred to in this thesis as RDCL). These mechanisms allow new classes to be dynamically loaded at runtime, possibly from external sources [11,12,13,14]. Recently, we have seen RDCL used to deliver malicious payloads in highly elusive malware [15, 16, 17, 18, 19, 20]. *When a static analysis approach is used to analyze apps with RDCL, it cannot provide complete coverage on its own because additional classes can be added to the existing code-base, making the initial static analysis results incomplete and creating the need to perform analysis on the entire code-base even for a small code change.*

Challenge 3. Today's apps can communicate with each other as well as provide services for one another. For example, Android provides a mechanism called Inter-App Communication (IAC) to allow apps to communicate and share services among themselves. Through mechanisms such as IAC, a new breed of malware in which multiple apps can collude has been introduced [21, 22]. Thus, there is an emerging need to analyze these complex interactions among apps in a device to identify such threats. Unfortunately, the current compiler-based static analysis approaches address such issues by analyzing one app at a time and afterward, combine the results [1, 23, 24, 25, 26]. *Due to the need to perform analysis one app at a time,*

these techniques are too inefficient, making them unsuitable for at-speed per-device security analysis.

1.1 Motivation

These challenges clearly illustrate the inherent limitations of existing compiler-based program analysis approaches that make the closed-world assumption. To effectively address these challenges, Tsutano et al. recently developed JITANA, a different type of static analysis framework that constructs static program analysis information incrementally. JITANA's major underlying insight to is to break the traditional closed-world assumption, and instead, rely on the "open-world" assumption achieved through the notion of "incrementality". The underlying idea is to *incrementally* load and analyze only the necessary code that is the transitive closure of all the programs' code and the underlying *Android Development Framework* (*ADF*) code. The idea is inspired by advancements in language runtime systems such as *Java Virtual Machine* (*JVM*) or *Android Virtual Machine* (*AVM*) to support incremental execution via class-loading and optimization of an application through dynamic compilation.

The first inspiration for JITANA is a class-loader, a runtime system used in both Java and Android VMs to load only the necessary classes at runtime. A class-loader takes advantage of application structures that partition code into classes to naturally and incrementally load each class as it is needed for execution. It also supports various forms of late binding to quickly resolve the ambiguity of finding which class to load through delegation. At the heart of JITANA is the *Class-Loader Virtual Machine* (*VCL*) that fully adheres to the published class-loader specification [27], but it operates as a stand-alone component in JITANA; i.e., it does not operate as part of a language runtime system such as a JVM or AVM. The VCL uses reachability analysis to uncover classes that must be loaded and delegates to resolve all possible statically discoverable late binding targets.

The second inspiration is the Just-In-Time (JIT) compiler commonly used in modern programming language runtime systems (e.g., those used to support Java, Python, and JavaScript). A JIT compiler naturally performs program analysis in an *incremental* fashion; i.e., it only analyzes a small portion of the code at a time (e.g., a method or a trace [28]) and then performs optimization to generate the backend code. By utilizing VCL, we can already load code incrementally. As such, it creates an opportunity to incrementally analyze each class right after it is loaded. In this approach, the initial analysis would be intraprocedural. Once the analysis on that class is completed, reachability analysis is used to uncover other reachable classes and construct and propagate interprocedural information (e.g., method call graph) among analyzed classes.

1.2 Contributions

In this thesis, we attempt to quantify the underlying difference in memory usage between JITANA and SOOT, a widely used program analysis and optimization framework for Java. As previously stated, we hypothesize that the incremental nature of JITANA would allow it to analyze more code than a traditional, compiler-based program analysis framework such as SOOT, given the same amount of memory.

To validate our hypothesis, we design an experiment to evaluate memory usage of each approach. Our evaluation consists of using 18 apps. Each app is analyzed by both analysis frameworks. The evaluation methodology includes developing tools to periodically measure virtual memory usage during an analysis task performed by each framework. We also report the number of methods analyzed by each approach. Note that the reported number of methods by each analysis framework is different, because JITANA can analyze the complete framework code while SOOT does not.

In summary, the main contribution of this work is to compare the memory performance

between the new incremental program analysis approach (represented by JITANA) and the traditional compiler-based program analysis approach (represented by SOOT). Our evaluation result indicates that the incremental nature of JITANA can result in 81% memory conservation than the traditional approach. As a result, the memory saving allows JITANA to analyze more code, including critical components in the Android framework used by the application.

1.3 Organization

The rest of this thesis is organized as follows. Chapter 2 describes exising program analysis challenges, which motivate the development of JITANA. Chapter 3 describes the design of JITANA and illustrate some of the differences in analysis results between JITANA and SOOT. Chapter 4 provides the detailed information related to our experiments. Chapter 5 reports the experimental results. We conclude the thesis in Chapter 6.

Chapter 2

Static Analysis Challenges and How They Are Addressed

As mentioned in Chapter 1, static program analysis frameworks are facing more demanding analysis requirements to address emerging dependability and security issues. In this section, we highlight three emerging challenges that today's static program analysis frameworks cannot address effectively and efficiently.

Challenge 1. The complex interactions between today's application and the underlying frameworks and libraries create the needs for static program analysis framework to be able to analyze the application code in unison with the underlying framework code [9, 10, 29]. As an example, solving dependability and security issues due to platform updates in a smart-mobile device such as Android [30, 31] (e.g., determining why an app crashes after a platform update) would require that the application code and the underlying Android Development Framework (ADF) be analyzed together [4, 8, 26, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42]. However, analyzing ADF code along with application code has been shown to result in excessive memory consumption.

Figure 2.1 illustrates a typical memory requirement of a compiler-based approach when used to analyze an Android application and ADF code. In our illustration, we focus on *App 0* with its code size equals to A_0. It is currently running on ADF API-Level 25. This ADF has a code size of B_0. To thoroughly analyze all relevant components used by *App 0*, the

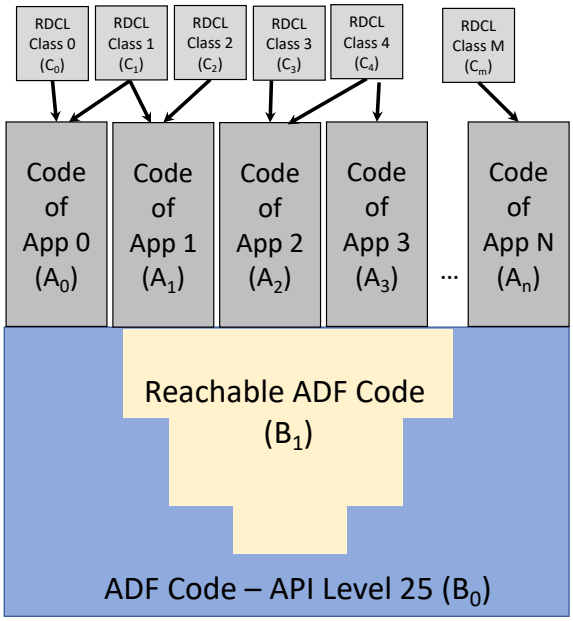

Figure 2.1: *An Illustration of Different Static Analysis Challenges.*

amount of code that a compiler-based static analysis approach needs to load is $A_0 + B_0$ due to the closed-world assumption. Currently, the average Android app file size is about 12 MB; the file size of API-Level 25 is 160 MB. Thus, the total code that must be loaded is approximately 172 MB (12 MB + 160 MB or 14 times the size than analyzing *App 0* alone). This is before performing any analysis. Typically, constructing program analysis information such as control flow, data flow, and points to graphs would require a significant amount of additional memory. For example, SOOT a widely used program analysis framework for Java and Android, recommends a minimum heap size of 10 GB [11]. Even with that large heap size, when we use SOOT to analyze large apps, we still see occasional "out of memory" errors.

There have been approaches that attempt to reduce memory consumption while considering the underlying ADF code. They achieve this goal by using modeling, mining ADF code, or automating learning of existing code [26, 41, 42] in place of performing analysis of application and ADF code simultaneously. These approaches perform analysis in multiple steps. For example, CID first performs analysis of the application code to identify API calls into ADF.

It then analyzes the ADF code to construct API Life Cycle Models and Conditional Call Graph, which performs path-sensitive backward interprocedural data flow analysis toward the code that performs version checks to reduce false positives [41]. A study by Scalabrino et al. [42] reported that CiD cannot finish analyzing nearly 20% of apps (1,971/11,863) after one hour.

ACRyL, on the other hand, only analyzes the application code and uses automated learning to observe code changes with respect to API compatibility. This approach can also suggest repairs. However, both approaches do not detect any compatibility issues that occur through nested API calls within the ADF code. In addition, both approaches rely on pre-analyzed data so that can produce incomplete results (e.g., in the case that some necessary features are not part of the modeling efforts) and are not responsive to changes due to frequent ADF updates or introduction of new issues (e.g., changes or new issues that were not part of the learning datasets) [26, 42]. *We conclude that excessive memory requirements can cause program analysis techniques to be inefficient and even unreliable when there is not enough memory. Furthermore, approaches to conserve memory can be ineffective in situations where code is frequently updated.*

Challenge 2. As previously mentioned, mechanisms such as RDCL allow new classes to be dynamically loaded at runtime, possibly from external sources. RDCL is commonly used to support several features in Android apps, including backward compatibility, dynamic updates, component plugins, and serving advertisements. Recently, we have also seen RDCL used to deliver malicious payloads in highly elusive malware [15, 16, 17, 18, 19, 20]. Our preliminary inspection of 60 games (e.g., NBA Live and Roblox), social network apps (e.g., Facebook and Pinterest), and multimedia apps (e.g., Pandora and Spotify) from Google Play revealed that 57% of these apps use RDCL to load classes dynamically. This prolific use of RDCL is a significant increase from an earlier reported result, wherein only 16% of the top 50 apps from 2013 were found to use RDCL [12].

When a static analysis approach is used to analyze apps with RDCL, it cannot provide complete coverage on its own because additional classes can be added to the existing code-base from external sources, making the initial static analysis results incomplete. Note that there are existing static analysis approaches that can analyze apps with RDCL as long as those dynamically loaded classes are accessible at the analysis time (e.g., these classes are hidden in the code area of the project) [6, 7, 43, 44]. However, their analyses cannot discover classes that are downloaded at runtime or hidden in inaccessible areas. To ensure completeness, dynamically loaded classes must also be included as part of subsequent analysis attempts. Work by Bodden et al. (TamiFlex) Zhauniarovich et al. (StaDyna) and Rasthofer et al. (Harvester) perform hybrid analysis to address this challenge [11, 12, 13, 14]. First, their approaches use dynamic analysis to identify and capture the dynamically loaded classes and then add these classes to the project. Afterward, they statically reanalyze the project with the addition of newly discovered classes.

Once again, we focus on *App 0* in Figure 2.1. As shown, it uses RDCL to load two additional classes (*Class 0* and *Class 1*) of sizes C_0 and C_1, respectively. Let us further assume that these two classes are acquired from external sources, so they are not available anywhere in the original code-base. If *App 0* has already been statically analyzed, as soon as *Class 0* and *Class 1* are loaded, *App 0* must be analyzed again. In the case that there is a very long delay between the loading of *Class 0* and the loading of *Class 1*, an analysis is likely done after the loading of each respective class. In the compiler-based approach, the effort to reanalyze the project after *Class 0* is loaded (i.e., $A_0 + B_0 + C_0$ in this case) is not proportional to that amount of code that changes (i.e., just C_0 in this case). *The inability to scalably deal with dynamism to today's programming languages make compiler-based static analysis approaches ineffective and inefficient in addressing today's dependability and security concerns such as these.*

Challenge 3. Today's apps can communicate with each other as well as provide services for

one another. For example, Android provides a mechanism called Inter-App Communication (IAC) to allow apps to communicate and share services among themselves. Through mechanisms such as IAC, interactions among apps installed on a device can be quite complex, and current compiler-based static analysis approaches are not capable of scalably analyze these apps with timely results that can be used to address current security threats.

One of such emerging threats is colluding malware [21,22]. By leveraging IAC, sophisticated collusive security threats exploit multiple apps to create longer calling paths to launch malicious activities. Identifying this type of threat requires that all apps in the calling paths be analyzed. Thus, the most effective and efficient way to detect colluding malware is to perform an analysis per device because calling paths of colluding malware can only involve apps installed on that device. Unfortunately, a recent study shows that, on average, there are close to 100 apps installed on an Android device. There are currently over 3.3 million Android apps on Google Play, and Google has made 17 releases of Android Platforms. The sheer variability among apps and platforms make pre-analyzing all existing apps or even just the most popular apps infeasible.

As we have shown in the first challenge, current static analysis approaches analyze one app (optionally with ADF) at a time. Therefore, techniques that have been designed to detect colluding malware, but built on top of compiler-based static analysis approaches, also analyze one app at a time and then perform inter-component and inter-app analysis afterward [45, 46, 47, 48]. As an illustration, we revisit Figure 2.1. We have *App 0* to *App N* installed on a device running ADF API-Level 25.

There are two general approaches to detect potential collusion. The first approach analyzes a small set of programs for connections at a time. Li et al. introduce (IccTA), an approach based on Soot. It consists of several tools [1, 23, 24] that perform per app analysis (i.e., it needs to analyze *App 0* to *App N*, one app at the time). It then performs cross-app analysis on the results to find possible connections among apps. Once apps with connections are

identified, they are combined into a non-executable but analyzable app using APKCombiner, a tool capable of combining only very few apps for analysis [49]. We have experimented with this approach and found it to be non-scalable.

The second approach aims to achieve better scalability by creating and then analyzing architectural models. SEALANT [25] combines static analysis with runtime monitoring to prevent IAC attacks. It statically analyzes (again using Soot) each app to identify channels. It then extracts the architectural model of each app and performs compositional analysis of these models to detect vulnerable channels [26]. By analyzing models, this approach is more scalable than the first approach. By not fully analyzing the source code, the analysis is more restrictive (i.e., it can only analyze issues that can be found through the models). In addition, the compositional analysis iteratively builds analysis results from one app on top of prior results. If more comprehension is needed (e.g., debugging or formulating repairs), per app analysis is still needed to provide additional information. *Due to the need to perform analysis one app at a time, these techniques are too inefficient to be used for near-real-time or at-speed security analysis.*

In the next chapter, we describe Jitana, a program analysis framework introduced by Tsutano et al. and highlight how it can address these three challenges.

Chapter 3

Overview of JITANA

Tsutano et al. introduced JITANA, an incremental program analysis framework, to address these three challenges [50]. The critical characteristic that enables JITANA to be able to address these three challenges is the notion of "incrementality". By being able to load and analyze code incrementally, JITANA requires less memory, allowing it to tackle large projects while incurring reasonable memory usage. JITANA achieves incrementality by the use of a *Virtual Class-Loader* (*VCL*).

3.1 Design of JITANA

As previously mentioned, existing runtime concepts that include class-loading and dynamic compilation and optimization inspire the creation of *VCL*. Naturally, class-loading supports incrementally loading of only the necessary classes. Each instance of `ClassLoader`, which is a Java class inherited from an abstract class `Ljava/lang/ClassLoader;`, has a reference to a parent class-loader.

The class-loader specification [27] supports *Delegation Hierarchy Principle* that is used to discover and load classes through delegation [51]. When a class-loader cannot find a class, it delegates the task to its parent class-loader. A class can be located and loaded by one of the three class-loaders (i.e., Bootstrap, Extension, and Application). There are five supported

methods: `loadClass`, `defineClass`, `findClass`, `findLoadedClass`, and `Class.forName`. These methods can be used to define a class or find a class, load it, and initialize it. The loaded classes are also unique.

VCL in JITANA strictly follows the Java classloading specification from Oracle [27]. It supports all essential methods to support various class-loading and defining activities. It also supports *Delegate Hierarchy Principle*, *Visibility Principle*, and *Uniqueness Property*. The *Delegate Hierarchy Principle* contains several rules to define how to find and load classes and ensure that the classloader does not load any duplicate classes. The principle also defines how delegation among the three classloaders should work.

As an example, if VCL needs to load a new class, it first delegates the request to *Application Classloader*, the request is further delegated to *Extension Classloader*, and finally, the last delegation is made to *Bootstrap Classloader*. Delegation creates the searching and loading hierarchy. *Bootstrap Classloader* first searches in the Bootstrap classpath to find the class. If it cannot find it in the Bootstrap classpath, *Extension Classloader* searches in the Extension classpath. If it cannot find it in the Extension classpath, *Application Classloader* searches in the Application classpath to find and load the class. If it is still not found, *VCL* generates an error.

During static analysis, VCL uses reachability analysis to identify classes that we need to load. The main idea is to analyze the methods in each class to identify any additional method calls within those methods to load classes to which these methods belong. For each class discovered through reachability analysis, VCL applies the *Delegate Hierarchy Principle* to locate and load that class. This capability provides JITANA with the ability to discover and analyze loaded classes incrementally.

In addition, VCL also preserves the *Visibility Principle*, which states that a class loaded by a parent class-loader (e.g., *Extension Classloader*) is visible to the child class-loader; i.e.,

the class is visible to *Extension* and *Application Classloaders* but not *Bootstrap Classloader*. It also ensures that each loaded class is unique (*Uniqueness Property*).

In dealing with dynamic polymorphism, *VCL* exploits an insight that at runtime, both a method or class name and the classloader information define a method or a class. Thus, it records class-loader information as part of an analysis, so that it can have information about the defining class-loader for a class. This information is similar to what being kept inside the JVM or AVM to resolve dynamic method dispatch. However, the information is not as precise, so all possible targets of a virtual interface are included for analysis. While this can add additional classes and methods that must be analyzed, it is still much smaller than loading the entire code-base. It is also a small tradeoff to ensure completeness.

3.2 Program Analysis with JITANA

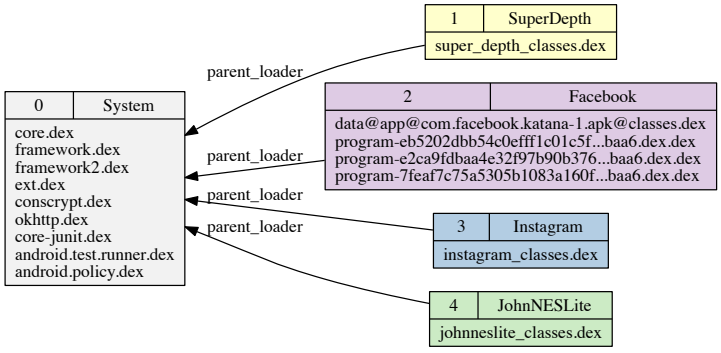

Figure 3.1: An illustration of a class-loader graph

JITANA presents the analysis results as hierarchical graphs that are Boost compliant [52]. These graphs consist of class-loader graph, class graph, method-call graph, and instruction graphs. The class-loader graph contains information about apps under analysis. Figure 3.1

illustrates a class-loader graph that includes four apps: *SuperDepth*, *Facebook*, *Instagram*, and *JohnNESLite*.

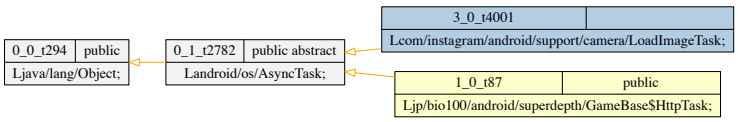

Figure 3.2: A snippet of a class-graph related to AsyncTask

A class graph, shown in Figure 3.2, includes the relationship among classes. The figure only shows a portion of the class graph that is related to `AsyncTask`. A method-call graph, shown in Figure 3.3 includes method-call relationship.

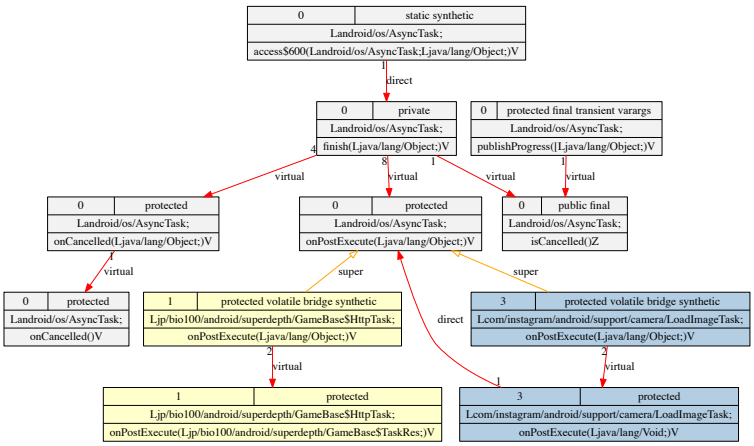

Figure 3.3: An illustration of a method-call graph

For each method, there is an instruction graph (shown in Figure 3.4) containing Dex instructions. It also contains intraprocedural data-flow (red arrows) and control-flow (blue arrows) information.

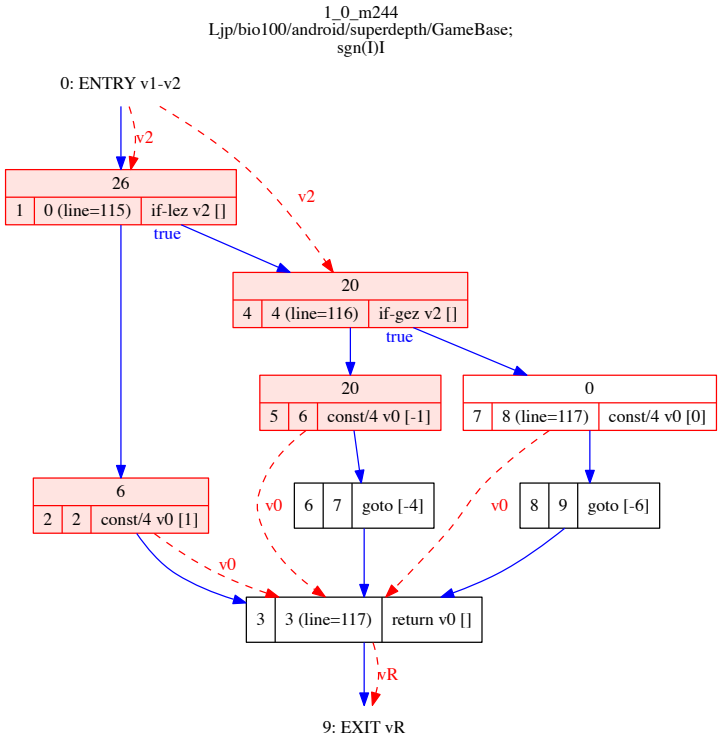

Figure 3.4: An illustration of an instruction graph

3.3 Comparing Results Produced by JITANA and SOOT

We create a simple *HelloWorld* program that contains three classes: `HelloWorld`, `HelloWorldTwo`, and `HelloWorldThree`. `HelloWorld` invokes `helloClass2Method1`, which is a method in `HelloWorldTwo`. Method `helloClass2Method1` simply calls `println` with is a native method in `java.io` library. Note that the program does not invoke any method in class `HelloWorldThree`. Figure 3.5 illustrates the method call graph produced by SOOT, and Figure 3.6 illustrates the method call graph produced by JITANA. Also note that we use the same program for analysis but the one used by SOOT was compiled into Java bytecode and the one used by JITANA was compiled into Android Dex code.

As shown in Figure 3.5, SOOT treats any calls to the underlying library (e.g., `java/lang/Object`

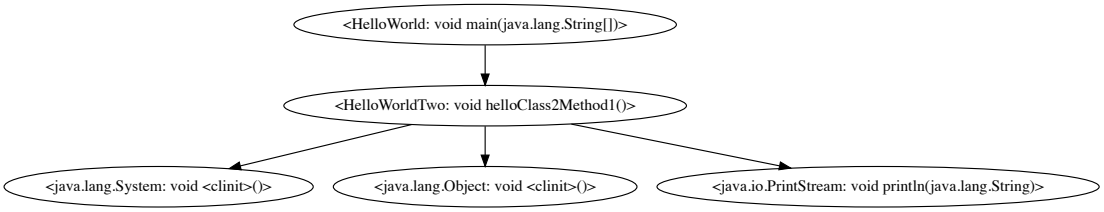

Figure 3.5: A Method-call-graph of *HelloWorld* Produced by SOOT

and `java/io/PrintStream`) as terminals. That is, the analysis ends at these calls. JITANA, on the other hand, continue to load any of those system classes written in Dex. As such, its method call graph also includes system-level methods such as `finalize`, `hashCode`, and `wait`. The call appearing within the red circle in Figure 3.6, represents the actual call from class `HelloWorld` to a method in class `HelloWorldTwo`.

The graph generated by JITANA also shows existing methods in a class even though they have not been invoked. (Note that each red arrow indicates a method invocation.) For example, methods such as `notifyAll`, `notify`, and `finalize` belong to class `java/lang/objects` but they are not invoked. Similarly, methods `helloWorldClass2Method2` and `helloWorldClass3Method1` belong to `hellowWorldClass2` and `hellowWorldClass3`, respectively. The are also not invoked. However, they are included because JITANA analyzes all the Dex methods in every loaded class.

Also note that JITANA method call graph does not show `java/io/println`. Upon further inspection, because `java/io/PrintStream` is a native library, *VCL* cannot load it, and therefore, it is not analyzed. However, the call to `println` still appears in the instruction graph as a method call.

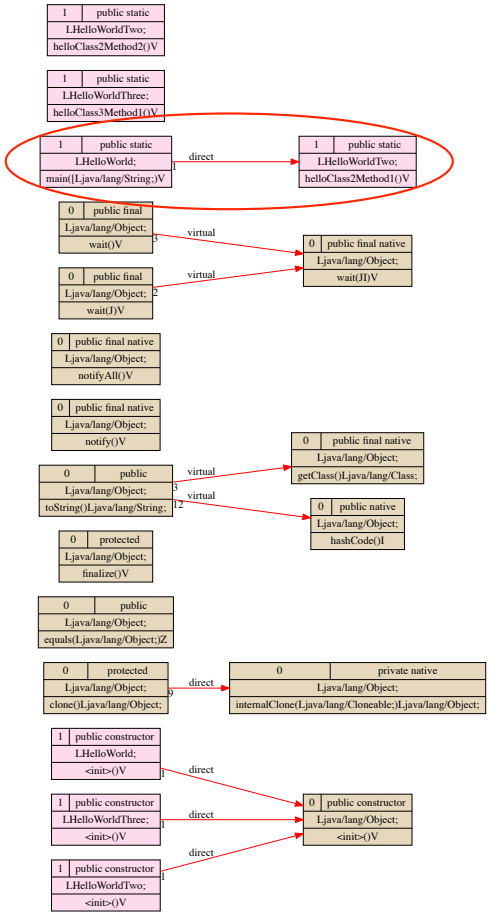

Figure 3.6: A Method-call-graph of *HelloWorld* Produced by JITANA

3.4 Addressing Existing Challenges

Concerning the previously mentioned first challenge, JITANA [50] addresses it by being able to simultaneously analyze application code and ADF code efficiently and scalably within a typical memory availability of a workstation or laptop. Unlike existing approaches to reduce memory consumption, JITANA achieves this goal by loading just the necessary classes from the app and ADF. This way, the effectiveness to detect, debug, and repair complex dependability and security issues is not compromised.

To address the second challenge, JITANA can perform hybrid analysis to uncover RDCL

classes and analyze them efficiently. Unlike existing approaches to repeatedly analyze the entire code-base for each code change, JITANA achieves this goal by expending the analysis effort that is proportional to the amount of code that changes. As shown in Figure 2.1, JITANA only analyzes C_0 and appends the new results to the previously computed results.

To address the third challenge, JITANA can analyze an extensive collection of apps or apps installed on a device at speed. Unlike existing approaches that combine analysis results from analyzing a single app at a time, JITANA achieves this goal by being memory-efficient and better organization of analysis data to allow multiple apps to be analyzed simultaneously. However, we can still distinguish each app within an analysis attempt.

In the next chapter, we investigate the memory usages of SOOT and JITANA and report the overall performances of both systems.

Chapter 4

Experimental Evaluation

In this chapter, we report the results of our investigation to compare memory usage and performance of SOOT and JITANA. We also used APKTOOL [53] for decompilation of the APK so that we can assess the number of lines of code. In our evaluation, we address the following two research questions:

Research Question 1: Is JITANA more memory efficient than SOOT and if yes, by how much?

Research Question 2: Is JITANA more efficient, in terms of analysis time than SOOT?

4.1 Objects of Analysis

We want to quantify the difference in memory usage between the compiler-based approach, represented by SOOT, and the classloader-based approach, represented by JITANA. By intuition, the classloader-based approach incrementally loads code and therefore should be much more memory-efficient and scalable. In this work, we attempt to verify the intuition and quantify the savings. We used both approaches to analyze 18 Android apps and measured the memory usage over time. We also recorded the analysis time.

To ensure a fair and comprehensive evaluation, we selected 18 Android apps with sizes ranging from less than 1 MB to over 50 MB as subjects for our experiment. We used open-source repositories, such as F-Droid and APK Pure to obtain these apps [54, 55, 56]. The intention of having a wide range of app sizes is to observe how increasing size impact memory performance for both analysis platforms.

Next, we establish baselines for the relative size of each Android app. Although the size of APK is what an end-user would normally see when downloading an Android app, a large portion of the APK is dedicated to related assets and resources, which do not contribute to the logic of the app. To gain a more thorough understanding of the relative size of apps, we first import and analyze the APK files in Android Studio, which gives us an overview of the basic structure of each APK, such as the number and size of DEX files, number of classes/methods defined, etc [57]. As a backup measure, we also decompiled each APK using Apktool and counted the number of smali lines from each decompiled APK. In the end, we used the total size of DEX files (DEX code is the Android VM binary) and the number of lines of smali code (human-readable DEX disassembly) as the two main baseline indicators of the complexity of performing static analysis. Table 4.1 describes the basic characteristics of each app.

4.2 Variables and Measures

Independent Variables. Our independent variables involve the baseline technique used in our study. We use FlowDroid as the baseline system. FlowDroid is a program analysis framework based on Soot that has been modified to support analysis of Android apps. In its original form, Soot performs analysis by assuming that the main method is the only entry point into the program. However, Android applications can have multiple entry points.

App Name	APK Size	DEX Size	Number of Smali Lines	Number of defined methods
Snake	18 KB	7.3 KB	2,830	44
Battery Indicator	2 MB	180 KB	116,593	2337
BitClock	566 KB	271 KB	158,884	4743
AdBlockPlus	2.6 MB	404 KB	240,226	5656
Guitar Flash	45.2 MB	730 KB	439,016	10608
Calculator	4.3 MB	674 KB	461776	10623
iFixit	3.3 MB	743 KB	471,658	10323
Slots Pharaohs Fire	45.1 MB	994 KB	673,772	17066
TypoLab	45.2 MB	1.1 MB	841,209	21735
Cute Animals	45.2 MB	1.3 MB	1,114,023	25647
Dolphin EMU	13.8 MB	2 MB	1,258,159	28648
BBC Weather	9.2 MB	2.4 MB	1,659,200	40105
Bike Citizens Bicycle GPS	45.2 MB	2.7 MB	1,827,500	43604
The Child of Slendrina	45.1 MB	2.2 MB	1,839,378	43611
Moto Rider	45.1 MB	2.7 MB	2,120,364	48812
Doodle Army	45.1 MB	3.8 MB	2,820,695	6295
BBC News	15.5 MB	4.3 MB	3,257,594	65677
Adobe Lightroom	80.4 MB	6.2 MB	4,485,608	101022

Table 4.1: Basic characteristics of experimental subjects.

FlowDroid, an Android taint analysis tool built on top of Soot, solves this issue by creating a custom main method that considers all possible combinations of outgoing methods.

Since we are interested in comparing the static analysis performance of FlowDroid and Jitana, we set up both tools to generate only the method call graph of a single app at a time. On the other hand, FlowDroid performs taint analysis, which is an extra step on top of generating the method call graph. We wrote a Java program to configure a Soot instance with desired parameters, and then we instantiated FlowDroid to use the existing Soot instance to generate the method call graph without performing further analyses.

To make sure the same set of Android Framework methods are available, both FlowDroid and Jitana were configured to reference Android API level 25 framework files. We used the latest version of both FlowDroid (2.7.1) and Jitana (2018.4).

Dependent Variables. To measure the memory usage of Jitana and FlowDroid, we use existing performance monitoring tools available in macOS. We start the analysis and `top`, the process monitor utility, at the same time, and we take a snapshot of `top` every second to monitor the per-process memory usage reported in MB. We are particularly concerned with

the peak memory usage, as it is the most common limiting factor to perform very complex analyses.

Next, we calculate memory efficiency (ME) using the following formula:

$$ME = \frac{\text{Number of Methods in the Callgraph}}{\text{the Peak Memory Consumption in MB}}$$

Note that a higher value of ME reflect how efficient an analysis framework uses memory. To measure overall efficiency, we measure the record to perform each analysis and report the result in seconds

4.3 Study Operation

We set up our evaluation environment on an Apple MacBook Pro with 2.7 GHz Quad-Core Intel Core i7 and 16 GB LPDDR3 RAM running macOS Catalina (10.15.3). We repeat the experiments three times and measure the amount of memory and time required to perform the analysis of each app using the analysis techniques, each averaged over three attempts.

4.3.1 Threats to Validity

The primary threat to external validity in this study involves the object programs utilized. In this work, our objects are based on programs that have been widely used by prior research work [54, 55, 56]. We also ensure that they all can run on the same ADF (version 25 in this case).

The primary threat to internal validity involves potential errors in the implementation of our measurment process. To limit these, we extensively validated all of our measurement components and scripts to ensure correctness.

The primary threat to construct validity relates to the fact that we study efficiency measures relative to applications of JITANA, but do not yet assess whether the approach helps

software engineers or analysts addresses dependability and security concerns more quickly than current approaches. Next, we report the results of our evaluation.

Chapter 5

Results

Next, we answer the two research questions. We formulate these answers based on our empirical investigations to observe memory consumption and analysis time.

5.1 RQ1: Memory Savings

We report the overall performance in Table 5.1. We report the number of analyzed methods in columns II and V for SOOT and JITANA, respectively. Note that JITANA also analyzes methods in the ADF code, and therefore, it processes more methods than those processed by SOOT. In columns III and VI, we report the amounts of memory (in MB) needed to support the analysis of each app by SOOT and JITANA. We report the analysis time of both systems in columns IV and VII.

Soot completed analysis for all but one Android app, *Adobe Lightroom*, presumably due to incompatibility from newer programming practices. FlowDroid terminated with a runtime exception during the callgraph-construction phase. In all 18 apps, JITANA analyzed significantly more methods than SOOT. In the case of *BitClock*, the difference in the numbers of analyzed methods is about 81 times. In the case of *Battery Indicator*, the difference is about 3 times. On average, JITANA analyzes 11.84 times more methods than SOOT.

While JITANA analyzes more methods than SOOT in every application, it also requires less

App Name [I]	Soot Analyzed Methods [II]	Soot Utilized Memory (MB) [III]	Soot Analysis Time (Seconds) [IV]	Jitana Analyzed Methods [V]	Jitana Utilized Memory (MB) [VI]	Jitana Analysis Time (Seconds) [VII]
Snake	84	130	1	727	21	1
Battery Indicator	1,195	330	4	3,548	64	3
BitClock	57	175	1	4,631	57	2
AdBlockPlus	632	293	3	7,048	112	3
Guitar Flash	2,244	481	6	10,856	135	5
Calculator	2,850	737	8	12,898	180	5
iFixit	3,423	797	10	12,365	161	5
Slots Pharaohs Fire	3,148	787	12	18,421	218	6
TypoLab	4,242	834	13	22,782	252	8
Cute Animals	1,221	455	5	23,834	273	9
Dolphin EMU	2,867	786	12	28,290	390	12
BBC Weather	6,307	1,390	23	37,503	374	12
Bike Citizens Bicycle GPS	6,948	1,455	26	40,444	431	16
The Child of Slendrina	3,047	838	20	39,748	436	14
Moto Rider	4,880	1,369	28	44,279	460	14
Doodle Army	10,336	1,742	61	48,903	520	16
BBC News	9,461	1,727	67	49,139	570	16
Adobe Lightroom	n/a	1,408	n/a	47,304	503	21

Table 5.1: Comparing Memory Usage and analysis time between Soot and Jitana

memory than Soot to complete the analysis for each application. In the case of *Snake*, our smaller app, Soot needs over 5 times more memory to analyze 7.6 times fewer methods. In the case of *Cute Animals*, Soot needs 67% more memory to analyze 19 times fewer methods. On average, Soot needs 2.07 times more memory to analyze an app.

Next, we report memory efficiency (ME) in Table 5.2. As a reminder, ME is the ratio between the number of methods in a method call graph and peak memory usage.

As the table shows, ME of Jitana is significantly higher than that of Soot. In many cases, the efficiency gain is as high as 249 times. On average, the ME of Soot is 3.87 methods per one MB, while the ME of Jitana is 79.98 methods per one MB. The average efficiency gain of Jitana over Soot is 36.35 times.

App	ME_{Soot}	ME_{Jitana}	Gain ($\frac{ME_{Jitana}}{ME_{Soot}}$)
Snake	0.65	34.62	53.57
Battery Indicator	3.62	55.44	15.31
BitClock	0.33	81.25	249.44
AdBlockPlus	2.16	62.93	29.17
Guitar Flash	4.67	80.42	17.24
Calculator	3.87	71.66	18.53
iFixit	4.29	76.80	17.88
Slots Pharaohs Fire	4.00	84.5	21.13
TypoLab	5.09	90.40	17.77
Cute Animals	2.68	87.30	32.53
Dolphin EMU	3.65	72.54	19.89
BBC Weather	4.54	100.28	22.10
Bike Citizens Bicycle GPS	4.78	93.84	19.65
The Child of Slendrina	3.64	91.17	25.07
Moto Rider	3.56	96.26	27.00
Doodle Army	5.93	94.04	15.85
BBC News	5.48	86.21	15.74
Average	3.70	79.98	36.35

Table 5.2: Comparing Memory Efficiency Between Soot and Jitana

5.2 RQ2: Efficiency

In term of analysis time, Table 5.1 shows that for the small size apps (i.e., *Snake* to *Guitar Flash*), the analysis times of Soot and Jitana are comparable. However, as the apps become large, Soot spends more time to analyze these apps. The two exceptions are *Cute Animals* and *Dolphin EMU*. These are small apps; however, they invoke a huge number of methods from the underlying framework. Small numbers of methods allow Soot to perform analysis quickly. However, Jitana ends up analyzing more than 19 times and 9 times more methods in these apps, respectively. Jitana takes longer than Soot to analyze *Cute Animals*, and it takes about the same time as Soot to analyze *Dolphin EMU*. For very large apps such as Doodle Army and *BBC News*, the analysis times of Soot are 2.81 to 3.18 times higher than those of Jitana.

While the peak memory consumption can provide the highest memory watermark for an analysis system, it does not provide the amount of memory the analysis system is using concerning time. For example, a system that occupies a large amount of memory for a

short time may perform better than another system that uses less memory but for a much longer time. To observe memory usage over time, we recorded both Jitana's and Soot's memory usage throughout execution. We plot time (in seconds) on the x-axis and the memory consumption on the y-axis to form the memory-over-time graphs. Here we discuss the results of three apps that are most representative of the entire range of apps tested.

By the sizes of their DEX files, *BBC News* (4.3 MB), *Dolphin EMU* (2.0 MB), and *Cute Animals Names and Sounds* (1.3 MB) represent large, medium, and small Android apps. Again, when mentioning the size of Android apps, we are referencing the relative logical complexity, reflected by the size of DEX files or the number of lines of code. For example, *Cute Animals* has the largest APK size but the smallest DEX size, meaning that most of the files in its APK are not related to how the application works; instead, they are accessory files, such as media, used to support the application's functions.

We intend to analyze how SOOT and JITANA perform in analyzing real-world apps of varying complexities. For a small app, SOOT and JITANA achieve the same time of completion; however, the amount of memory needed by JITANA reaches plateau much quicker. It also requires less peak memory. For a medium-sized app such as Dolphin EMU, JITANA manages to use around 300 MB of memory during most of the analysis, while SOOT's memory usage continued to increase to almost 800 MB.

The performance difference in memory usage becomes even more staggering when both systems analyze large apps. *BBC News*, whose APK has multiple DEX files, represents a reasonably complex Android app that an end-user would encounter in real life. As Figure 5.3 shows, JITANA outperforms SOOT by a big margin, both in terms of speed and memory usage.

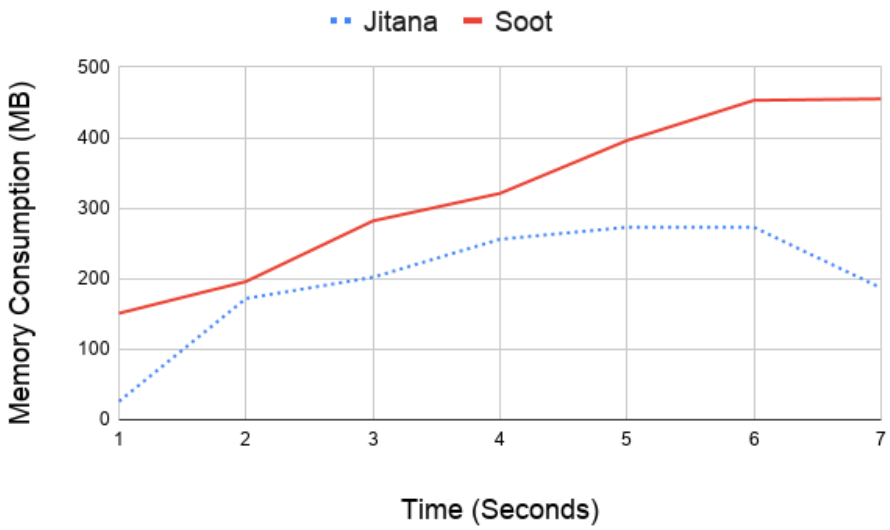

Figure 5.1: *Memory Usage Over Time for Analyzing a Simple App.*

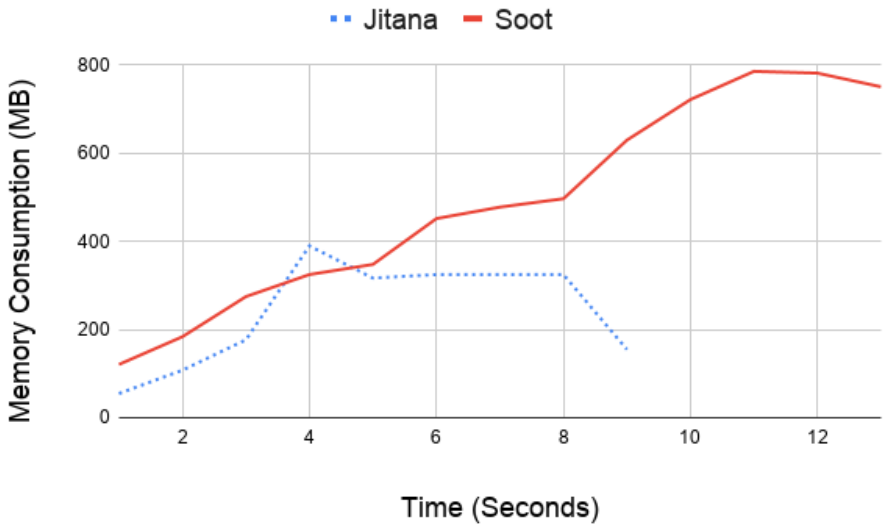

Figure 5.2: *Memory Usage Over Time for Analyzing a Medium-Sized App.*

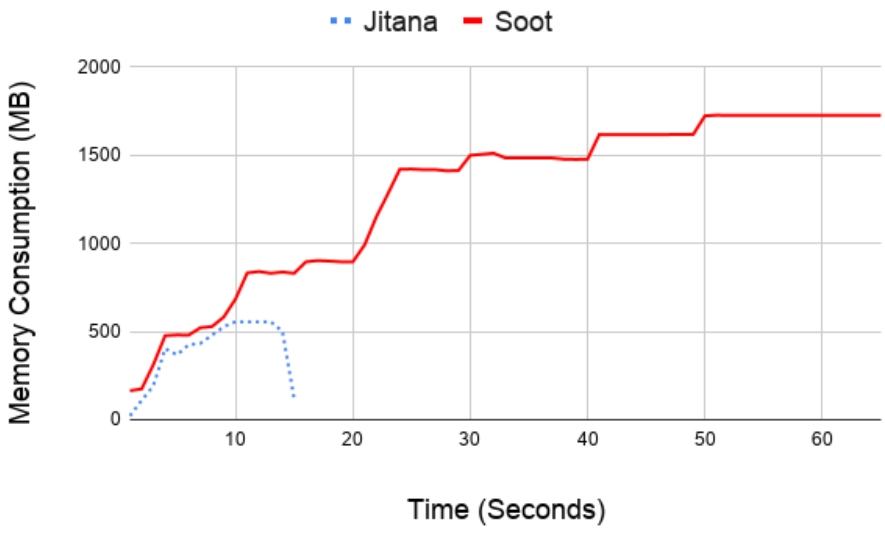

Figure 5.3: *Memory Usage Over Time for Analyzing a Complex App.*

Chapter 6

Conclusions

In this study, we analyzed the advantages of the class-loading approach in static analysis over the traditional compiler-based approach. We also designed experiments to measure the memory efficiency and performance of Soot and Jitana for constructing method call graphs across a wide range of Android apps. Our results show that Jitana conserves up to 5.2 times memory and achieves an average efficiency gain of 36.35 times over Soot.

As the average size of smartphone apps grows, tools based on the class-loading approach like Jitana will scale more easily than compiler-based approaches like Soot. Jitana also made it feasible to analyze methods from both the application itself and the underlying Android framework, thereby achieving a more complete analysis. In general, the compiler-based approach is even more inefficient for analyzing the Android framework code, since it would have to load the entire codebase, only to analyze a relatively small number of reachable classes within the framework. Though it is theoretically possible to analyze both the application and the framework code with Soot, the memory overhead would make it infeasible on most personal computers. Therefore, we conclude that the class-loading approach, represented by Jitana, is more memory-efficient and scalable for performing static analysis of today's complex applications.

Bibliography

[1] R. Vallée-Rai, "Soot: A Java Bytecode Optimization Framework," Master's thesis, McGill University, 2000.

[2] J. Späth and P. Lam, "Using Soot and TamiFlex to analyze DaCapo," August 2014, https://github.com/Sable/soot/wiki/Using-Soot-and-TamiFlex-to-analyze-DaCapo.

[3] J. Abraham, P. Jones, and R. Jetley, "A formal methods-based verification approach to medical device software analysis," February 2010, https://www.embedded.com/a-formal-methods-based-verification-approach-to-medical-device-software-analysis/.

[4] "lint," http://tools.android.com/tips/lint, 2019.

[5] A. Desnos, "Androguard: Reverse Engineering, Malware and Goodware Analysis of Android Applications," https://github.com/androguard/androguard, 2013.

[6] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst, "Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15, Lincoln, NE, USA, November 2015, pp. 669–679.

[7] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "DroidRA: Taming reflection to support whole-program analysis of android apps," in *Proceedings of the 25th International*

Symposium on Software Testing and Analysis, ser. ISSTA 2016, Saarbrucken, Germany, 2016, pp. 318–329.

[8] D. Wu, X. Liu, J. Xu, D. Lo, and D. Gao, "Measuring the declared sdk versions and their consistency with api calls in android apps," in *Wireless Algorithms, Systems, and Applications*, L. Ma, A. Khreishah, Y. Zhang, and M. Yan, Eds. Cham: Springer International Publishing, 2017, pp. 678–690.

[9] T. Jim, "Legacy C/C++ code is a nuclear waste nightmare that will make you WannaCry," http://trevorjim.com, June 2017.

[10] B. Chandra, "A technical view of the open ssl heartbleed vulnerability," https://www.ibm.com/developerworks/community/files/form/anonymous/api/library/38218957-7195-4fe9-812a-10b7869e4a87/document/ab12b05b-9f07-4146-8514-18e22bd5408c/media, May 2014.

[11] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proceedings of the International Conference on Software Engineering*, May 2011, pp. 241–250.

[12] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. V igna, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications." in *Proc. of NDSS*, vol. 14, San Diego, CA, 2014, pp. 23–26.

[13] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci, "Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '15, San Antonio, Texas, USA, 2015, pp. 37–48.

[14] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques," in *Proc. of NDSS*, 2016.

[15] Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang, "Things You May Not Know About Android (Un)Packers: A Systematic Study based on Whole-System Emulation," in *Proceedings of Network and Distributed System Security Symposium, NDSS*, San Diego, California, USA, February 2018.

[16] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 627–638. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046779

[17] L. Xu, "Techniques and tools for analyzing and understanding android applications," Ph.D. dissertation, UNIVERSITY OF CALIFORNIA, DAVIS, 2013.

[18] B. Davis and H. Chen, "Retroskeleton: Retrofitting android apps," in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '13. New York, NY, USA: ACM, 2013, pp. 181–192. [Online]. Available: http://doi.acm.org/10.1145/2462456.2464462

[19] S. Liang, M. Might, and D. V. Horn, "Anadroid: Malware analysis of android with user-supplied predicates," *CoRR*, vol. abs/1311.4198, 2013.

[20] P. Ponomariov, "Shedun: Adware/malware family threatening your Android device," September 2015, https://blog.avira.com/shedun/.

[21] A. M. Memon and A. Anwar, "Colluding apps: Tomorrow's mobile malware threat," *IEEE Security and Privacy*, vol. 13, no. 6, pp. 77–81, November 2015.

[22] R. Chirgwin, "Uk universities, mcafee collude to beat collusion attacks," http://www.theregister.co.uk/2014/02/27/uk_unis_mcafee_collude_to_beat_collusion_attacks/, February 2014.

[23] R. Vallee-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing java bytecode using the soot framework: Is it feasible?" in *In International Conference on Compiler Construction, LNCS 1781*, 2000, pp. 18–34.

[24] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon, "Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot," in *ACM Sigplan International Workshop on the State Of The Art in Java Program Analysis*, June 2012.

[25] Y. K. Lee, J. y. Bang, G. Safi, A. Shahbazian, Y. Zhao, and N. Medvidovic, "A sealant for inter-app security holes in android," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, Buenos Aires, Argentina, 2017, pp. 312–323.

[26] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "COVERT: compositional analysis of android inter-app permission leakage," *IEEE Trans. Software Eng.*, vol. 41, no. 9, pp. 866–886, 2015. [Online]. Available: http://dx.doi.org/10.1109/TSE.2015.2419611

[27] Oracle Corp, "Loading, Linking, and Initializing," November 2019, https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html.

[28] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[29] A. Hern, "Stagefright: New android vulnerability dubbed "Heartbleed for Mobile"," 2015, the Guardian.

[30] AndroidCentral, "Phone Died During System Update," 2013, http://forums.androidcentral.com/htc-desire-c/265098-phone-died-during-system-update.html.

[31] Z. Epstein, "Did Apps Just Start Crashing Constantly on Your Android Phone?" 2015, http://bgr.com/2015/04/28/android-tips-tricks-fix-crashing-apps/.

[32] T. McDonnell, B. Ray, and M. Kim, "An Empirical Study of API Stability and Adoption in the Android Ecosystem," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ser. ICSM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 70–79. [Online]. Available: https://doi.org/10.1109/ICSM.2013.18

[33] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How Do API Changes Trigger Stack Overflow Discussions? A Study on the Android SDK," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014, Hyderabad, India, 2014, pp. 83–94.

[34] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "Api change and fault proneness: A threat to the success of android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, Saint Petersburg, Russia, 2013, pp. 477–487.

[35] T. Luo, J. Wu, M. Yang, S. Zhao, Y. Wu, and Y. Wang, "Mad-api: Detection, correction and explanation of api misuses in distributed android applications," in *Artificial Intelligence and Mobile Services – AIMS 2018*, M. Aiello, Y. Yang, Y. Zou, and L.-J. Zhang, Eds. Cham: Springer International Publishing, 2018, pp. 123–140.

[36] H. Huang, L. Wei, Y. Liu, and S.-C. Cheung, "Understanding and detecting callback compatibility issues for android applications." in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 532–542.

[37] D. He, L. Li, L. Wang, H. Zheng, G. Li, and J. Xue, "Understanding and detecting evolution-induced compatibility issues in android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 167–177.

[38] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2013, pp. 70–79.

[39] L. Wei, Y. Liu, and S.-C. Cheung, "Taming android fragmentation: Characterizing and detecting compatibility issues for android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 226–237.

[40] D. Wu, X. Liu, J. Xu, D. Lo, and D. Gao, "Measuring the declared sdk versions and their consistency with api calls in android apps," in *International Conference on Wireless Algorithms, Systems, and Applications*. Springer, 2017, pp. 678–690.

[41] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "Cid: Automating the detection of api-related compatibility issues in android apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: ACM, 2018, pp. 153–163. [Online]. Available: http://doi.acm.org/10.1145/3213846.3213857

[42] S. Scalabrino, G. Bavota, M. Linares-Vásquez, M. Lanza, and R. Oliveto, "Data-driven Solutions to Detect API Compatibility Issues in Android: An Empirical Study," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR '19, Montreal, Quebec, Canada, 2019, pp. 288–298.

[43] D. Landman, A. Serebrenik, and J. Vinju, "Challenges for static analysis of java reflection – literature review and empirical study," in *Proceedings of the International Conference on Software Engineering*, Buenos Aires, Argentina, May 2017.

[44] B. Livshits, J. Whaley, and M. S. Lam, "Reflection analysis for java," Tech. Rep., 2005.

[45] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting inter-component privacy leaks in Android apps," in *Proceedings of the 37th ACM/IEEE International Conference on Software Engineering*, 2015, pp. 280–291.

[46] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "I know what leaked in your pocket: Uncovering privacy leaks on Android apps with static taint analysis," *CoRR*, vol. abs/1404.7431, 2014.

[47] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon, "Effective inter-component communication mapping in Android: An essential step towards holistic security analysis," in *USENIX Security Symposium*, 2013, pp. 543–558.

[48] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11, 2011, pp. 239–252.

[49] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, and Y. L. Traon, *ICT Systems Security and Privacy Protection: 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings*. Cham: Springer International Publishing, 2015, ch. APKCombiner: Combining Multiple Android Apps to Support Inter-App Analysis, pp. 513–527.

[50] Y. Tsutano, S. Bachala, W. Srisa-an, G. Rothermel, and J. Dinh, "An efficient, robust, and scalable approach for analyzing interacting android apps," in *Proceedings of the International Conference on Software Engineering*, Buenos Aires, Argentina, May 2017.

[51] GeeksforGeeks, "ClassLoader in Java," https://www.geeksforgeeks.org/classloader-in-java/, May 201r.

[52] Boost.org, "Boost C++ Library," March 2018, http://www.boost.org/doc/libs/develop/libs/graph/doc/

[53] R. Wiśniewski and C. Tumbleson, "Apktool - A tool for reverse engineering Android apk files," https://ibotpeaches.github.io/Apktool/.

[54] "F-Droid," https://f-droid.org, last Accessed: 2020-3-15.

[55] "Download APK free online downloader | APKPure.com," https://apkpure.com/, last Accessed: 2017-5-23.

[56] "Download APK free online downloader | APKPure.com," https://apkpure.com/, last Accessed: 2020-3-15.

[57] "Analyze your build with APK Analyzer | Android Developers," https://developer.android.com/studio/build/apk-analyzer, last Accessed: 2020-3-7.