ReHAna: An Efficient Program Analysis Framework to Uncover Reflective Code in Android

Shakthi Bachala[1], Yutaka Tsutano[1], Witawas Srisa-an[1](✉), Gregg Rothermel[2], Jackson Dinh[1], and Yuanjiu Hu[1]

[1] School of Computing, University of Nebraska-Lincoln, Lincoln, NE 68588, USA
{sbachala,ytsutano,jdinh,yhu}@cse.unl.edu, witawas@unl.edu

[2] Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA
gerother@ncsu.edu

Abstract. The recent adoption of dynamic features such as Java reflection and Android dynamic code downloading (RDCL) coupled with recent security attacks that can be detected only at runtime have led to higher usage of hybrid analysis to address dependability and security concerns. While effective, however, hybrid analysis can be inefficient due to a multi-step process involving static analysis, code instrumentation, and runtime information logging. As such, existing hybrid analysis techniques can work during code development and testing, but are too slow for production and security vetting.

In this paper, we introduce ReHAna, a hybrid analysis framework for Android apps. We designed our framework to perform hybrid analysis efficiently through the use of a Virtual Class-Loader (VCL), which enables incremental program analysis. We then conducted a study to assess the program analysis performance of using VCL and found that it yields several benefits over the existing compiler-based program analysis approach. We also illustrated the hybrid analysis capability of ReHAna by implementing a technique to detect and analyze dynamically loaded components based on reflection and dynamic code loading mechanisms in Android apps. We compared the performance of ReHAna against that of StaDynA, a hybrid analysis approach that performs the same task. Our empirical evaluation shows that ReHAna is as effective as StaDynA but also significantly more efficient and scalable.

Keywords: Program analysis · Android · Java reflection

1 Introduction

Researchers and practitioners have used static program analysis to enhance software quality, dependability, and security over the last few decades. The main idea

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2022
Published by Springer Nature Switzerland AG 2022. All Rights Reserved
T. Hara and H. Yamaguchi (Eds.): MobiQuitous 2021, LNICST 419, pp. 347–374, 2022.
https://doi.org/10.1007/978-3-030-94822-1_19

of static program analysis is to analyze software (source code, intermediate representation code, or binary code) without executing programs [1,2,9,13,16,27,29]. The first step to perform static program analysis is loading a 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, it analyzes the *complete* code to produce the results that *cannot change* [3]. 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 better address emerging security and dependability issues. Below, we highlight two existing challenges due to these changes.

Challenge 1. Modern programming languages and platforms provide rich library support. Therefore, loading just the application code alone may not be sufficient to ensure the dependability and security of an application. Recently, we have seen various instances of security vulnerabilities and software defects that affect a large number of computer systems worldwide [5,14,30]. 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, compiler-based analysis techniques can 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 work as RDCL). These mechanisms can load new classes at runtime, possibly from external sources [3,21,23,32]. Recently, we have seen RDCL used to deliver malicious payloads in highly elusive malware [8,10,11,18,22,31]. *When a compiler-based approach analyzes apps with RDCL, it cannot provide complete coverage because the application can add more classes to the existing code-base. These new classes make the initial static analysis results incomplete, creating the need to perform analysis on the entire code-base once again, even for a small code change* [3,32].

These challenges illustrate the inherent limitations of program analysis approaches that make the closed-world assumption. In this work, we address these two challenges by eliminating the "closed-world" assumption, and instead, rely on the "open-world" assumption achieved through the notion of "incrementality". Over the past two decades, there have been several advancements in *Java Virtual Machine* (*JVM*) and *Android Virtual Machine* (*AVM*) that make incremental static program analysis feasible.

The first advancement is the 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 needed class for execution. It also supports various forms of late binding to quickly resolve the ambiguity of finding which

class to load through delegation. The second advancement is the Just-In-Time (JIT) compiler. 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 [26]) and then performs optimization to generate the backend code. With a JIT compiler and a class-loader, it is possible to analyze each class right after it is loaded incrementally.

In this work, we propose ReHAna, a new incremental, hybrid program analysis framework, to address these two critical challenges. The key idea to support incremental analysis is by virtualizing the class-loading operations to load the program code on a per-class basis. We create a Virtual Class-Loader (VCL) to accomplish this task. Our *VCL* fully adheres to the published class-loader specification [20], but it operates as a stand-alone component in our framework; i.e., it does not operate as part of a language runtime system such as an Android Virtual Machine (AVM). The VCL uses reachability analysis to uncover classes that must be loaded and delegates to resolve all possible statically discoverable late binding targets. By employing VCL, we eliminate the need to load the entire project in one shot.

As our framework loads each class, it analyzes all the methods within that class to uncover more methods and classes to which these methods belong. This approach mimics a method-level JIT compiler that analyzes methods belonging to each loaded class. Our incremental analysis can generate class-loader, class, intraprocedural control-flow and data-flow, and method-call information as hierarchical graphs. It is important to note that our class-loading and analysis mechanisms do not distinguish between the application classes and the ADF classes so that it can include relevant classes in the ADF as part of an analysis effort. This capability addresses Challenge 1.

Because our VCL operates like an actual class-loader inside an AVM, it can also operate as a shadow class-loader to perform dynamic and hybrid analyses (i.e., an analysis approach that combines static and dynamic analysis results [4,21,24,28]). Through a connection with an AVM, ReHAna can observe relevant runtime events and generates essential program analysis information incrementally while the app is still running, allowing it to hide the analysis cost by interleaving it as part of the execution. This capability addresses Challenge 2.

We then investigate several research questions. First, we wish to identify and assess the performance differences between closed-world and open-world static analysis approaches. In this study, we compared the static analysis performance of ReHAna with that of Soot, a widely used program analysis framework for Java and Android. Next, we used ReHAna to analyze apps with RDCL and compare its effectiveness, efficiency, and scalability with those of StaDynA, a state-of-the-art approach in analyzing apps with RDCL. The results of our investigation reveal significant performance benefits of using ReHAna over Soot. Our results also indicate that ReHAna is as effective as StaDynA. However, it can cut down analysis time by as much as 50 times.

We organize the rest of this paper as follows. Section 2 provides the background information related to RDCL. Section 3 describes VCL and ReHAna.

Section 4 elaborates on our experimental designs. Section 5 reports the results of our investigation. Section 6 discusses our observations concerning the reported results. Section 7 highlights prior works that are closely related to ours. We conclude this paper in Sect. 8.

2 Background and Motivation

In this work, we use our proposed ReHAna framework to create an efficient hybrid analysis approach to detect and analyze RDCL components. As such, we spend part of this section describing the underlying mechanism of RDCL. We also describe the mechanism by which hybrid analysis detects RDCL and discuss an existing state-of-the-art hybrid analysis technique for identifying RDCL components in Android applications. We then assess the cost of performing a closed-world analysis.

2.1 Reflection and Dynamic Code Loading (RDCL)

RDCL is commonly used to support several features in Android apps. These features include backward compatibility, dynamic updates, and component plugins. More recent usage of RDCL is to serve advertisements. Next, we describe how RDCL can be used to support these features.

An Android app is compiled to Dalvik Executable code (*dex code*) and then stored in the main dex file. The distinguishing feature of RDCL, where Android apps are concerned, is that the dynamically loaded classes are not part of the main dex file. There are at least two mechanisms for accessing classes in these external dex files at runtime. First, additional Dex files can already be inside the APK. Facebook uses this mechanism to overcome an Android limitation of having only a 16-bit integer to represent a method id, whereas Shedun uses this approach to hide malicious intents [22]. Another approach is to download additional dex files from remote servers. Software update features and serving advertisements are two examples that use this approach. We have also seen malware that uses this mechanism to deliver malicious components.

For example, through our analysis of downloaded viruses, we found particular adware that is a repackaged version of the *ispconfig* app, and that makes several attempts to download files from remote servers. One particular web-link (update.topapk.mobi) is reported to be malicious by virusshare.com. This particular downloaded file contains a custom class loader that can load more classes from downloaded dex files. They then use Java reflection to invoke methods to display advertisements that can link to a malicious website (when we tried to run the app, we found that the destination link was not active).

To further assess the extent to which Android apps use RDCL, we collected 60 apps from three major app categories (games, social network, and multimedia/miscellaneous) from Google Play in 2018. These apps included the top 20 games (e.g., NBA Live, Roblox), the top 20 social network apps (e.g., Facebook, Pinterest), and the top 20 miscellaneous apps (e.g., Pandora, Spotify). We performed a reflection analysis to statically determine whether each of these apps

can dynamically load code by identifying the presence of RDCL call sites. Our analysis reveals that 34 out of 60 apps (57%) can use RDCL, which is a significant increase from an earlier reported result, wherein only 16% of the top 50 apps from 2013 used RDCL [21].

2.2 Hybrid Analysis Approaches for Detecting Dynamic Code Loading in Android Apps

The prevalence of RDCL motivates the need for a practical approach for efficiently and scalably detecting the use of RDCL and dynamically capturing downloaded components. In this work, we propose a framework to accomplish these goals. However, before we discuss our proposed framework, we point out some of the shortcomings of the current state of the art hybrid analysis approaches that help engineers and analysts effectively analyze Android apps that use RDCL.

As mentioned in Sect. 1, both TamiFlex and StaDynA use dynamic analysis to discover classes dynamically loaded via reflection and DCL, to increase the soundness of static analysis results. However, TamiFlex does not work on Android apps due to the lack of support for load-time instrumentation in the Android framework. As such, we turn our attention to StaDynA [32], a system that performs dynamic analysis (Phase 1) by modifying Dalvik, an Android VM, to extract the necessary runtime information. There are two major components in StaDynA: a client and a server.

Client. The client component is the modified Dalvik VM. Zhauniarovich et al. [32] modified the `libcore` component of the VM to capture events related to opening dex files. They also modified the invoke method to obtain class, method, and parameter information (e.g., paths with which to retrieve dynamically loaded classes and method names). Each invocation event is also accompanied by a stack trace so that an RDCL call site's information can be determined (e.g., which call site is responsible for the invocation). The logcat mechanism in Android sends information to the server component.

Server. The server component is responsible for constructing and annotating MCGs. According to Zhauniarovich et al. [32], the server component can support different program analysis engines. To date, Androguard[1] has been used as the analysis engine to construct MCGs. In the first step, Androguard analyzes an app to determine whether any RDCL call site exists within the app. If the app contains any RDCL call site (each call site is referred to in Reference [32] as a Method of Interest or MoI), it is installed onto a real device or an Android Virtual Device (AVD). The server component also statically generates the MCG of that app. Note that in this step, RDCL call sites appear as terminals in the initial MCG. Next, random event sequence generation tools such as Monkey are used to exercise the app.

[1] Available at https://github.com/androguard/androguard.

As previously mentioned, the client component monitors events related to class-loading and method invocations. This information is then sent to the server component by the client through the logcat mechanism. The server component selects only events related to MoIs and then copies the dynamically loaded classes from the client component and stores them for further processing. At the end of the run, it annotates the initial MCG with additional method calls. This step, in effect, concludes the dynamic analysis portion or Phase 1 of the RDCL analysis.

In Phase 2, the MCG built during Phase 1 is used by static analysis engines to perform more complex analyses (e.g., generating control flow, data flow, and points-to graphs). As noted earlier, the authors of Reference [32] did not conduct any evaluation of StaDynA concerning this phase. In the case of TamiFlex, however, Soot was used to perform Phase 2 by statically analyzing previously-stored dynamically loaded classes and then transforming these class files to produce a program that replaces reflective method calls with standard Java method calls. It then performs further analyses of the transformed program to construct objects such as control flow and data flow graphs.

Note further that as TamiFlex transforms programs, it also uses instrumentation capabilities in Soot to insert runtime checks to ensure that execution of any MoIs that have not been captured in the prior runs would provide warning messages to the user. These messages indicate that there are new MoIs that have not been previously exercised, and the user can conduct the Phase 1 analysis again to capture runtime information about these MoIs, and use this to further annotate the MCG. Any change to the MCG requires that Phase 2 analysis be reperformed to transform the newly captured methods and reconstruct various program analysis graphs.

In the preceding context, repeating Phase 2 is necessary because commonly used program analysis engines, including Soot and Androguard, operate in a closed-world fashion. That is, before performing analysis, information that needs to be analyzed (e.g., code for all methods to be analyzed) must be available. For example, if an analysis chooses to ignore any RDCL call sites, all it needs to proceed further is statically discoverable methods. Later, if the same analysis is extended to include dynamically loaded classes, then a transformed program that includes dynamically discovered method calls would be needed. Any subsequent changes to the annotated MCG (e.g., an input sequence that exercised one or more previously uncovered MoIs or an RDCL target is replaced with a new version) would require the complete analysis to be repeated in order to account for these changes.

3 Approach

This section describes ReHAna, our proposed VCL-based hybrid program analysis framework. We first describe the proposed VCL and then how we integrate it into ReHAna. As a hybrid-analysis framework, ReHAna can perform both static and dynamic analyses, and when applicable, it can integrate the analysis results. Next, we explain the essential operation of the Virtual Class-Loader (VCL), which is the critical component of ReHAna.

3.1 Virtual Class-Loader

As previously mentioned, the runtime concepts such as 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 [20] supports *Delegation Hierarchy Principle* that is used to discover and load classes through delegation [12]. 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 follows the Java classloading specification from Oracle [20]. 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, VCL applies the *Delegate Hierarchy Principle* to locate and load that class. This capability enables ReHAna 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. 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.

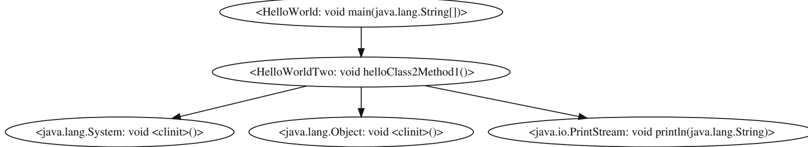

Fig. 1. A method-call-graph of *HelloWorld* produced by Soot.

To illustrate the operational difference between compiler-based and VCL-based analysis, 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 1 illustrates the method call graph produced by Soot, a compiler-based framework, and Fig. 2 illustrates the method call graph produced by our VCL-based approach. 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 the VCL-based analyzer was compiled into Android Dex code.

As shown in Fig. 1, Soot considers any calls to the underlying library (e.g., `java/lang/Object` and `java/io/PrintStream`) as terminals. That is, the analysis ends at these calls. The VCL-based approach, on the other hand, continues 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 Fig. 2, represents the actual call from class `HelloWorld` to a method in class `HelloWorldTwo`.

The graph generated by the VCL-based approach 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. They are also not invoked but included because our approach analyzes all the Dex methods in every loaded class as part of the reachability analysis to ensure completeness.

Also note that our 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 as an instruction in the instruction graph (i.e., a list of instructions in a

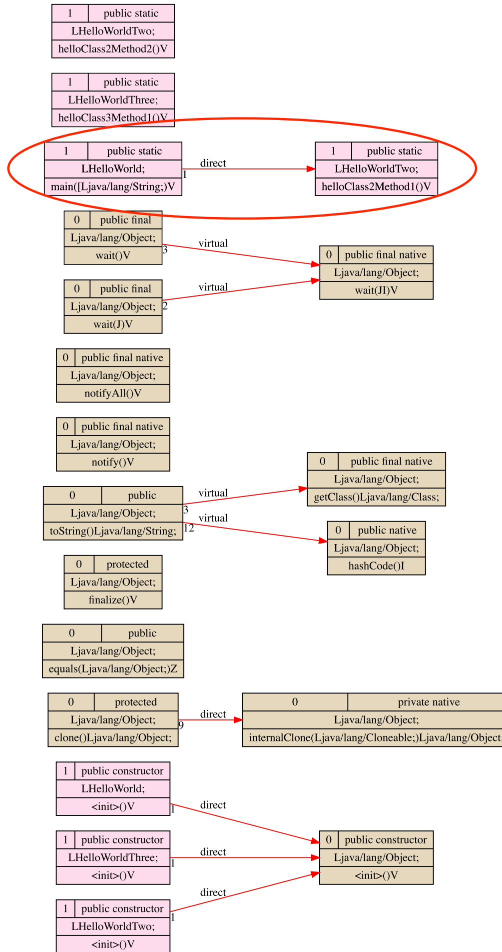

Fig. 2. A method-call-graph of *HelloWorld* produced by VCL-based analysis.

method) of `helloWorldClass2Method1`. Next, we describe how we utilize *VCL* to design ReHAna.

3.2 Introducing ReHAna

Imagine a scenario where an analyst is utilizing a compiler-based static program analysis as part of a hybrid analysis to detect and analyze RDCL components. The analysis process would involve performing the static analysis (STA_1) to create a method-call-graph (MCG) and identify methods that use RDCL functionality. The process continues with running the program using an existing input generator approach to perform dynamic analysis (DYN). If it needs more detailed analysis information after DYN, another round of static analysis

(STA_2) that also considers the code of the newly detected RDCL components is needed to generate the new MCG.

The goal of ReHAna is to *reduce analysis overhead by removing the cost of performing the static analysis (STA_2) from the hybrid analysis process.* That is, our approach performs STA_1, and then DYN becomes a *continuous analysis* process wherein the cost of STA_2 is *proportional* to the size of the code change and is *hidden* as part of program execution time. As such, the process of generating detailed information should take as much time as the time needed to run the program during dynamic analysis.

To perform the continuous analysis, we exploit a general but essential characteristic of event-based apps, namely, that there are typically delays between events. These delays allow us to export information from an Android device to another workstation that performs the analysis. To accomplish this goal, we modified the Android VM to allow us to capture runtime events that occur as an app executes. At least two other techniques have modified the Android VM to help with RDCL analysis [21,32]. However, these techniques still log runtime events and perform static analysis in a separate off-line phase. This is because the analysis engines that they use to perform analysis, as previously mentioned, operate by using a closed-world approach. Our approach, on the other hand, performs analysis on-the-fly.

Figure 3 provides a conceptual view of ReHAna when it is used to analyze an app that downloads classes from a remote server. Specifically, the figure shows how components within the framework are connected. It depicts an app running on an Android tablet with our modified Android VM. As it executes, dynamic information is generated (e.g., the app downloads a class from a remote server, a basic block is executed, or an app sends a request to another app) and processed by ReHAna.

The static analysis component of ReHAna is based on VCL, which has been designed to operate natively on Dex code. It produces various program analysis context data that includes a class-loader graph, class graph, method-call graph, and instruction graphs. The class-loader graph contains class-loader information that can be used to identify the defining class-loader for a class. The class graph contains all reachable classes that have been loaded for analysis. This includes both reachable application classes and ADF classes. The Method-call graph contains method-call relationships. Each instruction graph contains all instructions in a method. It also includes intraprocedural control-flow and data-flow information.

For ReHAna to take runtime information from an Android VM, we created interfaces that would allow ReHAna to receive information from the VM and send information to control the VM. To do this, we relied on the *Java Debug Wire Protocol* (*JDWP*) over *Android Debug Bridge* (*ADB*). We then designed the data structures maintained by ReHAna to match those maintained by the Android VM so that incoming information can be readily processed without the need to perform conversions. For example, both ReHAna and the Android VM (Dalvik or ART) maintain the same data structures to record classes that have

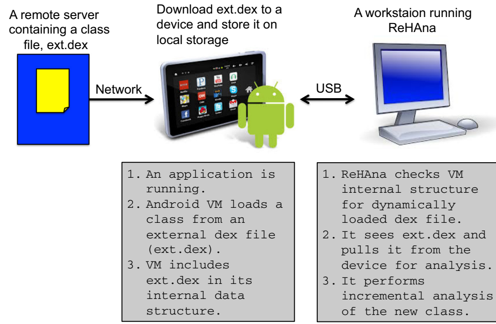

Fig. 3. Conceptual view of ReHAna.

already been loaded. As such, when a new class is loaded through reflection, for example, ReHAna specifically queries for information that includes the class name, method index, and class location to ensure precision.

We also discovered that in several instances, after a device downloads a class and the Android VM loads it, the original dex file for that class is deleted. This is a common practice when serving advertisements. This scheme can also be used to hide malicious content that has been dynamically downloaded. To prevent class files from being deleted, we cached dex files that have been downloaded as part of class loading; ReHAna then pulls any new class from the device so that it has an exact copy on the analysis workstation, which performs on-the-fly analysis to create the necessary information. The result is then added to the existing analysis graphs.

Note that ReHAna does not require that apps be instrumented. Runtime events are captured as they happen, so analysis can be performed without interruption. In all, we required about 500 lines of code to implement these features in both Dalvik and ART.

4 Evaluation

We want to assess the performance differences between the closed-world and open-world analysis. Thus, our evaluation compared the static analysis capability between Soot and ReHAna. We observed the number of methods analyzed by each approach. The amount of committed memory and analysis time during an analysis attempt. To investigate the effectiveness, efficiency, and scalability of ReHAna to perform hybrid analysis, we implemented a version of the framework to perform hybrid analysis of apps that utilize reflection and dynamic code loading (RDCL). To do this, we took the same approach as StaDynA, wherein our modified VM captures relevant events related to RDCL (e.g., construction of reflective classes and invocation of reflective methods). We then conducted a study comparing the static analysis results produced by the two systems. Next, we conducted dynamic analyses to assess ReHAna's effectiveness against that

of StaDynA. We also compared the total hybrid analysis times of the two approaches. Last, we evaluated the scalability of ReHAna by using real-world apps downloaded from the Play Store.

To conduct our investigations, we requested access to the source code from the authors of StaDynA, and they kindly granted it. We then successfully ported the modifications to the source tree of Android API 19 so that we could use the Dalvik VM's most advanced implementation. We, however, were not able to port the modifications to the later versions of Android. This issue was due to the dynamic analysis component of StaDynA that can only run on the older Android VM (i.e., Dalvik VM).

Next, we implemented the dynamic analysis component of ReHAna on Android API 19 (KitKat), allowing us to evaluate StaDynA and ReHAna on the same API platform, using the same set of apps. While API 19 is quite dated, apps developed for this API can run on 90% of existing devices. Further, as shown later, we were able to run all of the selected Play Store apps on this platform without any issues. For long term usability, we are porting our implementation to run on Android API 29 (the latest stable version).

In this section, we describe our methodology to investigate the following Research Questions (RQs):

RQ1: What are the performance differences between Soot and ReHAna?
RQ2: Is ReHAna more effective than StaDynA?
RQ3: Is ReHAna more efficient than StaDynA?
RQ4: Is ReHAna more scalable than StaDynA?

4.1 Objects of Analysis

To answer all four RQs, we required apps that can run on ReHAna, Soot, and StaDynA. To compare the performances of Soot and ReHAna (RQ1), we randomly select a set of apps from Google's Play store. Our main criterion is to include apps of varying sizes that can run on the latest stable Android platform (Android 10). As part of this process, we collected 18 apps with APK sizes ranging from 18 KB to 80 MB.

To compare StaDynA with ReHAna (RQ2 to RQ4), we focused on collecting apps that can run on Android API 19. We began by requesting access to the apps used to evaluate StaDynA. The authors of StaDynA kindly provided the 10 apps they had used; these included five benign apps and five malware samples. We also downloaded a set of apps from Play Store that can run on both systems. We selected 20 apps from each of the three categories (social media, games, and miscellaneous). We then attempted to determine how many apps used RDCL and found that 34 of them did. At the end of this step, we had 44 apps.

We also need to be able to automatically and repeatably exercise any apps that we wish to study. To do this we experimented with *Monkey*,[2] a random event sequence generator and *UI Automator*,[3] a UI testing framework, in an

[2] https://developer.android.com/studio/test/monkey.html.
[3] https://developer.android.com/training/testing/ui-automator.html.

Table 1. Basic characteristics of the Android apps (provided by the authors of StaDynA and downloaded from Play Store) used in our study.

App	APK size (MB)	# of methods	App	APK size (MB)	# of methods
Play Store Apps–API 29 (18)					
Snake	0.018	44	Battery Indicator	2	2337
BitClock	0.57	4743	AdBlockPlus	2.6	5656
Guitar Flash	45.2	10608	Calculator	4.3	10623
iFixit	3.3	10323	Slots Pharaohs Fire	45.1	17066
TypoLab	45.2	21735	Cute Animals	45.2	25647
Dolphin EMU	13.8	28648	BBC Weather	9.2	40105
Bike Citizens Bicycle GPS	45.2	43604	The Child of Slendrina	45.1	43611
Moto Rider	45.1	48812	Doodle Army	45.5	62957
BBC News	15.5	65677	Adobe Lightroom	80.4	101022
StaDynA Apps (8)					
Fakenotify	0.72	618	Anserverbot1	0.79	2961
Basebridge4	1.5	3148	ImageView	1.1	7563
FlappyBird	0.92	10880	Avast	9.2	30887
Symantec	5.4	41544	ViberVOIP	20	42200
Play Store Apps–API 19 (13)					
GeometryJumpLite	42	52176	ooVoo	53	52927
LedFlashLight	5	55213	SoundCloud	33	55894
Zedge	12	56085	Hulu Plus	31	56432
Pinterest	17	56438	slither	19	57858
SgiggleProduction	31	58080	RollingSky	33	58297
Waze	48	58833	duoLingo	12	59186
Mercari	24	59389			

attempt to exercise the apps. While neither approach was able to provide high code coverage, we found that *Monkey* could cover slightly more code than UI Automator, particularly in exercising events to reach known MoIs. (This observation confirms a prior finding by Choudhary et al. [7].) As such, we used *Monkey* to generate event sequences for use in dynamic analysis.[4]

We configured *Monkey* to generate 10,000 events, using five different seeds to generate five random but distinct sequences. For each app, we ran each of the five sequences; each run took between 275 and 400 s. We chose the seed that exercises the most significant number of RDCL call-sites, as shown by StaDynA. For example, in the case of *RollingSky*, we had a seed that exercised six RDCL call sites and another that exercised seven RDCL call sites, so we chose the last seed.

We were able to use Monkey to exercise only eight of the 10 apps provided by the authors of StaDynA. Two malware samples, *DroidKungfu* and *smsSend*, no longer operated and therefore were not included. We also found that Monkey could exercise RDCL call sites in 15 of the 34 Play Store apps. However, one of

[4] We are currently experimenting with a different, recently published technique [6] to see if it can exercise the 19 apps for which Monkey could not reach any RDCL call sites. If we are successful, we will include results on these in the next version of this paper.

these apps, (*Snapchat*), crashed after about two thousand events, so we removed it. We also used StaDynA to analyze these apps and found that it cannot analyze *colorSwitch*, so we also removed it. At the end of this step, 21 apps remained viable for use in the study (8 from StaDynA and 13 from Play Store). We used the apps provided by the authors of StaDynA to answer the dynamic analysis portions of RQ2 and RQ3 and the 13 Play Store apps to answer RQ4. Using the 13 Play Store apps allows us to examine whether both approaches can analyze them and whether one is more scalable than the other. Table 1 summarizes the basic characteristics of all 39 apps.

4.2 Variables and Measures

Independent Variables. Typically, when studying a new program analysis technique, we choose the technique itself as our primary independent variable, locating one or more other techniques to compare against or use as baselines. To answer RQ1, we use FlowDroid as the baseline system. FlowDroid is a program analysis framework based on Soot that has been modified to support the 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. FlowDroid, an Android taint analysis tool built on top of Soot, solved 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 Soot and ReHAna, we set up both tools to generate only the method call graph (MCG) of a single app at a time. 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. In the remainder of this paper, we simply refer to FlowDroid as Soot.

To address RQ2 to RQ4, we performed the comparisons in a limited context. This is because StaDynA, the platform that works on Android apps, performs dynamic analysis (Phase 1) and then light-weight static analysis to construct the annotated MCG to include newly discovered components. It does not employ more complex static analyses to generate more analysis graphs. On the other hand, TamiFlex, the system that performs more complex static analysis after dynamic analysis, does not work for Android apps. We also considered Harvester but were not able to obtain it for this study. As such, the results reported for ReHAna include the times spent on constructing analysis graphs, whereas the results reported for StaDynA do not.

Dependent Variables. As dependent variables, we consider metrics that track memory usage, efficiency, effectiveness, overhead, and slowdown factor.

Memory Usage. We first measure the memory usage of ReHAna and FlowDroid by periodically monitoring the per-process memory usage reported in MB. We

are particularly concerned with peak memory usage, as it is the most common limiting factor in performing very complicated 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 reflects a higher memory efficiency. This metric is used to answer RQ1. We repeat the experiments three times and measure the amount of memory required to perform the analysis of each app using the analysis techniques, each averaged over three attempts.

Efficiency. We calculate efficiency by measuring the time required by the techniques to perform their analyses (e.g., $time_{\text{Soot}}$ and $time_{\text{ReHAna}}$). To do so, we measure the time in seconds required by each approach to perform analysis for each app in our collection with varying sizes. The measurement begins at the time we run an app and ends when the analysis result is reported. For RQ2 to RQ4, *the reported time does not include the cost of performing the initial static analysis used to identify MoIs.*

For StaDynA, the result reported for each app is the annotated MCG. For ReHAna, the results reported include various graphs generated during continuous analysis and the annotated MCG. Each reported result includes three overhead components, these being the time required to (i) execute an app, (ii) identify and capture dynamically loaded classes, and (iii) integrate these classes and produce the updated MCGs. We ran each measurement three times, and we report the average time in seconds.

Effectiveness. We assess effectiveness in two ways. First, we compare the results of using static analysis to determine MoIs using StaDynA and ReHAna. This is to determine whether our static analysis approach for identifying MoIs produces results similar to those of StaDynA. Second, we compare the number of RDCL targets that can be dynamically identified by ReHAna to the number dynamically identified by StaDynA.

To compare the detected RDCL targets, we begin by analyzing the report produced by StaDynA. The report lists the exercised RDCL method call sites (S_E), RDCL targets (S_T), and uncovered RDCL method call sites (S_U). Information in the report is at the method level. That is, if S_E contains more than one RDCL call statement, it is still treated as one method by StaDynA. ReHAna's report also lists the exercised RDCL method call sites (R_E) and detected targets (R_T). R_E, in this case, is reported at the statement level (i.e., each R_E is presented as a method and instruction index pair). However, we consider only method information, not instruction index information, to be consistent with StaDynA. Our analysis attempts to find cases in which $R_E \cap S_E$ and $R_E \cap S_U$. Together, these represent R_E sites from $S_U \cup S_T$ or MoIs targeted by StaDynA. We ran each analysis three times and reported the largest number of targets dynamically detected by each approach and the corresponding number of Monkey-generated events.

Overhead. We also calculate analysis overhead to help further explain the efficiency of each technique. We calculate overhead by comparing the time required by the techniques to perform their dynamic and static analyses (the same time used to calculate efficiency) to the time required to execute an app by itself using our input generator. The percent overhead of these two reported times is calculated using the equation $(\frac{time_{Analysis}}{time_{Monkey}} - 1) \times 100$, where $time_{Analysis}$ refers to the time taken by ReHAna or StaDynA to complete its analysis.

Slowdown Factor. To assess the scalability of these two approaches (RQ3), we quantify the effect of hybrid analysis on the performance of each system in terms of the slowdown factor. We calculate the slowdown factor using the equation $\frac{time_{Analysis}}{time_{Monkey}}$. We then used the slowdown factor to report scalability.

4.3 Study Operation

To address RQ1, we compare the memory usage and efficiency of Soot and ReHAna. We report the number of methods analyzed by each technique and the time required to analyze each app. To address RQ2 to RQ4, we performed a head-to-head comparison between ReHAna and StaDynA by observing the numbers of RDCL call sites or (MoIs) and targets identified by each approach (statically as *MoIs* and dynamically as *targets*) and measuring the time required for each approach to complete a similar analysis to generate the intended analysis result that also includes dynamically loaded code. For RQ3, we report the overhead of each approach and the speed-up of ReHAna over StaDynA. For RQ4, we observe the analysis time in terms of the slowdown factor, as code size increases.

We conducted our experiments using several Nexus 7 tablets running the modified Android API 19 (Kitkat). Both ReHAna and the server component of StaDynA ran on a laptop with Intel Core i7 and 16 GB of RAM running Apple OS X (Mojave). The communication between the tablets and the laptop was via USB-2. ReHAna uses JDWP for communication, while StaDynA uses logcat to transport logging information.

4.4 Threats to Validity

Where external validity is concerned, we compared the performances of ReHAna, Soot, and StaDynA using only a subset of apps from Google Play store, together with a few others provided by the authors of StaDynA; however, the former subset of apps represent an important contingent of the most recent, commonly used apps that are available for Android devices. They also vary in size and complexity in terms of code size and reflection usage.

Where internal validity is concerned, errors in the tools we rely on could affect our results, but we have attempted to test them rigorously. As a first step, we created several micro-benchmark apps to test both implementations to ensure that they reported analysis correctly.

Where construct validity is concerned, we measured the times that ReHAna, Soot, and StaDynA need to perform analysis. We also compared each analysis time to the time required by each app to execute without any analysis as a method for calculating the overhead of each technique. We also compared their analysis results (both static and dynamic analyses) to evaluate their effectiveness. However, we do not collect any measures related to the actual engineer effort required to perform analyses in a company setting.

5 Results and Analysis

This section reports the results of our empirical investigations to answer the prior research questions.

RQ1: What are the performance differences between Soot and ReHAna?

We report the overall performance in Table 2. We report the number of analyzed methods in columns II and V for Soot and ReHAna, respectively. Note that ReHAna 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 ReHAna. We report the analysis time of both systems in columns IV and VII.

Table 2. Comparing memory usage and analysis time between Soot and ReHAna.

App name [I]	Soot			ReHAna		
	Analyzed methods [II]	Utilized memory (MB) [III]	Analysis time (Seconds) [IV]	Analyzed methods [V]	Utilized memory (MB) [VI]	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

Soot completed analysis for all but one Android app, *Adobe Lightroom*, presumably due to incompatibility from newer programming practices. In this situation, FlowDroid terminated with a runtime exception during the callgraph-construction phase. In all 18 apps, ReHAna analyzes 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, ReHAna analyzes 11.84 times more methods than Soot.

While ReHAna analyzes more methods than Soot in every application, it also requires less memory than Soot to complete the analysis for each application. In the case of *Snake*, our smallest app, Soot needs over 6 times more memory to analyze 8 times fewer methods. In the case of *Cute Animals*, Soot needs 1.67 times more memory to analyze 19 times fewer methods. On average, Soot needs 3.45 times more memory to analyze an app.

Next, we report memory efficiency (ME) in Table 3. 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 ReHAna 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 ReHAna is 79.98 methods per one MB. The average efficiency gain of ReHAna over Soot is 36.35 times.

Table 3. Comparing memory efficiency between Soot and ReHAna.

App	ME_{Soot}	ME_{ReHAna}	Gain $\left(\frac{ME_{\text{ReHAna}}}{ME_{\text{Soot}}}\right)$
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

In term of analysis time, Table 2 shows that for the small size apps (i.e., *Snake* to *Guitar Flash*), the analysis times of Soot and ReHAna 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, ReHAna ends up analyzing more than 19 times and 9 times more methods in these apps, respectively. ReHAna 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 3.81 to 4.18 times higher than those of ReHAna.

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 ReHAna'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

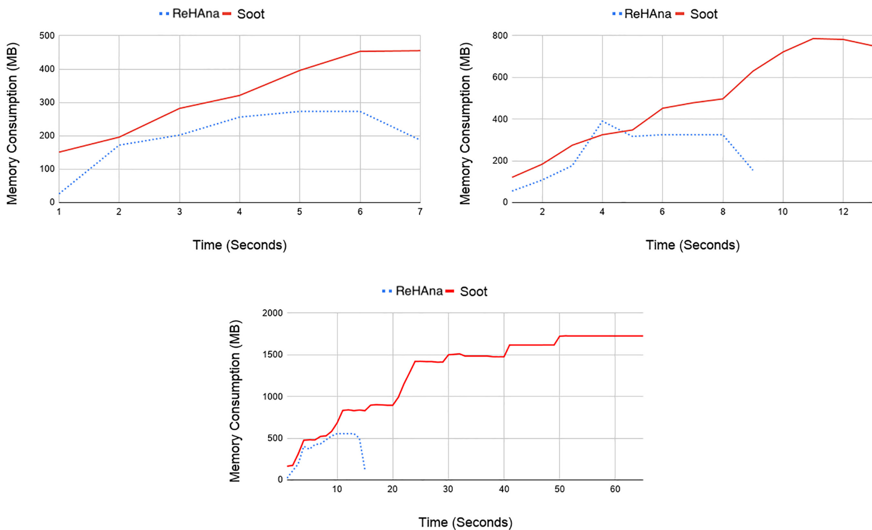

Fig. 4. Memory usage over time for analyzing a small app–cute animal sounds (top left), medium-sized app–Dolphin EMU (top right), and large app–BBC News (bottom).

Table 4. RDCL callsites statically identified by ReHAna and StaDynA and RDCL targets dynamically detected and captured by both systems. We used the eight apps provided by the authors of StaDynA.

App (I)	Uncovered RDCL callsites (MoIs)		ReHAna ∩ StaDynA (IV)	ReHAna \ StaDynA (V)	StaDynA \ ReHAna (VI)	Captured RDCL targets		ReHAna ∩ StaDynA (IX)	ReHAna \ StaDynA (X)	StaDynA \ReHAna (XI)
	ReHAna (II)	StaDynA (III)				ReHAna (VII)	StaDynA (VII)			
Avast	24	5	4	21	1	21	4	4	17	0
Symantec	19	1	1	18	0	18	1	1	18	0
ViberVOIP	26	8	4	22	4	21	4	4	17	0
FlappyBird	19	3	1	18	2	18	1	1	17	0
ImageView	36	4	4	33	0	30	4	3	27	1
Anserverbot1	21	2	2	19	0	20	2	2	18	0
Basebridge4	54	2	2	52	0	40	2	2	38	0
Fakenotify	31	10	10	21	0	16	1	1	15	0

referencing the relative logical complexity, reflected by the size of Dex files. 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 ReHAna perform in analyzing real-world apps of varying complexities. For a small app shown in Fig. 4(top left), both systems achieve the same time of completion; however, the amount of memory needed by ReHAna reaches plateau much quicker. It also requires less peak memory. For a medium-sized app (Dolphin EMU) shown in Fig. 4(top right), ReHAna 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 Fig. 4(bottom) shows, ReHAna outperforms Soot by a big margin, both in terms of speed and memory usage.

RQ2: Is ReHAna more effective than StaDynA?

We consider the effectiveness of each approach to perform hybrid analysis. As the first step, we evaluate the effectiveness of the static analysis portions of both systems. Table 4, columns II to VI report the number of uncovered RDCL call sites or MoIs within each program. For example, ReHAna uncovered 24 call sites while StaDynA only uncovered 5 call sites. We also report call sites uncovered by both approaches (column IV), ReHAna only (column V), and StaDynA only (column VI).

To identify why ReHAna uncovers more MoIs than Androguard (column V), we analyzed the log files generated by both systems. We found that the differences have to do with how ReHAna and Androguard operate. Androguard only analyzes application code. ReHAna, on the other hand, can also uncover additional call sites from within the framework and library code.

Table 5. Time required for ReHAna and StaDynA to analyze the eight apps provided by the authors of StaDynA.

App (I)	Monkey (sec.) (II)	Analysis time (sec.)		Overhead (%)		Speed-up (%) (VII)
		ReHAna (III)	StaDynA (IV)	ReHAna (V)	StaDynA (VI)	
Avast	321.55	368.17	744.30	14.50	131.47	102.16
Symantec	280.14	325.19	618.79	16.08	131.59	99.51
ViberVOIP	267.23	283.24	955.42	5.99	257.53	237.32
FlappyBird	295.09	311.28	332.01	5.49	12.51	6.66
ImageView	324.28	368.90	373.24	13.76	15.15	1.23
Anserverbot1	290.91	317.91	534.84	9.28	83.85	68.23
Basebridge4	288.52	320.23	454.58	11.02	57.55	41.91
Fakenotify	303.12	308.31	323.19	1.69	6.60	4.83

We also investigated why Androguard uncovers MoIs not uncovered by ReHAna (column VI). We found that the differences also have to do with how the two systems operate. For example, in Avast (Column VI), StaDynA identifies one additional MoI not detected by ReHAna. ReHAna analyzes only classes that can be loaded. For the analysis portion of ReHAna, any class that cannot be referenced is also not loaded (i.e., dead code in this case), and the static analysis engine does not analyze it. On the other hand, Androguard , as a compiler-based approach, analyzes all the code in an APK, including classes that would not have been loaded. As such, RDCL call sites in these classes are identified by Androguard but not by ReHAna.

Ultimately, the goal of both systems is to identify classes that have been loaded through RDCL so that both systems can include them when computing their initial static analysis results. We compare the numbers of dynamically loaded classes that both techniques can detect and capture (Table 4). As shown in Column IX, ReHAna can detect and capture the same target classes as StaDynA in seven out of eight apps. The only exception is *ImageView*. Also note that for all eight apps, Monkey succeeded in generating the requested 10,000 events for both systems, and it was able to exercise most of the MoIs so that the number of dynamically detected classes are very close to the uncovered MoIs. The two major exceptions are *Basebridge4* and *Fakenotify*, where Monkey failed to exercise at least 75% of MoIs.

To better understand why ReHAna is not able to capture a class in *ImageView*, we investigated reports generated by both systems. We discovered that even when we used the same seed to generate Monkey events, we could still experience execution non-determinism. This factor is quite profound in *ImageView*. We found that it received an RSS feed that Monkey would attempt to touch. When we ran StaDynA, Monkey was successful in instigating events on an RSS feed, so it called a method that belongs to a class that handles multi-touch gestures.[5] This scenario occurred only once in five execution attempts.

[5] The specific class name is `Ldk/nindroid/rss/ClickHandler$MultitouchHandler`.

When we ran ReHAna, however, the same method was not called. We conducted multiple runs to see if we could invoke the method, but none succeeded.

In summary, ReHAna was able to detect the same RDCL classes detected by StaDynA in all but one app, which was due to non-determinism. As such, we conclude that ReHAna is as effective at detecting RDCL targets as StaDynA.

RQ3: Is ReHAna more efficient than StaDynA?

Table 5 reports results related to RQ3. To evaluate overhead, we also needed to measure the time required to execute an app with 10,000 Monkey-generated events, but without any analysis overhead (Column II). We then measured the times for ReHAna and StaDynA to perform analysis based on the same 10,000 Monkey-generated events. We report the results in Table 5, Columns III, and IV. Columns V and VI in the table report the overhead of ReHAna and StaDynA, respectively. The overhead is the additional time needed by each system to perform an analysis of an app in terms of percentage. Note that the overheads of ReHAna range from 1.69% to 16.08%, whereas the overheads of StaDynA range from 6.60% to 257.53%.

Finally, Column VII reports the speed-ups of ReHAna over StaDynA, in the form of the ratio of StaDynA's analysis time (Column IV) to ReHAna's analysis time (Column III). As the data shows, the amount of time required by ReHAna to perform RDCL analysis for each app is consistently much less than the time required by StaDynA. The speed-up of ReHAna over StaDynA ranged from 1.23 % (for *ImageView*) to 237.32 % (for *ViberVOIP*, which is the largest app in terms of code size among the eight apps). As such, we conclude that ReHAna is more efficient than StaDynA.

Table 6. RDCL callsites statically identified by ReHAna and StaDynA and RDCL targets dynamically detected and captured by both systems. We used the 13 apps downloaded from Play store.

App (I)	Uncovered RDCL callsites (MoIs)		ReHAna ∩ StaDynA (IV)	ReHAna \ StaDynA (V)	StaDynA \ ReHAna (VI)	Captured RDCL targets		ReHAna ∩ StaDynA (IX)	ReHAna \ StaDynA (X)	StaDynA \ ReHAna (XI)
	ReHAna (II)	StaDynA (III)				ReHAna (VII)	StaDynA (VII)			
duoLingo	172	160	160	12	0	53	14	13	39	1
slither	101	99	99	2	0	23	1	1	22	0
HuluPlus	146	143	142	4	1	34	16	6	28	10
Mercari	97	93	93	4	0	38	9	9	29	0
ooVoo	87	86	82	5	4	21	5	5	16	0
Pinterest	98	96	96	2	0	8	6	2	6	4
GeometryJumpLite	105	102	102	3	0	27	5	5	22	0
SgiggleProduction	119	116	116	3	0	24	5	4	20	1
SoundCloud	100	99	97	3	2	10	5	4	6	1
LedFlashLight	146	140	140	6	0	27	14	8	19	6
RollingSky	152	143	143	9	0	13	7	0	13	7
Waze	106	103	103	3	0	22	3	3	19	0
Zedge	129	124	122	7	2	55	7	6	49	1

RQ4: Is ReHAna more scalable than StaDynA?

For RQ4, we focus on the abilities of ReHAna and StaDynA to analyze the 13 Play Store apps. As the first step, we collected the result of our static analysis investigation, as reported in Table 6, columns II to VI. The data in the table shows that ReHAna was able to identify the same MoIs in nine out of 13 apps. In the other four apps, the static analysis engine in StaDynA was able to find between one and four MoIs not detectable by ReHAna. We also noticed that these large apps contain more MoIs than the eight used to answer RQ2 and RQ3.

Table 6, columns VII to XI compare the numbers of target classes that have been detected by both approaches. ReHAna can detect the same target classes as StaDynA in 7 out of 13 apps. We also found several instances in which non-determinism is a factor that allows StaDynA to detect one more class than ReHAna (as seen in *duoLingo*, *SgiggleProduction*, *SoundCloud*, and *Zedge*.) In apps with highly complex GUIs (*Hulu Plus* and *Pinterest*), we also found that information used to construct the GUIs is served dynamically. Thus, across different runs, the GUIs can differ substantially. This discrepancy can result in significant differences in the way these apps are exercised. For *LedFlashlight*, the difference is due to advertisements. For *RollingSky*, the game logic also deviates execution from run to run.

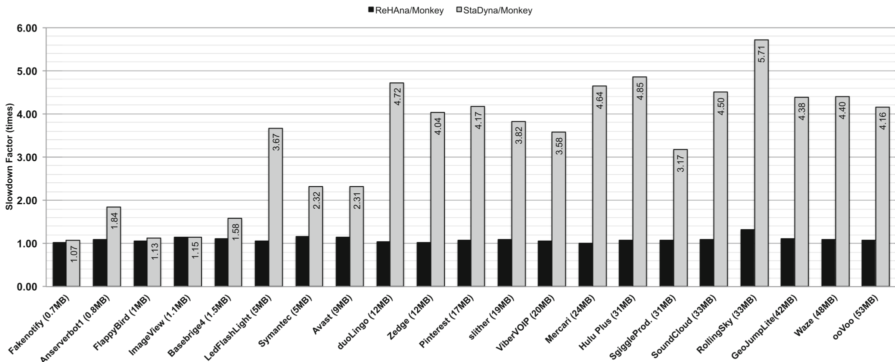

Fig. 5. Comparing the scalability of ReHAna and StaDynA using all 21 apps (sorted from smallest to largest APK sizes); each exercised by 10,000 Monkey-generated events.

Figure 5 shows the scalability of each approach based on its ability to analyze all 21 apps. The x-axis is sorted based on the app's code size, from smallest to largest (we also report the code size after each app's name). As the figure shows, the slowdown factors of ReHAna remain below 1.32 (1.0 indicates no overhead). The smallest slowdown factor is 1.01 (or 1%) for *Mercari*. The largest slowdown factor is 1.32 (or 32%) for *RollingSky*. In contrast, StaDynA incurs slowdown factors ranging from 1.07 (for *Fakenotify*, the smallest app) to 5.71 (for *RollingSky*), which is the same app that incurs the highest overhead with

ReHAna. For the largest app, *ooVoo* (53 MB), StaDynA incurs a 4.16 slowdown while ReHAna incurs a 1.07 slowdown.

Based on these results, we conclude that ReHAna is more scalable then StaDynA. In addition, ReHAna was able to analyze *colorSwitch* while StaDynA could not.

6 Discussion

Concerning RQ1, we further confirm that the primary reason for the difference in the numbers of methods analyzed by Soot and ReHAnais due to the additional ADF methods. We explain this difference through an example. Figure 6(left) illustrates a snippet of the MCG generated by Soot for the Snake app. As shown in this figure, Soot does not analyze any ADF method and therefore, such API calls (i.e., `requestFocus, setFocusable, and setFocusableInTouchMode`) would be directed to `runTimeException.init` since those API methods are not in the project. On the other hand, when ReHAna analyzes the same app, it also loads the class in the ADF to which these APIs belong. As shown in Fig. 6(right), it uncovers and analyzes more than 30 additional methods in these APIs.

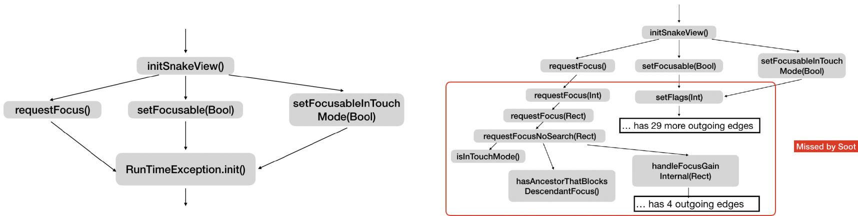

Fig. 6. Portions of method call graphs generated by Soot *(left) and* ReHAna *(right)*

For RQ2 to RQ4, ReHAna also analyzes more methods than Androguard, the static analysis framework used in StaDynA. After performing a manual inspection of the generated results, we discovered that there are many RDCL call sites in these ADF classes. These RDCL call sites were not detectable by Androguard, because Androguard does not analyze these classes.

For Avast (Column VI), StaDynA identifies one additional MoI not detected by ReHAna. Our further investigation shows that this difference stems from differences in the ways the two underlying static analysis frameworks operate. ReHAna analyzes only classes that can be loaded. For the analysis portion of ReHAna, any class that cannot be referenced is also not loaded (i.e., dead code in this case), and the static analysis engine does not analyze it. On the other hand, Androguard , as a compiler-based approach, analyzes all the code in an APK including classes that would not have been loaded. As such, RDCL call sites in these classes are identified by Androguard but not by ReHAna.

In terms of performance, two major time components can dominate the cost of dynamic analysis: (1) the actual execution time of an app, and (2) the analysis time required by each approach. The analysis time includes the time required to perform initial static analysis to build the initial MCG for StaDynA or to build analysis context graphs such as CFGs, DFGs, and initial ICFGs for ReHAna. This time component is highly dependent on the amount of code that must be statically analyzed. For example, *Mercari* has nearly 60,000 methods (Table 1); therefore, StaDynA incurs an analysis time of 737 s. It turns out that ReHAna only needed about two seconds of wall-clock time to perform the same analysis. This is possible because ReHAna can hide a significant portion of the static analysis cost within its dynamic analysis phase. StaDynA, on the other hand, cannot perform static analysis until the dynamic analysis is complete; this is because it needs to include dynamically loaded code components as part of its analysis context.

7 Related Work

Several research efforts attempt to improve the soundness of static analysis in the presence of dynamically loaded code through Java reflection. Livshits et al. [19] propose a static analysis algorithm that can approximate reflection targets using points-to information. Felt et al. [11] discuss the challenges of handling reflection in Android applications and then attempt to address them using Stowaway, a static analysis tool that is capable of identifying reflective calls and tracking reflection targets by performing flow-sensitive analysis. More recent static analysis approaches aim to improve precision. These approaches include DroidRA [16], Sparta [2], and SOLAR [17]. DroidRA adapts TamiFlex to analyze Android apps for dynamically loaded code statically. Unlike TamiFlex, DroidRA does not execute apps; instead, it uses a constraint solver to resolve reflection targets. It also uses its version of *Booster* to manipulate Jimple, an immediate representation used by Soot directly. TamiFlex, on the other hand, manipulates Java bytecode. Sparta implements annotations in the Checker framework to track information flow and a type inference system to trace reflective calls. Sparta also operates at the source code level and not the bytecode or Dexcode level. However, these static analysis approaches can only work in cases where we can identify reflection targets from the source code. For the most up-to-date and comprehensive review of static analysis approaches for handling reflection, see Landman et al. [15].

Concerning dynamic analysis, Davis et al. [8] provide an app rewriting framework named RetroSkeleton that is capable of intercepting reflections at runtime; however, this approach does not work with custom classloaders. Sawin et al. [25] propose an approach that combines static string analysis with dynamic information to resolve dynamic class loading via Java reflection. This approach operates only on the standard Java library. Execute This! [21] is a dynamic analysis approach that relies on an Android VM modification to detect reflection calls. It logs runtime events and performs static analysis off-line. Our approach, on the other hand, performs the continuous analysis.

8 Conclusion and Future Work

The ability to perform program analysis "at speed" is critical for enabling software engineers to quickly remove defects at production time and enable security analysts to evaluate deployed apps for vulnerabilities quickly. Currently, existing program analysis approaches, while useful, are too inefficient to provide such analysis capabilities. One primary reason for this inefficiency is that these approaches perform analyses in a closed-world fashion.

In this paper, we introduce ReHAna, our proposed open-world analysis framework for Android. At the heart of ReHAna is our proposed Virtual Class-Loader (VCL) that is capable of incrementally load and analyze classes. This capability is suitable for performing hybrid program analysis. We implemented an instance of ReHAna to analyze apps for RDCL components and then used it to analyze a large corpus of apps. Our evaluation indicates that ReHAna is as effective as StaDynA, an existing closed-world RDCL analysis approach, while significantly reducing analysis time.

In future work, we plan to create other instances of ReHAna to support real-time code coverage visualizations and real-time policy enforcement. We plan to use the class-loading capability of ReHAna to perform program decomposition that can make event sequence generation and verification of Android apps more tractable. We will make the source code of ReHAna publicly available upon the acceptance of this paper.

Acknowledgments. We would like to thank Yury Zhauniarovich, Maqsood Ahmad, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci for sharing the source code of StaDynA and the applications used to evaluate StaDynA with us.

References

1. Abraham, J., Jones, P., Jetley, R.: 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/
2. Barros, P., et al.: Static analysis of implicit control flow: resolving Java reflection and android intents (t). In: Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), ASE 2015, pp. 669–679, Lincoln, NE, USA, November 2015
3. Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., Mezini, M.: Taming reflection: aiding static analysis in the presence of reflection and custom class loaders. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 241–250, Honolulu, Hawaii, USA, May 2011
4. Bond, M.D., Coons, K.E., McKinley, K.S.: PACER: proportional detection of data races. In: Proceedings of the Conference on Programming Language Design and Implementation, pp. 255–268, Toronto, Ontario, Canada, June 2010
5. Chandra, B.: A technical view of the open SSL heartbleed vulnerability, May 2014. https://www.ibm.com/developerworks/community/files/form/anonymous/api/library/38218957-7195-4fe9-812a-10b7869e4a87/document/ab12b05b-9f07-4146-8514-18e22bd5408c/media

6. Chen, Y., et al.: Mass discovery of android traffic imprints through instantiated partial execution. In: Proceedings of CCS, pp. 815–828, Dallas, Texas, USA (2017)
7. Choudhary, S.R., Gorla, A., Orso, A.: Automated test input generation for android: are we there yet? In: Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), ASE 2015, pp. 429–440 (2015)
8. Davis, B., Chen, H.: Retroskeleton: retrofitting android apps. In: Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2013, pp. 181–192, New York, NY, USA. ACM (2013)
9. Desnos, A.: Androguard: reverse engineering, malware and goodware analysis of android applications (2013). https://github.com/androguard/androguard
10. Duan, Y., et al.: 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
11. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, pp. 627–638, New York, NY, USA. ACM (2011)
12. GeeksforGeeks. ClassLoader in Java, May 201r. https://www.geeksforgeeks.org/classloader-in-java/
13. Google. Lint (2019). http://tools.android.com/tips/lint
14. Jim, T.: Legacy C/C++ code is a nuclear waste nightmare that will make you WannaCry, June 2017. http://trevorjim.com
15. Landman, D., Serebrenik, A., Vinju, J.: 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
16. Li, L., Bissyandé, T.F., Octeau, D., Klein, J.: Droidra: taming reflection to support whole-program analysis of android apps. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, pp. 318–329, Saarbrücken, Germany (2016)
17. Li, Y., Tan, T., Xue, J.: Understanding and analyzing java reflection. ACM Trans. Softw. Eng. Methodol. **28**(2), 1–50 (2019)
18. Liang, S., Might, M., Horn, D.V.: Android: malware analysis of android with user-supplied predicates. CoRR, abs/1311.4198 (2013)
19. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: Proceedings of the 14th Conference on USENIX Security Symposium, SSYM 2005, vol. 14 (2005)
20. Oracle Corp. Loading, linking, and initializing, November 2019. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html
21. Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., Vigna, G.: Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In: Proceedings of NDSS, vol. 14, pp. 23–26, San Diego, CA (2014)
22. Ponomariov, P.: Shedun: adware/malware family threatening your Android device, September 2015. https://blog.avira.com/shedun/
23. Rasthofer, S., Arzt, S., Miltenberger, M., Bodden, E.: Harvesting runtime values in android applications that feature anti-analysis techniques. In: Proceedings of NDSS (2016)
24. Rus, S., Rauchwerger, L., Hoeflinger, J.: Hybrid analysis: static & dynamic memory reference analysis. Int. J. Parallel Program. **31**(4), 251–283 (2003)
25. Sawin, J., Rountev, A.: Improving static resolution of dynamic class loading in java using dynamically gathered environment information. Autom. Softw. Eng. **16**(2), 357–381 (2009)

26. Smith, J., Nair, R.: Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design). Morgan Kaufmann Publishers Inc., San Francisco (2005)
27. Späth, J., Lam, P.: Using Soot and TamiFlex to analyze DaCapo, August 2014. https://github.com/Sable/soot/wiki/Using-Soot-and-TamiFlex-to-analyze-DaCapo
28. Tikir, M., Hollingsworth, J.K.: Efficient instrumentation for code coverage testing. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2002, pp. 86–96, Roma, Italy (2002)
29. Vallée-Rai, R.: Soot: a java bytecode optimization framework. Master's thesis, McGill University (2000)
30. Wu, D., Liu, X., Xu, J., Lo, D., Gao, D.: Measuring the declared SDK versions and their consistency with API calls in android apps. In: Ma, L., Khreishah, A., Zhang, Y., Yan, M. (eds.) Wireless Algorithms. Systems, and Applications, pp. 678–690. Springer, Cham (2017)
31. Xu, L.: Techniques and tools for analyzing and understanding android applications. PhD thesis, University of California, Davis (2013)
32. Zhauniarovich, Y., Ahmad, M., Gadyatskaya, O., Crispo, B., Massacci, F.: 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, CODASPY 2015, pp. 37–48, San Antonio, Texas, USA (2015)