University of Nebraska - Lincoln

DigitalCommons@University of Nebraska - Lincoln

CSE Journal Articles

Computer Science and Engineering, Department of

2019

JITANA: A modern hybrid program analysis framework for android platforms

Yutaka Tsutano

Shakthi Bachala

Witawas Srisa-an

Gregg Rothermel

Jackson Dinh

Follow this and additional works at: https://digitalcommons.unl.edu/csearticles

Part of the Computer and Systems Architecture Commons, and the Systems and Communications Commons

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in CSE Journal Articles by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

University of Nebraska Lincoln
digitalcommons.unl.edu

JITANA: A modern hybrid program analysis framework for Android platforms

Yutaka Tsutano, Shakthi Bachala, Witawas Srisa-an, Gregg Rothermel, Jackson Dinh

Department of Computer Science and Engineering,
University of Nebraska-Lincoln, Lincoln, NE, 68588, USA

Corresponding author — Witawas Srisa-an, *email* witty@cse.unl.edu

Email addresses — ytsutano@cse.unl.edu (Y. Tsutano), sbachala@cse.unl.edu (S. Bachala), grother@cse.unl.edu (G. Rothermel), jdinh@cse.unl.edu (J. Dinh).

Abstract
Security vetting of Android apps is often performed under tight time constraints (e.g., a few minutes). As such, vetting activities must be performed "at speed", when an app is submitted for distribution or a device is analyzed for malware. Existing static and dynamic program analysis approaches are not feasible for use in security analysis tools because they require a much longer time to operate than security analysts can afford. There are two factors that limit the performance and efficiency of current analysis approaches. First, existing approaches analyze only one app at a time. Finding security vulnerabilities in collaborative environments such as Android, however, requires collaborating apps to be analyzed simultaneously. Thus, existing approaches are not adequate when applied in this context. Second, existing static program analysis approaches tend to operate in a "closed world" fashion; therefore, they are not easily integrated with dynamic analysis processes to efficiently produce hybrid analysis results within a given time constraint.

In this work, we introduce JITANA, an efficient and scalable hybrid program analysis framework for Android. JITANA has been designed from the ground up to be

Published in *Journal of Computer Languages* 52 (2019), pp 55–71.
doi 10.1016/j.cola.2018.12.004
Submitted 29 October 2018; revised 25 November 2018; accepted 16 December 2018; published 24 April 2019.
Copyright © 2019 Elsevier Ltd. Used by permission.

used as a building block to construct efficient and scalable program analysis techniques. JITANA also operates in an open world fashion, so malicious code detected as part of dynamic analysis can be quickly analyzed and the analysis results can be seamlessly integrated with the original static analysis results. To illustrate JITANA's capability, we used it to analyze a large collection of apps simultaneously to identify potential collaborations among apps. We have also constructed several analysis techniques on top of JITANA and we use these to perform security vetting under four realistic scenarios. The results indicate that JITANA is scalable and robust; it can effectively and efficiently analyze complex apps including Facebook, Pokémon Go, and Pandora that the state-of-the-art approach cannot handle. In addition, we constructed a visualization engine as a plugin for JITANA to provide real-time feedback on code coverage to help analysts assess their vetting efforts. Such feedback can lead analysts to hard to reach code segments that may need further analysis. Finally we illustrate the effectiveness of JITANA in detecting and analyzing dynamically loaded code.

Keywords: Android security, Static analysis, Dynamic analysis

1. Introduction

Currently, over 90% of malicious software or malware developed for smart-mobile devices targets Android systems. This is due to the open-source nature of the Android OS and its popularity (there were at least 1.4 billion Android devices deployed as of the beginning of 2017). To deal with widespread and rapid releases of malware, security analysts use program analysis approaches to identify potential malware, and then analyze potentially malicious apps to assess their behaviors [7,35,39]. While these analysis approaches and tools have been effective for identifying faults during a typical software development life cycle and even after software deployment, security analysis often requires more stringent criteria in terms of scalability, performance, and analysis time. For example, an Android app can utilize features from other apps to perform services. As such, malware can utilize features from other apps to perform malicious acts such as leaking sensitive information. For an analyst to have a clear picture of what an app is capable of doing, the analyst needs to scalably analyze other apps and services in conjunction with the target app.

While existing approaches can perform such analyses, they are highly inefficient [15,21,22]. As such, they are cannot be used to address today's security needs. As a simple use case, many organizations have adopted Bring-Your-Own-Device policies, allowing their own employees and visitors to connect their personal devices to the company's networks. Prior to connecting these devices to the networks, they should be vetted for malware. Vetting in this case should be done "at speed" in a matter of minutes. Unfortunately, using existing state-ofthe- art approaches and tools to analyze apps in a device for connections can take many hours, making them infeasible for at-speed security vetting.

Because existing analysis techniques can take a long time to complete, a common vetting technique is to run a virus scanner on apps, or exercise them for a few minutes to see if they perform malicious deeds. As an example, Google's earlier attempt to vet apps was BOUNCER. According to multiple sources, BOUNCER spent only 5 min vetting a given app. Therefore, we have seen reports of methods for bypassing BOUNCER's vetting effort [19,31]. Approaches adopted by malware authors include delaying malicious actions for more than 5 min and avoiding malicious actions if the app is running on an emulator, which is the platform that BOUNCER employs.

Prior work has shown that combining static and dynamic analysis approaches can yield effective analysis solutions [4,25,39]. As such, Google's recent Cloud-based security analysis performs both static and dynamic analysis as part of the application review process [7]. This process consists of an automated risk analyzer that performs a two-step analysis process. In the first step, it statically analyzes the code by extracting *application features* to detect potentially bad behaviors (e.g., sending out senstive information via SMS and application code to perform cloaking strategies to evade detection during dynamic analysis). It then performs dynamic analysis to identify "interactive behaviors" that are not detectable by static analysis. If an app is flagged as potentially harmful, it is referred to a security analyst for manual review [7].

As part of Google's dynamic analysis, a qualitative metric could involve measuring code coverage to see if most paths have been exercised. However, measuring code coverage can add a significant amount of time to the risk assessment process. Typically, to obtain code coverage information, static analysis is first performed on the app under investigation to identify paths. A vetting system such as that used by Google could combine this step with their static analysis effort. However, the next step requires that the app being analyzed be instrumented [29], which can be intrusive. (The term "instrumentation" refers to a mechanism used to obtain dynamic path information, such as via Dex code instrumentation or using a modify VM.) The analyst could then run the instrumented version of the app to log paths that have been exercised during the run. Afterward, the path information is compared against static analysis information to determine coverage. As described, this process requires at least three distinct steps that may require a substantial amount of time.

In addition, modern Android apps may not disclose all of their code in the Android Package Kit (APK). There are several mechanisms by which an Android app can add additional code at run time. For example, one version of Facebook can add three additional Dex files in addition to the main Dex file at run time. When static analysis is applied to this particular app, it analyzes only the main Dex file. A similar mechanism is also used by *Shedun*, a malware that at its peak infected thousands of devices per day [26]. In this case, additional Dex files containing malicious code are loaded at runtime to obfuscate malicious components and prevent them from being identified via static analysis. To analyze these apps, we must include additional Dex files in order to construct various program analysis contexts (e.g., control-flow and data-flow information). These additional files, however, are typically captured at runtime.

While a combination of static and dynamic analysis techniques could eventually capture all of the code needed to perform analysis, the process for combining the results of such analyses is not efficient. This is because most static analysis tools used in such approaches operate in a "closed world" fashion. That is, all of the information needed must be included prior to performing static analysis. If additional information is discovered at runtime, it must first be included in the analysis scope and then the static analysis must be repeated. In dynamic systems such as Android, this can lead to multiple attempts to analyze an app, making the process unsuitable for near real-time or at-speed security analysis.

In this paper, we introduce JITANA, a new hybrid analysis framework designed to overcome the foregoing limitations by providing an effective, fast, scalable, and incremental Android program analysis framework enabling security analysts to construct complex program analysis techniques to detect real-world security vulnerabilities. One notable capability of JITANA is that instead of analyzing one app at a time, JITANA has been designed to analyze a set of apps simultaneously and quickly. This allows JITANA to quickly analyze all applications in a typical Android tablet (about 100 installed apps). Another notable capability is that instead of utilizing a complex workflow to perform hybrid analysis, JITANA's has been designed to allow hybrid analysis to be performed efficiently. We can even visualize hybrid analysis results (code coverage is used as a use case in this paper) in real-time. Furthermore, instead of using a "closed world" analysis, JITANA performs analysis in an "openworld" fashion. This allows static and dynamic analyses to be tightly integrated into one continuous process instead of being applied in the traditional multi-step, loosely integrated process.

We examine four scenarios in which JITANA can be used by security analysts to construct complex program analysis techniques to provide information that can help with security analysis. First, we show that JITANA can be used as a building block to statically analyze collections of real-world commercial apps for IACs; this shows that it can be realistically used for security analysis. Second, we show that JITANA is scalable and can be used to quickly analyze apps installed in a device for IACs. Third, we illustrate the ease with which JITANA can be enhanced by creating a visualization plugin called TRAVIS, that provides real-time code coverage information that can help analysts assess the quality of their vetting attempts. Fourth, we use JITANA to dynamically capture three additional dynamically loaded Dex files in *Facebook* and augment the initial static analysis result with information from these dynamically loaded classes.

The rest of this paper is organized as follows. Section 2 provides illustrations of existing obstacles that prevent at-speed analysis from being performed. These obstacles motivated us to develop JITANA. Section 3 describes the design and implementation of JITANA including its static and dynamic analysis components. Section 4 discusses scenarios in which JITANA can be used to perform at speed security analyses. Section 5 provides additional insights into the reported results. Section 6 highlights prior research efforts related to our work. Section 7 concludes.

2. Motivation

In this section, we provide three examples that highlight limitations of today's static and dynamic analysis approaches that prevent them from being utilized in real-time or near real-time security analysis. These three examples expose the challenges for: (i) Performing largescale IAC analysis, (ii) performing real-time code coverage analysis to determine the quality of a dynamic analysis run, and (iii) continuously analyzing an application that adds code at runtime, rendering static analysis inadequate.

2.1. Analyzing apps for IAC connectivity

As noted in Section 1, current approaches for analyzing groups of Android apps for problems related to interactions require the apps or their representations to be aggregated; they then can be analyzed by existing tools such as EPICC [22], IC3 [21], or SOOT [36]. Li et al. [16] describe three approaches for creating aggregated results. The first approach analyzes each app for possible flows and then combines the results to yield potential inter-app flows. Approaches such as EPICC, IC3, and SIFTA [37] fall into this category. This approach is scalable because most of the analysis effort is intra-app. However, by considering only the analysis results (e.g., possible flows, variability aware data structures), the approach does not preserve various graphs such as CFGs, DFGs, ICFGs, and points-to graphs (we refer to these henceforth as *analysis graphs*) constructed as part of intra-app analyses. Such graphs are crucial, however, for conducting richer and more precise analyses such as static taint or precise dataflow analysis.

The second approach for creating aggregated results combines CFGs instead of possible flows. However, combining CFGs from multiple apps can be memory intensive. One option for addressing this problem is to create CFGs in forms that can be persisted to external storage (e.g., serializable objects) before they are combined. CFGs of apps should also be generated using the same format (e.g., using the same tool) so that they can be merged. To date, we are not aware of any system that addresses the problem of IAC analysis using this approach. Bagheri et al. [3] first generate models of components and then combine them to perform analysis; this approach has been used to detect permission leakage [3]. However, their models include only portions of the analysis information relevant to the tasks we are concerned with.

The third approach for creating aggregated results is the hybrid approach [16] we described in Section 1. Again, the approach uses APKCOMBINER to merge a pair of apps at the bytecode level, to create a single app that can be analyzed using existing ICC analysis tools such as EPICC [22] or general program analysis frameworks such as SOOT [36]. APKCOMBINER resolves two types of naming conflicts. When two apps use a common method (same name, same code), only one is maintained. When two apps use two methods with the same name (same name, different code), one is renamed.

While this approach can cause the code in the resulting app to bloat, the authors argue that in the context of security, it is not likely that colluding malware would employ more than a few apps because it is less likely for a user to have all required apps installed on a device. However, there are other dependability and security contexts that may require engineers and analysts to consider more than a pair of apps at a time [12]. For example, an analyst may need to identify software components and apps installed on a device that may be connected to a shared vulnerable component. If this vulnerable component is heavily used, the number of apps connected to it could be quite large. As such, APKCOMBINER may face problems scaling up to sets of apps, because it can merge only a pair of apps at a time. The resulting app is also not guaranteed to run, so testing it may not be possible.

We experimented with using APKCOMBINER to iteratively combine multiple apps in a pairwise fashion. For example, to combine four apps (a_0, a_1, a_2, and a_3), a_0 and a_1 are first combined. The resulting app a_{01} is then combined with a_2, and the result a_{012} is combined with a_3. We found that APKCOMBINER cannot merge a resulting app, e.g., a_{01}, with another app a_2: In such cases, it ignores a_2 without reporting any error.

In summary, creating aggregated analysis results is a necessary step for approaches that analyze multiple connected apps using existing program analysis tools that have been designed to analyze one app at a time. We have discussed existing approaches for creating these aggregated results; they suffer from high complexity, or failure to preserve generated program analysis information. To preserve contexts while controlling the number of apps that need to be merged, Li et al. [16] introduced a hybrid approach. While this approach can analyze a small number of connected apps, we show in Section 4 that it faces scalability and robustness issues when it is used to merge a large number of apps or complex apps.

2.2. Providing real-time code coverage

Beginning with Android Studio version 0.10.0, a Java code coverage tool, JACOCO, has been integrated into Gradle (an opensource build automation framework) as a plugin. Android developers can use this plugin to quantify the quality of their test inputs. To do so, a developer configures Gradle to enable test coverage and then applies the JACOCO plugin. This code coverage framework has been designed to cover all variations of an Android product. However, to use it, the developer must have the Java source code of that particular product [8].

While JACOCO has been widely used by Android developers, it cannot be used by security analysts who typically need to analyze applications without Java source code. To obtain code coverage information without the Java source files, developers can use EMMA, a code coverage framework based on Dex code instrumentation. EMMA first divides the application into basic blocks, then instruments these blocks with counting methods. When the instrumented app is executed, these counting methods produce meta-data that can be used for post-run processing to generate code coverage reports. One side effect of using EMMA is larger code size due to instrumentation and degraded execution performance [29]. Such a multi-step process for obtaining code coverage information also makes EMMA less applicable to security analyses that must be done at-speed.

According to various reports, dynamic analysis is still used as part of Google’s risk assessment process to make decisions about whether submitted apps should be admitted to its Play Store. This Cloud-based analysis uses dynamic analysis to detect malicious behaviors within an app that cannot be detected by static analysis [7]. It scans for malware and then runs each app in a device emulator to observe its behaviors. If the app does not exhibit malicious behaviors, it can be admitted to the store.

While this approach is quite effective, reports have shown that it can still let malicious apps into the store [1,19]. Further, the approach used by Google does not assess the quality of each vetting effort with a typical metric such as code coverage. While the reason for this is not disclosed, we suspect that the inefficiency of existing code coverage approaches makes integrating them into the vetting workflow more difficult. To do so, the vetting time must be increased. With more than 300,000 apps admitted to Play Store in the last quarter, increasing each vetting time can significantly prolong the overall admission process.

To improve the effectiveness of vetting processes such as that used by Google, we need to provide real-time information so that the quality of a vetting attempt can be assessed. This means that within a specific risk assessment time budget, we need to be able to statically analyze, run, and determine the code coverage of that run to decide whether the app needs to be run longer. The computation of code coverage needs to be continuous so that real-time information can be provided while incurring low execution overhead. In Section 4, we show how JITANA can be used to provide real-time code coverage information to achieve this goal.

2.3. Capturing and analyzing dynamically loaded code

The Android Dex file system traditionally has a 16-bit limitation on the number of methods that can be in the main Dex file, typically named `classes.dex`. This translates to approximately 64,000 methods. In large apps such as *Facebook*, the number of methods may exceed this limitation. As such, these large apps need to find a way to have the system recognize and load the additional Dex files. In addition, complex malware such as *Shedun* attempts to insert additional Dex files at runtime to avoid detection. In this case, `classes.dex` does not contain code that can perform malicious behaviors; instead, the additional Dex files contain that code. In this way, this particular malicious app was able to defeat virus scanners that check only `classes.dex`.

Because the Android OS and its Virtual Machines (Dalvik and ART) are opensource, the approach that *Facebook* engineers use to load additional Dex files is by modifying the system class loader to recognize such files in addition to `classes.dex`. To do this, they use Java reflection to directly modify the internal structures of the system class loader and add additional space to the Dalvik internal buffer (from 5MB to 8MB) to support all the features. While this approach was used in the first version of *Facebook* for Android, it continues to be used in all versions of *Facebook* apps that can run on the Dalvik VM.

Approaches have been introduced for dealing with dynamic code loading [4,39]. Since code is loaded at runtime, these approaches rely on dynamic analysis to capture the dynamically loaded code. Once captured, such code is added to the original code and static analysis is performed again to produce more complete analysis results. If additional code is found later, the new code is once again added to the already known code and static analysis is repeated. This is an iterative process and each iteration requires multiple steps to complete (i.e., run the app, capture dynamically loaded code, and perform reanalysis). As such, the process is too inefficient to be used for at-speed security analysis especially when the vetting time is short.

To render the foregoing process more efficient, we need to change it from iterative to continuous. Captured dynamic information must be processed by static analysis immediately without restarting the static analysis process. Most current static analysis approaches including the widely used SOOT operate in a "closed-world" fashion in which continuous analysis is not possible [4]. Section 4 illustrates how JITANA can be extended into an "open-world" analysis framework so that dynamic information can be continuously processed without interruption. This can be particularly useful for detecting malware that uses dynamically loaded malicious code. We show how JITANA can be used to detect this type of malware.

3. Introducing JITANA: A hybrid program analysis framework

To provide a new hybrid program analysis framework for Android that can efficiently and scalably analyze multiple apps simultaneously while operating in an "open-world" fashion, we introduce JITANA,[1] a scalable and efficient framework for supporting static and dynamic program analysis techniques. JITANA is intended to be an efficient, scalable, and extensible program analysis framework. Next, we describe the main rationales and design choices used to achieve these three performance objectives.

Efficiency. The large amount of information that must be generated to set up the basic program analysis data structures (e.g., control-flow and data-flow graphs) required to analyze apps can result in

1. JITANA is available for download at: http://cse.unl.edu/~ytsutano/jitana/ .

high memory consumption. Bodden et al. [4] recommend that users of SOOT, a widely used program analysis and optimization engine for Java, set heap size to at least 10GB. Even with this much heap space, we have found that SOOT can still run out of heap memory when used to analyze a large Android app. Large heaps can also result in other runtime overhead including garbage collection and excessive paging if main memory is not sufficiently large. As such, a key design criterion for JITANA is to employ memory conservation.

In practice, one way to reduce memory consumption is to reduce the amount of information that must be processed. Our solution for achieving the desired level of memory consumption is inspired by the way runtime systems supporting object-oriented languages (e.g., Java Virtual Machines or JVMs) load classes and employ Just-In-Time (JIT) compilation. By structuring a program as a collection of classes, the entire codebase of a Java program is partitioned into small and independent units of code. The classloader incrementally loads each class as needed at runtime. Once a class is loaded, each method belonging to that class that needs to be executed is analyzed and then compiled to binary code by the JIT compiler. Prior work [10,23] has shown that the analysis results incrementally generated by the JIT compiler can be used to reconstruct common analysis context in forms such as controlflow and data-flow graphs.

The key ingredient in making this approach consume less memory than bulk analysis is its incremental nature. When a method belonging to a class is analyzed, memory is needed to perform the necessary analysis. However, once the analysis is performed, the memory can be recycled back to the system, and the only residual cost is the memory needed to store the analysis results. The efficient recycling of memory allows this approach to have a significantly smaller memory footprint than bulk analyses.

The major component of JITANA, inspired by the class-loading mechanism in the JVM, is the *Class-Loader Virtual Machine* (*CLVM*). Its main purpose is to virtualize an actual class loader. In a typical JVM, determining which class to load depends on the actual runtime needs of a program. Our CLVM, on the other hand, is designed to support efficient static and dynamic analyses. As such, it determines which classes to load by statically analyzing code to determine relationships among classes. Once classes are loaded, the system generates *class graphs* that can be used to represent those class relationships.

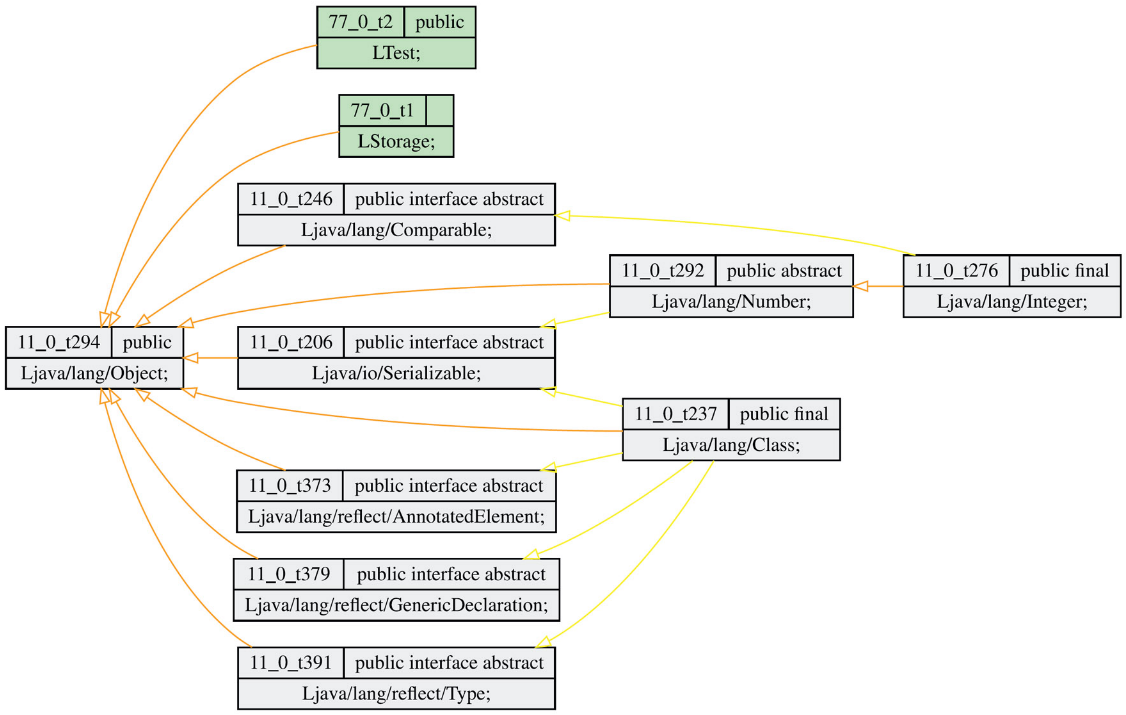
77_0_t2 | public
LTest;
77_0_t1
LStorage;
11_0_t246 | public interface abstract
Ljava/lang/Comparable;
11_0_t292 | public abstract
Ljava/lang/Number;
11_0_t276 | public final
Ljava/lang/Integer;
11_0_t294 | public
Ljava/lang/Object;
11_0_t206 | public interface abstract
Ljava/io/Serializable;
11_0_t237 | public final
Ljava/lang/Class;
11_0_t373 | public interface abstract
Ljava/lang/reflect/AnnotatedElement;
11_0_t379 | public interface abstract
Ljava/lang/reflect/GenericDeclaration;
11_0_t391 | public interface abstract
Ljava/lang/reflect/Type;

Fig. 1. A class graph for a simple *test* app.

Fig. 1 illustrates the class relationship within a simple *test* app. The app contains only two actual classes shown in green. The remaining classes (shown in gray) are from standard Java library classes, which are easily detected by JITANA. Classes in the figure were uncovered through our static analysis; i.e., they were uncovered without running the program. Our CLVM performs reachability analysis to load these classes. We discuss the actual implementation of CLVM in Section 3.1.

Our analysis approach is also inspired by the JIT compiler; i.e., it incrementally analyzes each method as the class it belongs to is loaded. However, our static analysis analyzes every method within that class. As each method is analyzed, the system produces an *instruction graph* for that method. It also computes *intra-procedural control-flow and data-flow information*. **Fig. 2** illustrates a simple instruction graph for method `GameBase.sgn(int)` in *SuperDepth*.

As more method calls are made by a target app, a method call graph for that app is also generated. For example, **Fig. 3** illustrates a small subset of the methods that are called during an analysis of two apps simultaneously, *SuperDepth* (shown as yellow method nodes)

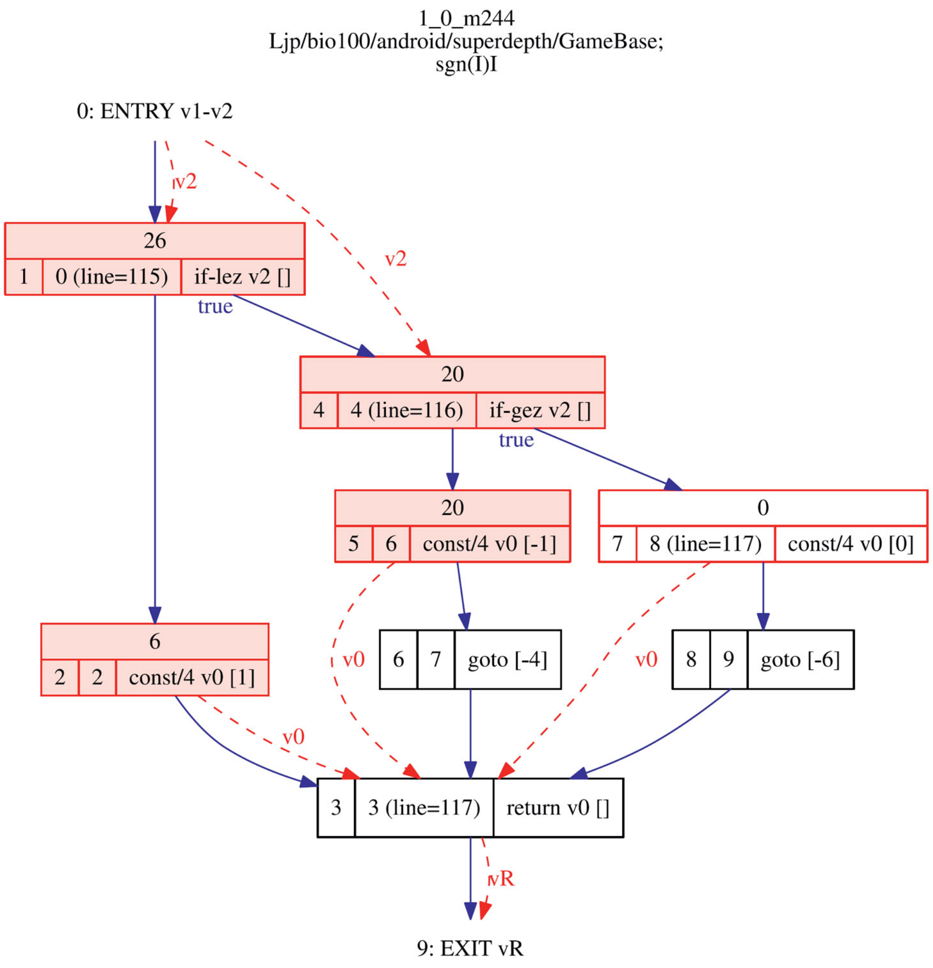
1_0_m244
Ljp/bio100/android/superdepth/GameBase;
sgn(I)I
0: ENTRY v1-v2
v2
v2
26
1 | 0 (line=115) | if-lez v2 []
true
20
4 | 4 (line=116) | if-gez v2 []
true
20
5 | 6 | const/4 v0 [-1]
0
7 | 8 (line=117) | const/4 v0 [0]
6
2 | 2 | const/4 v0 [1]
v0
6 | 7 | goto [-4]
v0
8 | 9 | goto [-6]
v0
3 | 3 (line=117) | return v0 []
vR
9: EXIT vR

Fig. 2. Instruction graph of a method in *SuperDepth*.

and *Instagram* (shown as blue method nodes). (We discuss our system's ability to analyze multiple apps simultaneously below.) The gray nodes represent method calls to Java and Android libraries. Each method call graph represents the method call and class relationships within an app as edges (e.g., direct, virtual and super). We explain these edges later in this section.

To allow reachability analysis to be performed on-line, JITANA does not remove edges that have been added to the method graph. This preserves prior reachability computations, allowing analysis to be performed on-line. For example, we can statically generate graphs of an app including a method call graph. At runtime, if a new class is dynamically loaded from an external source, our dynamic analysis can capture this class. We can then utilize the same analysis engine

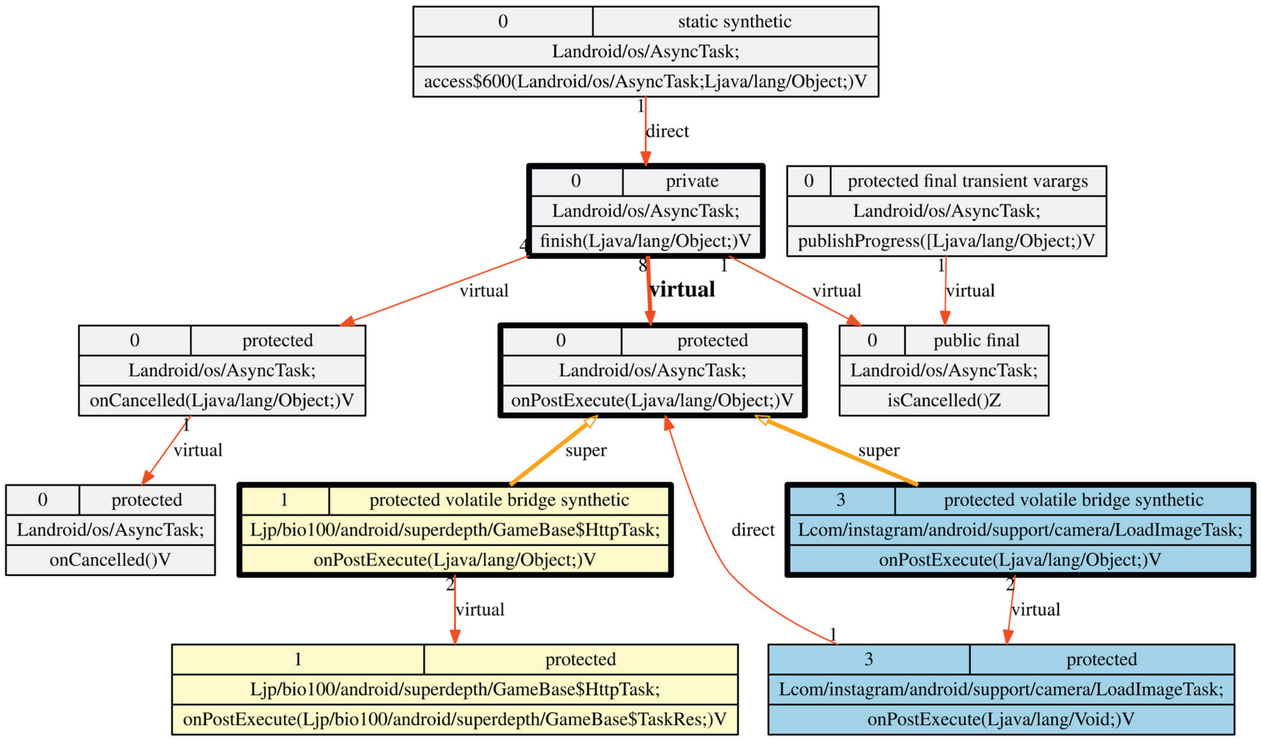
0 | static synthetic
Landroid/os/AsyncTask;
access$600(Landroid/os/AsyncTask;Ljava/lang/Object;)V
1 direct
0 | private
Landroid/os/AsyncTask;
finish(Ljava/lang/Object;)V
0 | protected final transient varargs
Landroid/os/AsyncTask;
publishProgress([Ljava/lang/Object;)V
4 virtual
8 virtual
1 virtual
1 virtual
0 | protected
Landroid/os/AsyncTask;
onCancelled(Ljava/lang/Object;)V
0 | protected
Landroid/os/AsyncTask;
onPostExecute(Ljava/lang/Object;)V
0 | public final
Landroid/os/AsyncTask;
isCancelled()Z
1 virtual
super
super
0 | protected
Landroid/os/AsyncTask;
onCancelled()V
1 | protected volatile bridge synthetic
Ljp/bio100/android/superdepth/GameBase$HttpTask;
onPostExecute(Ljava/lang/Object;)V
direct
3 | protected volatile bridge synthetic
Lcom/instagram/android/support/camera/LoadImageTask;
onPostExecute(Ljava/lang/Object;)V
2 virtual
2 virtual
1
1 | protected
Ljp/bio100/android/superdepth/GameBase$HttpTask;
onPostExecute(Ljp/bio100/android/superdepth/GameBase$TaskRes;)V
3 | protected
Lcom/instagram/android/support/camera/LoadImageTask;
onPostExecute(Ljava/lang/Void;)V

Fig. 3. A snippet of a method graph generated by analyzing two apps.

previously used to further analyze this new class and augment the class graph and method call graph with information related to newly uncovered classes and methods.

Note that we only show a few methods for comprehensibility. An actual method graph for a complex app can contain tens of thousand of nodes. Also note that these methods were uncovered through our static analysis.

We also design the data structures maintained by JITANA to be the same as those maintained by the Android VM so that incoming information can be readily processed without performing conversions. For example, both JITANA and the Android VM (Dalvik in this case) maintain the same data structures to record classes that have already been loaded. This allows analysis to be performed more efficiently by incurring less “translation overhead” (e.g., converting data from one data structure to another). As in our prior example, when a new class is dynamically generated, JITANA can quickly detect the class loading event, and the newly loaded class is pulled from the device for onthefly analysis to create the necessary information.

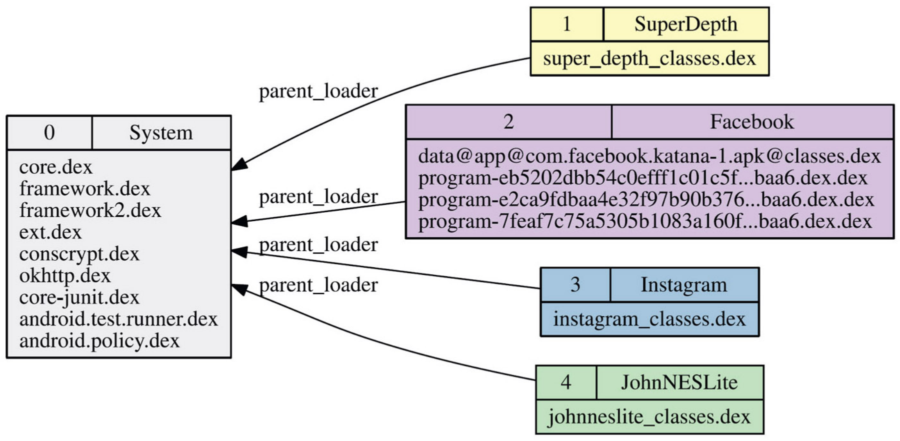

Fig. 4. Classloader graph after analyzing four large apps.

Scalability. Real-world security analysis processes may require an analyst to inspect multiple apps at the same time to uncover potential connections and hidden functionalities between two or more apps. Currently, existing approaches often analyze an app at a time and then compose the results. In contrast, by being memory efficient, our CLVM is able to analyze multiple apps at the same time. To support this capability, we exploit the explicit relationship between a classloader and the classes it loads to create a clear encapsulation of all classes in an application. For example, **Fig. 4** shows a classloader graph for four apps that have been loaded simultaneously as part of a single analysis: *SuperDepth, Facebook, Instagram*, and *JohnNESLite*. This property also allows our approach to naturally resolve class naming conflicts (i.e., two different classes with the same name), as these classes would belong to different classloaders.

The use of CLVM also provides an efficient way to analyze large libraries as previously shown in Figs. 1 and 3. Existing analysis tools need to analyze entire libraries to build graphs that can be used to supplement application analysis. Our approach analyzes only the classes within a library that have been referred to by the application, thus reducing the amount of analysis needed. As such, our framework analyzes classes from third party libraries as well as system classes. Also, JITANA does not require apps to be instrumented; runtime events are captured as they happen so analysis can be performed without interruption.

Extensibility. We want to ensure that classloader, class, method, and instruction graphs can be efficiently processed and persisted for future reuse without reanalyzing programs. Existing program analysis frameworks such as SOOT also produce graphs but the graph structures are built using Java objects and references. As such, designing analysis engines to process these graphs requires some knowledge of their structures to traverse these objects. The resulting engines can also be error prone due to memory reference faults (e.g., null pointer dereferences).

To make graph processing more extensible, JITANA employs a widely used, high performance BOOST Graph Library (BGL) as a graph processing engine [5]. BOOST provides existing implementations of advanced graph processing functions that are ready for use. The resulting graphs can also be persisted for future reuse. BOOST is also available on most computer systems. In addition, it is written in C++ and supports generic programming paradigms so code can be generically written to process different graphs.

3.1. Architectural overview of JITANA

To render our hybrid program analysis more efficient, we employ a classloader-based analysis approach instead of the traditional compilerbased analysis approach used by tools such as SOOT, which focus their analysis effort within the boundary of an application [4]. SOOT first loads the entire code of an app and performs analyses that includes construction of various program analysis graphs. By focusing only on the code within an app, it cannot analyze multiple apps simultaneously. This motivated Li et al. [16] to develop a workflow that uses SOOT twice: once to analyze each app for potential IAC sources and destinations and another to analyze the merged app. Compiler-based approaches also face difficulties handling large libraries because entire codebases must be analyzed. For example, AMANDROID, a framework introduced by Wei et al. [38], builds models of the underlying framework and libraries to facilitate program analyses. Our classloader-based static analysis approach, on the other hand, loads code in a fashion similar to that of an actual classloader inside any Java or Android VM; however, it uses reachability analysis to uncover classes.

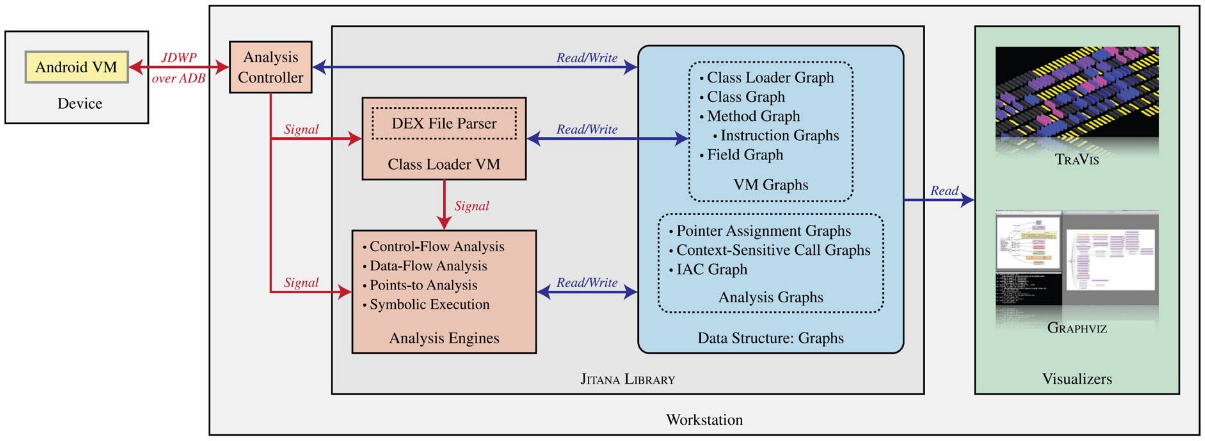

Fig. 5. Architecture of JITANA.

Fig. 5 provides an architectural view of the JITANA framework. We designed JITANA to be a highly efficient hybrid program analysis framework, so it needs to be able to interface with language virtual machines such as *Dalvik* and the *Android Runtime System* (*ART*). The VM interface is provided through the *Analysis Controller*, which is connected to the Android VM via *Java Debug Wire Protocol* (*JDWP*) over *Android Debug Bridge* (*ADB*) [24]. This connection is established primarily for use in dynamic analyses though it also assists with static analyses in cases in which code is dynamically generated during program initialization.

With our current implementation of JITANA, we are able to send runtime information that includes which basic block is being executed, which intent is being sent and received, and which class is being loaded. When dealing with dynamically loaded code, finding dynamic class loading information is critical. As such, JITANA specifically queries for information that includes the class name, method index, and class location to ensure precision. JITANA then pulls any new class from the device so that it has an exact copy on the analysis workstation. Next, we describe each component in Fig. 5, the underlying graph processing tool, and the programming paradigm used by JITANA.

3.2. Analysis controller

This particular component is used to communicate with the Android VM to support dynamic analysis. The initial version was implemented for Dalvik, wherein we add about 300 lines of code to Dalvik to create event notifications and a copy of each dynamically loaded class. The controller itself works with JDWP to make requests to the VM to provide the necessary data. For example, to capture a dynamically loaded class, Dalvik would notify the Analysis Controller about the class being loaded. The analysis controller would then issue the ADB pull command to retrieve the loaded class.

For our forthcoming version of JITANA, we are implementing the controller entirely using JDWP protocol, and without modifying ART, the latest Android VM. This provides greater portability. We anticipate that we will be able to release this version by the end of summer 2019.

3.3. Class-Loader Virtual Machine (CLVM)

We previously described the purpose of CLVM in JITANA. Now we describe its implementation. We implemented CLVM based on the Java Virtual Machine Specification [18]. A class, which is stored in a DEX file on a file system, must be loaded by a classloader. Each instance of `ClassLoader`, which is a Java class inherited from an abstract class `Ljava/lang/ClassLoader;`, has a reference to a parent classloader. When a classloader cannot find a class it delegates the task to its parent classloader. In the Android virtual machine, this process occurs as shown in **Algorithm 1**. Our approach employs a stand-alone CLVM to load reachable code based on the same algorithm.

```
Data: $N$: Name of the class to be loaded, $L_{\text{init}}$: Initiating class loader,
      and $\mathbf{D}_L$: An ordered set of DEX files for a class loader $L$.
Result: $\langle L_{\text{def}}, C \rangle$: A pair of defining class loader and pointer to a class or interface loaded.
1  begin
2      $L \longleftarrow L_{\text{init}}$;
3      $C \longleftarrow$ null;
4      do
5          foreach $D \in \mathbf{D}_L$ do
6              if $N \in$ class definitions list of $D$ then
7                  $C \longleftarrow$ address of loaded class;
8                  return $\langle L, C \rangle$;
9          $L \longleftarrow$ parent loader of $L$;
10     while $L \neq$ null;
11     return $\langle L, C \rangle$;
```

Algorithm 1. Class loading algorithm.

As previously mentioned, the explicit relationship between a classloader and classes that it loads creates a clear encapsulation of all classes in an application. Because CLVM operates in the same way as an actual classloader, it incrementally loads and analyzes classes. Our analysis approach has been designed to take advantage of this incremental nature of our system by employing work-list based algorithms to perform incremental reachability analysis. For example, our points-to analysis uses a pointer assignment graph (discussed in Section 3.4) that increases in size monotonically. This is because our system does not allow unloading of VM objects. As such, if a new node is added to the graph, our system simply adds it to the worklist and computes its reachability. This capability also allows JITANA to operate in an "open-world" fashion as any dynamically generated component or loaded component can easily be added to our analysis graphs and incrementally analyzed.

As an example of an "open-world" analysis, after we installed *Facebook* on a device, we discovered that it loads multiple additional DEX files dynamically. Our framework simply pulls these DEX files from the device directly for analysis. This means that constructing analysis context (e.g., control-flow, data-flow, and points-to graphs) is also performed incrementally. Thus, static and dynamic analysis contexts can be seamlessly integrated as static analysis can be employed to create the initial analysis context and dynamic analysis can *annotate* and *enhance* the initial context during runtime.

Once classes are loaded, the system generates a set of VM graphs (these VM graphs are representative of an app inside a VM) such as *classloader graphs, class graphs, method graphs*, and *instruction graphs*. Various *Analysis Engines* then process these to produce control-flow, data-flow and points-to information, which is then fed back in to the VM graphs. Other information is used to construct *Analysis Graphs* such as pointer assignment graphs, context-sensitive call graphs, and an IAC graph. The framework can run side-by-side with existing visualization tools such as GRAPHVIZ, an open-source graph visualization tool [9]. Next we describe these graphs in details.

3.4. Supported graphs

Most of the data structures used in JITANA are represented as hierarchical graphs. Typically, a node in such graphs represents a virtual

Table 1. JITANA graphs.

Name	*Type*	*Node*	*Edge*
Class Loader Graph	VM Graph	Class Loader	Parent Loader
Class Graph	VM Graph	Class	Inheritance
Method Graph	VM Graph	Method	Inheritance, Invocation
Field Graph	VM Graph	Field	
Instruction Graph	VM Graph	Instruction	Control Flow, Data Flow
Pointer Assignment Graph	Analysis Graph	Register, Alloc Site, Field/Array RD/WR	Assignment
Context-Sensitive Call Graph	Analysis Graph	Method with Callsite	Invocation
Inter-Application Communication Graph	Analysis Graph	Class Loader, Resource	Information Flow

machine object (e.g., a class, a method, an instruction) together with analysis information (e.g., execution counts), while an edge represents a relationship between two nodes (e.g., inheritance, control-flow, dataflow).

As shown in Fig. 5, JITANA separates data structures and algorithms. It represents programs as well-defined “Graph Data Structures”, and all “Analysis Engines” work on these. The core analysis algorithms are reusable and flexible because they are defined on concepts rather than concrete types. The data structures are also defined to be similar to those used in the actual virtual machine to reduce the overhead of exchanging dynamic information.

Table 1 lists the graph types currently generated by JITANA. There are two categories of graphs: *virtual machine (VM) graphs* and *analysis graphs*. Virtual machine graphs closely reflect the structure of Java virtual machines. A node in a virtual machine graph represents a virtual machine object (e.g., class, method) that can be created or removed only by the CLVM module in JITANA. Modification of a node property by an analysis engine is allowed and is one of the primary ways by which to track dynamic information such as code coverage. The edge type is erased with the `Boost.TypeErasure` library[2] so that analysis engines can add edges of any type. Examples of some of these graphs rendered with GRAPHVIZ have been previously shown in Figs. 2–4.

We briefly describe the classloader graph displayed in Fig. 4. Each class loader is assigned a unique ID (integers in the upper left corners)

2. http://www.boost.org/doc/libs/develop/doc/html/boost_typeerasure.html

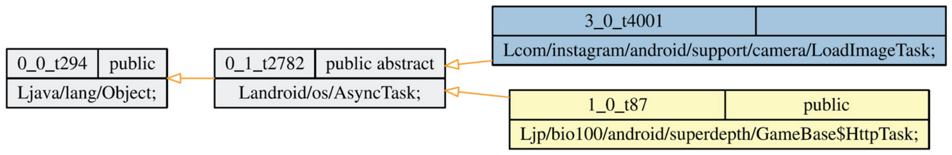
3_0_t4001
Lcom/instagram/android/support/camera/LoadImageTask;
0_0_t294
public
Ljava/lang/Object;
0_1_t2782
public abstract
Landroid/os/AsyncTask;
1_0_t87
public
Ljp/bio100/android/superdepth/GameBase$HttpTask;

Fig. 6. A class graph obtained by analyzing two apps.

so that classes with the same name from different apps can be distinguished. For example, both *Facebook* and *Instagram* ship a class named `Landroid/support/v4/app/Fragment;` with different method signatures because the Facebook app is obfuscated with PROGUARD;[3] therefore, the names of DEX files have no meaning. *Class-Loader 0* is the system class loader and is used to load necessary system classes. Each directed edge shows the parent/child relationship between two class loaders (e.g., the system class loader spawns off application class loaders).

Fig. 6 displays another *class graph*. This graph shows relationships between four classes; the directed edges display subclass relationships (e.g., `Lcom/instagram/... /LoadImageTask;` is a subclass of the abstract class `Landroid/os/AsyncTask;`). The figure also shows that `Ljp/bio100/android/superdepth/GameBase$HttpTask;` is a subclass of `Landroid/os/AsyncTask;`).

In Fig. 2, the instruction graph of a method also includes control-flow and data-flow information. The data-flow information is derived via reachability analysis performed on virtual registers. Data-flow information is presented in Fig. 2 as red dotted edges. Control-flow information is presented as blue edges.

Not depicted in any figure are field graphs. *Field graphs* store a list of fields as nodes, but by default JITANA does not add edges to this graph. This data can still be used for analysis purposes.

Fig. 3 also shows relationships among several methods within a set of analyzed apps. The numbers in the upper left corners of the nodes indicate the apps to which the methods belong. Nodes represent methods, and edges indicate whether method calls are *direct* or *virtual*. For virtual methods, we also report inheritance relationships

3. https://www.facebook.com/notes/facebook-engineering/under-the-hooddalvik- patch-for-facebook-for-android/10151345597798920

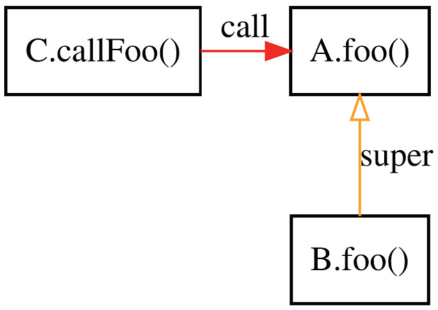

Fig. 7. An Illustration of Super Edge.

using edges labeled as "super". A method that inherits from another method has a super edge pointing back to the original method. For example, suppose we have the following three classes.

```
class A{
  public void foo(){
  }
}
class B extends A{
  public void foo(){
  }
}
class C {
  public void callFoo(A target){
      target.foo();
  }
}
```

The method belonging to these three classes would be depicted as shown in **Fig. 7**. As shown, there is a super edge pointing from `B.foo()` to `A.foo()`. The super edges in the method graph give us more comprehensive view (i.e., considering vtables) of a call graph without needing a class graph.

3.5. Supported analysis engines

Currently, JITANA supports interprocedural control-flow, intra-procedural data-flow, and points-to analysis graphs [36]. In Java, most function calls are made by using a dynamic dispatch mechanism. Therefore, knowing the actual type of an object in a pointer variable is essential for any interprocedural analysis. JITANA also supports a points-to analysis algorithm inspired by SPARK [14], a points-to analysis framework in SOOT [36].

JITANA's *interprocedural control-flow analysis* is based on Class Hierarchy Analysis (CHA) [6]. It includes both direct call edges and virtual call edges in a method graph as shown in Fig. 3 and explained previously. The approach is sound but imprecise due to over-approximation, a common issue for static analysis.

In terms of *data-flow analysis*, JITANA supports reaching definitions analysis, which is used to generate def-use pairs. The monotone dataflow algorithm used in JITANA's reaching definitions algorithm is implemented as a generic function; thus it can be used to perform other types of data-flow analyses such as available expressions or live variable analysis by defining appropriate functors. It also works on any graph types that model the concepts required for control-flow graphs.

The *IAC analysis* used by JITANA operates in multiple steps. In the first step, it searches for potential connections (entry and exit points) between components and apps; this is done by analyzing the code and manifest files. Note that this first step is similar to that used in EPICC [22]. Because JITANA analyzes a collection of apps all at once, all IAC connections within that set of apps appear in a single analysis effort. This is the main goal that APKCOMBINER tries to achieve by combining apps.

In the second step of JITANA's IAC analysis, when an exit point that can connect to an entry point in another app is uncovered, a connecting edge in an IAC graph is created. **Fig. 8** illustrates an IAC graph resulting from analyzing a bundle of 21 apps. There are seven pairs of apps that have intent connections between them. Note that each app in a collection being analyzed represents a node in the graph. As such, two connected apps can have multiple edges between them (e.g., app 20 and app 21 have two intent connections).

Currently, JITANA does not support interprocedural distributed environments (IDEs). However, once we implement it in JITANA, dataflow analysis can be performed across these IAC edges to increase analysis precision as previously done in EPICC and IC3.

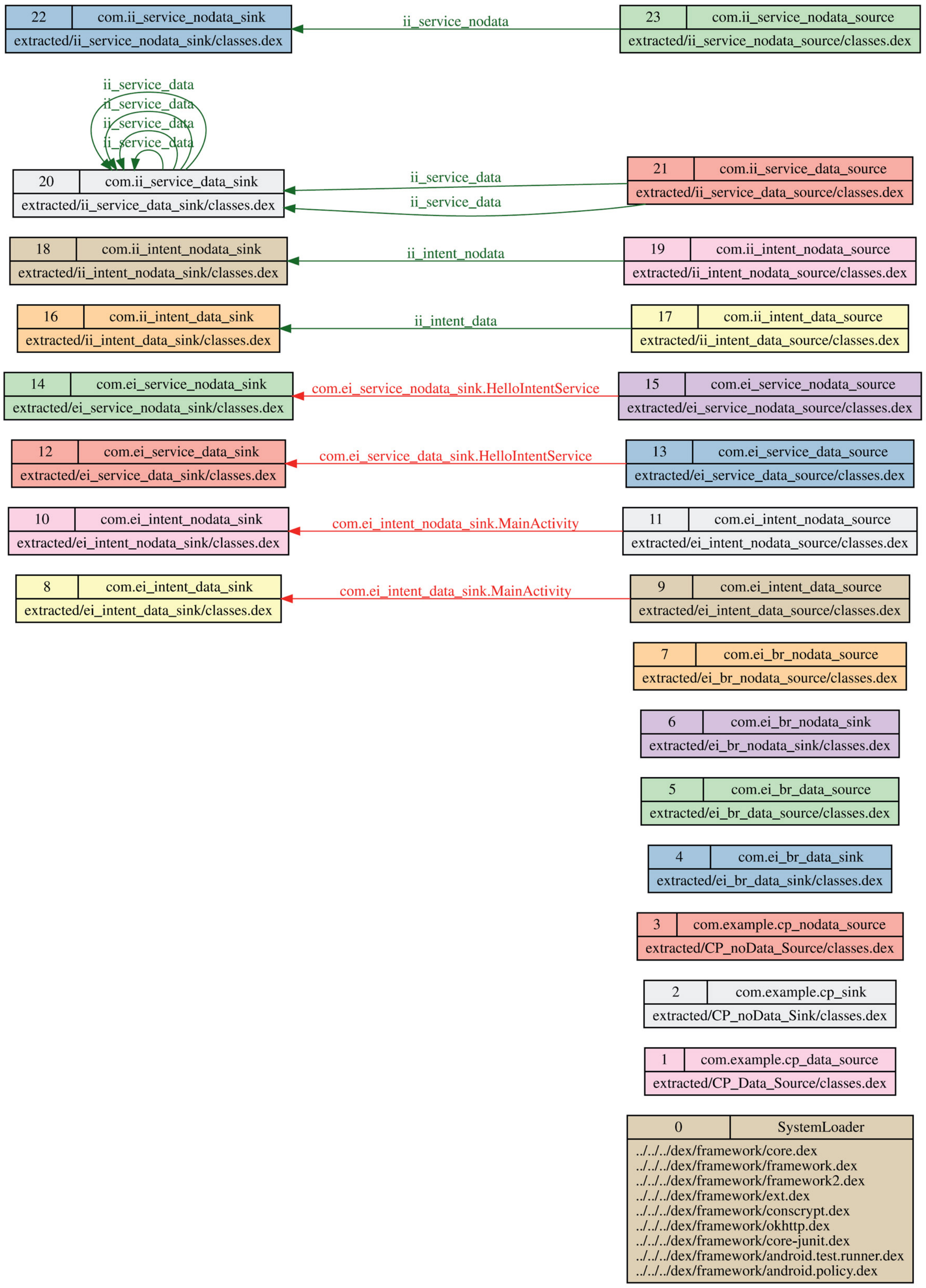
22 com.ii_service_nodata_sink
extracted/ii_service_nodata_sink/classes.dex
ii_service_nodata
23 com.ii_service_nodata_source
extracted/ii_service_nodata_source/classes.dex
ii_service_data
ii_service_data
ii_service_data
ii_service_data
20 com.ii_service_data_sink
extracted/ii_service_data_sink/classes.dex
ii_service_data
ii_service_data
21 com.ii_service_data_source
extracted/ii_service_data_source/classes.dex
18 com.ii_intent_nodata_sink
extracted/ii_intent_nodata_sink/classes.dex
ii_intent_nodata
19 com.ii_intent_nodata_source
extracted/ii_intent_nodata_source/classes.dex
16 com.ii_intent_data_sink
extracted/ii_intent_data_sink/classes.dex
ii_intent_data
17 com.ii_intent_data_source
extracted/ii_intent_data_source/classes.dex
14 com.ei_service_nodata_sink
extracted/ei_service_nodata_sink/classes.dex
com.ei_service_nodata_sink.HelloIntentService
15 com.ei_service_nodata_source
extracted/ei_service_nodata_source/classes.dex
12 com.ei_service_data_sink
extracted/ei_service_data_sink/classes.dex
com.ei_service_data_sink.HelloIntentService
13 com.ei_service_data_source
extracted/ei_service_data_source/classes.dex
10 com.ei_intent_nodata_sink
extracted/ei_intent_nodata_sink/classes.dex
com.ei_intent_nodata_sink.MainActivity
11 com.ei_intent_nodata_source
extracted/ei_intent_nodata_source/classes.dex
8 com.ei_intent_data_sink
extracted/ei_intent_data_sink/classes.dex
com.ei_intent_data_sink.MainActivity
9 com.ei_intent_data_source
extracted/ei_intent_data_source/classes.dex
7 com.ei_br_nodata_source
extracted/ei_br_nodata_source/classes.dex
6 com.ei_br_nodata_sink
extracted/ei_br_nodata_sink/classes.dex
5 com.ei_br_data_source
extracted/ei_br_data_source/classes.dex
4 com.ei_br_data_sink
extracted/ei_br_data_sink/classes.dex
3 com.example.cp_nodata_source
extracted/CP_noData_Source/classes.dex
2 com.example.cp_sink
extracted/CP_noData_Sink/classes.dex
1 com.example.cp_data_source
extracted/CP_Data_Source/classes.dex
0 SystemLoader
../../../dex/framework/core.dex
../../../dex/framework/framework.dex
../../../dex/framework/framework2.dex
../../../dex/framework/ext.dex
../../../dex/framework/conscrypt.dex
../../../dex/framework/okhttp.dex
../../../dex/framework/core-junit.dex
../../../dex/framework/android.test.runner.dex
../../../dex/framework/android.policy.dex

Fig. 8. An IAC graph.

4. Using JITANA to perform at-speed security analysis

In this section, we provide four scenarios in which security analysts can use JITANA to perform at-speed security analyses.

4.1. Using JITANA to identify apps with IAC connections

In this scenario, we use JITANA to statically analyze collections of real-world Android apps for IAC connections. We conducted this investigation to determine whether JITANA is sufficiently robust to handle large commercial apps without any issues.

4.1.1. Procedure

For security analysts to use JITANA for their daily security needs, it must be robust and be able to analyze real-world apps that are available through various distribution channels such as Google's Play store. As such, to assess JITANA's robustness, we used real-world applications available for free in the Google Play Store. Our selected apps were all on the top 100 chart during January 2017. We focused on three categories of apps due to their popularity, and as groupings of apps we chose apps from the same category because they are more likely to share components, rendering IAC connections more complex. We chose 10 apps from the "social network" category including *Facebook, Snapchat*, and *Instagram*, and 10 apps from the "games" category including *Pokémon Go, Suicide Squad*, and *NBA Live*. The third category is "miscellaneous". Notable apps in this category include *Spotify, Google Photos, Pandora*, and the top seven additional apps in the top 100 chart that do not belong to the first two categories. **Table 2** lists all of the apps in each category.

We next attempted to use APKCOMBINER to merge these 30 apps.[4] Because APKCOMBINER merges two apps at a time, we attempted to find pairs of apps in each category that it could merge. Unfortunately,

4. As previously mentioned (Section 2), APKCOMBINER is a tool designed to help with IAC analysis. Work by Li et al. [16] reported that APKCOMBINER was able to successfully combine over 2600 pairs of real Android apps. We initially considered using these apps but the authors provided no information about them (except for *Edabah Evaluation* and *ClipStore*), so it was impossible to find them. We did investigate *Edabah Evaluation* and *ClipStore* and found that they were both released prior to 2012 and have not been updated for at least four years. As such, they were too old to reveal insights that may be applicable to modern apps.

Table 2. Apps downloaded from Google Play for scenarios 1 and 2.

Category	*Listing of Apps*
Social Network	Facebook, TextNow, Snapchat, Instagram, Pinterest, Twitter, ooVoo, POF Free Dating, Tumblr, Tango
Games	Pokémon Go, Slither.io, Rolling Sky, Subway Surfers Color Switch, Roblox, Suicide Squad, NBA Live Farm Heros Super Saga, Geometry Dash
Miscellaneous	Messenger, YouTube Music, Pandora, Spotify, Wish Google Photos, News Master Topbuzz, Mercari Marco Polo Walkie Talkie, Remind

APKCOMBINER was not able to merge any pair of apps in any of our three categories. For the social network and miscellaneous categories, it simply failed to produce any merged apps for any combinations. In some cases, it initially appeared to be successful but when we inspected the merged apps they contained errors and their output sizes were too small to be valid. As such, we could not gather any data with which to evaluate the ability of APKCOMBINER to handle large apps.

Next, we turned our focus to EPICC and IC3 and attempted to use them to search for entry/exit points in our collection of apps so that we could compare the results produced by these systems with those produced by JITANA. Unfortunately, both EPICC and IC3 were unable to return results for the apps. We also encountered a few instances in which both systems simply crashed. One possible reason for these problems involves failures during code retargeting, which has been previously reported [22]. As a second possible reason, our experiment objects are from a different generation of apps than those previously used to evaluate these systems (prior apps used were from 2013) [20–22], and the systems might simply not be able to handle the characteristics of this newer generation of apps. Finally, because both systems achieve their precision through the use of the IDE framework and a precise points-to analysis in SOOT, it is also possible that analysis graphs in these apps may be too complex to be feasibly produced.

We next used JITANA to simultaneously analyze all ten apps in each category to produce analysis graphs (control-flow and data-flow graphs), compute IAC connections, and generate the IAC graph for each category. We envision that this particular technique can be useful in a scenario where a security analyst wants to compute IAC connections between a set of known apps anda particular app (e.g., an app developed by an organization for their employees to use in the field) to identify possible leakage of information. A typical approach that relies on pair-wise analysis would be required to perform 45 analysis attempts to evaluate possible connections among the 10 apps. JITANA, on the other hand, requires only one attempt. We measured the time required to complete this process.

4.1.2. Results

Table 3 reports results relative to our first scenario. The first column indicates the configuration used to evaluate JITANA; that is, it reports the number of apps being simultaneously analyzed. The second column reports the combined size of the APKs. Table 2 lists the apps that we used in this study based on category, and we combined the apps in the order in which they are listed. For example, the first row of the social network category includes the first two apps from that category: *Facebook* and *TextNow*. The combined size of the two APKs is 76 MB. The third column reports the number of classes that have been loaded by *CLVM* and analyzed by JITANA. The last column reports the time by needed by JITANA to analyze all the apps in each collection.

As the data show, JITANA successfully analyzed all sets of between two and ten apps in each of the three categories. It also processed

Table 3. Scenario 1: Analysis data for JITANA.

# of Apps	*Size of APKs (MB)*	*# of Loaded Classes*	*Time (sec)*
Social Network			
2	76	9,154	38.64
4	144	24,253	81.37
6	179	38,355	109.28
8	295	53,923	152.06
10	351	65,399	176.37
Game			
2	79	14,023	26.82
4	173	27,929	51.98
6	298	38,210	71.62
8	391	53,588	107.01
10	452	68,179	128.85
Miscellaneous			
2	22	14,719	29.75
4	87	25,972	59.10
6	124	45,281	93.75
8	157	57,896	136.61
10	191	72,717	167.85

large amounts of code, ranging from 22 MB to over 450 MB. The number of classes loaded and analyzed by JITANA exceeded 50,000 in several cases. The analysis times ranged from 27 s to 176 s. Note that these times also include the time required to compute IACs.

4.2. Analyzing all apps in a device

In this scenario, we consider an increasingly common scenario in which organizations allow their employees to bring their own devices to work. In this scenario, security analysts working for the organization wish to vet employees' devices to detect whether there are any connections that may allow apps already installed on a device to communicate with the company's own apps directly or indirectly (through a shared component). Since these devices would most likely have different apps installed on them, the analysis must be done quickly to determine IAC connections as employees check in their devices.

4.2.1. Procedure

We applied JITANA to three non-trivial real devices (Asus Nexus 7s), using different numbers of installed apps. These devices communicate with JITANA through USB ports, allowing it to query and download the APKs installed on each device directly. JITANA then simultaneously analyzed all of the apps on each device to build analysis contexts, computed the IAC connections among the apps, and generated the IAC graph.

We configured our three devices to have 99 apps (the default apps and system services that come with a new installation of Android 5.1.0), 114 apps, and 129 apps, respectively, with the second and third devices including the apps that were present on the first and second devices, respectively. These additional apps include the 30 apps used to investigate the robustness of JITANA (15 apps installed on the second device and 15 additional apps installed on the third device). We then measured, for each device, the times required to pull the apps from the device to the workstation running JITANA, to perform simultaneous analysis to build analysis contexts, to compute IAC connections, and to produce the IAC graph for the device.

Note that no existing approaches are capable of performing this task at this scale; thus, in this study we cannot compare JITANA to any

Table 4. Scenario 2: Analysis data for JITANA, using three devices with different numbers of Apps.

ID	*# of Apps*	*Total Size (MB)*	*# of Classes (×1K)*	*# of Methods (×1K)*	*Apps with Intents*	*Apps with Content Providers*	*Time (sec)*
1	99	340.7	126.0	480.3	63	0	379
2	114	848.1	215.8	1,313.5	77	11	674
3	129	1,357.2	320.7	2,001.3	92	21	1147

baseline approach. However, if we were able to do so, the baseline approach based on APKCOMBINER used in the prior study would need to perform 4851, 6441, and 8256 pair-wise IAC analyses for the three devices, respectively.

4.2.2. Results

Table 4 reports our results. Columns 2–4 provide data on the device configurations used to evaluate the scalability of JITANA, including the number of apps, the amount of disk space needed to store these apps on the device, the number of classes, and the number of methods. Other columns report the number of apps that we found to be connected using intents and content providers, and the amount of time needed to analyze the entire device for IACs and construct analysis graphs.

As the data shows, JITANA was able to analyze a device with 129 apps in about 19 min (Device 3). For Device 3, JITANA loads and analyzes over 320,000 classes, which amounts to about 1.35 GB of code. For the two devices with smaller numbers of apps (Devices 1 and 2), it was able to analyze all apps on each device in about 6 min and 11 min, respectively. It is also worth noting that each app on a device has a tendency to connect with other apps. The default apps use only intents for IACs but other downloaded apps use content providers in addition to intents for IACs. We also observe that apps use intents more often than content providers. In cases in which an app uses both intents and content providers, the app is reported in both Columns 6 and 7. This behavior clearly emphasizes the need to analyze interacting apps efficiently.

4.3. Real-time code coverage

In this scenario, we consider an example in which jitana supports the creation of a real-time visualization engine to provide real-time code coverage information. We imagine that such a scenario can be useful for analysts who wish to quantify the quality of their vetting effort. This information can be used to determine whether the dynamic analysis should continue or has sufficiently explored the code. A security analyst can also visualize code sections that may be difficult to reach so that more effort can be spent trying to reach them. In addition, to assess the overhead we run our code coverage analysis while an interactive game is being played.

4.3.1. Procedure

We applied jitana to provide real-time code coverage. To measure code coverage in Android, emma is still commonly used. emma was initially created as a code coverage tool for Java, but it now also works for Android. It supports both targeted unit testing and random testing using monkey, an Android UI exerciser.[5] The largest difference between jitana and emma is that emma requires instrumentation of the application and then post processing of the generated runtime logs. jitana, on the other hand, can observe runtime events without instrumentation.

emma first instruments classes for coverage; this can be done offline or on-the-fly through a custom class loader. The latter approach is used in our study to render the entire process seamless. emma is a low-overhead coverage tool that incurs only 5%–20% overhead. The main benefit in using it lies in its ease of deployment and the fact that no instrumentation is needed. Coverage can be measured at different granularities including class, method, line, and basic block. emma then produces a report file in plain text, HTML, and XML. Currently, jitana can measure code coverage. However, due to our research needs, we measure code coverage only at the basic block level, and we report coverage only in tab-delimited plain text. These limitations can easily be addressed; however, they lead us to compare the two approaches only at the basic block level using plain text reports.

5. http://developer.android.com/tools/help/monkey.html

Table 5. Comparing execution times of code coverage measurements between EMMA and JITANA.

Application	*LoDC*	*No Coverage (seconds)*	*EMMA (seconds)*	*JITANA (seconds)*	*Coverage (%)*
Calculator	534	49	54	51	80
ColorMatcher	706	244	246	248	88
CountdownTimer	1265	146	150	145	83
MorseCode	600	239	246	248	94
SplitTimer	166	50	56	53	99

4.3.2. Results

With EMMA's low runtime overhead, we do not anticipate any significant difference in the runtime performance between an app that is running with JITANA performing analysis in the background and the same app that has been instrumented to include the counting methods. To verify this, we identified a set of benchmarks with source code from the DARPA APAC project.[6] These benchmarks have unit test suites. **Table 5** lists these benchmarks together with their lines of Dex code (LoDC). We measured the time required to execute an application with the unit test suite provided and no code coverage measured (Column 3), the time required to execute the same test suite with EMMA enabled (Column 4), and the time required to execute the test suite with JITANA enabled (Column 5). The times are reported in seconds. As shown, the runtime overheads of both systems are very low. The slight differences can be attributed to small runtime variations.

In addition, due to the event-based nature of these apps, the dominating costs of running them are the delays between instigating pairs of events. Such delays, in fact, hide the cost of transferring information from the actual device to the workstation running JITANA. EMMA on the other hand, cannot fully take advantage of these delays as the processing cost is incurred afterwards. However, it can still hide the cost of generating traces behind these delays.

The main advantage of JITANA in this context, however, is that code coverage analysis is done on-the-fly so that information can be generated and efficiently used in real time to provide feedback to security analysts or automated vetting systems such as the cloud-based system used by Google. EMMA, on the other hand, requires a post-processing

6. These benchmarks are available from http://cse.unl.edu/~ytsutano/jitana.

step to combine dynamic information with prior static analysis results to generate code-coverage reports. This separate post-processing step renders EMMA unsuitable for at-speed security analysis.

To demonstrate the real-time feedback capability of JITANA, we developed a visualizer to display coverage information *in situ*. We next describe the process we followed to create our visualizer, TRAVIS, as a plugin of JITANA. We then illustrate the capabilities of TRAVIS by showing how it can process the execution information sent by Dalvik to measure code coverage.

Via a JDWP connection, TRAVIS periodically receives the dynamic execution information necessary to visualize the traces from the Dalvik VM that was modified as described in Section 3.1. As soon as a DEX file on the device is loaded on the virtual machine, TRAVIS is notified with the file name. Upon notification, TRAVIS: (i) Copies the loaded DEX file from the device to the workstation using an `adb pull` command; (ii) creates a buffer in which to store counter values for the DEX file; and (iii) lets JITANA load the DEX file to update the VM graphs.

TRAVIS also polls for counter values every 50 milliseconds. The values are sent as an array of pairs of an instruction offset and the number of times it has been executed since the last poll. The counter values are accumulated in the buffer that is created when the DEX file is loaded. The instruction graphs are updated with the new counter values from this buffer. This data is presented as traces on screen with OpenGL renderings, and as instruction graphs are rendered in a GRAPHVIZ viewer.

Fig. 9 illustrates how TRAVIS can be used by a security analyst. A device (Nexus 7 in this case) is first connected to a workstation. Runtime information is sent from the device to a workstation running JITANA. In the figure, a person is playing *SuperDepth*, a classic video game (shown in the lower right quadrant). While the user plays, Dalvik sends execution information on-the-fly to JITANA, which processes the information to calculate code coverage, which is then fed to TRAVIS. The application requires no instrumentation.

In Fig. 9, the upper left quadrant displays the method graph for *SuperDepth* and the upper right quadrant displays two instruction graphs. Shaded boxes indicate entry instructions in basic blocks. In each such box, there is also a counter to indicate the number of times the basic block has been executed. For example, the block highlighted by an ellipse has been executed 20 times. The block above

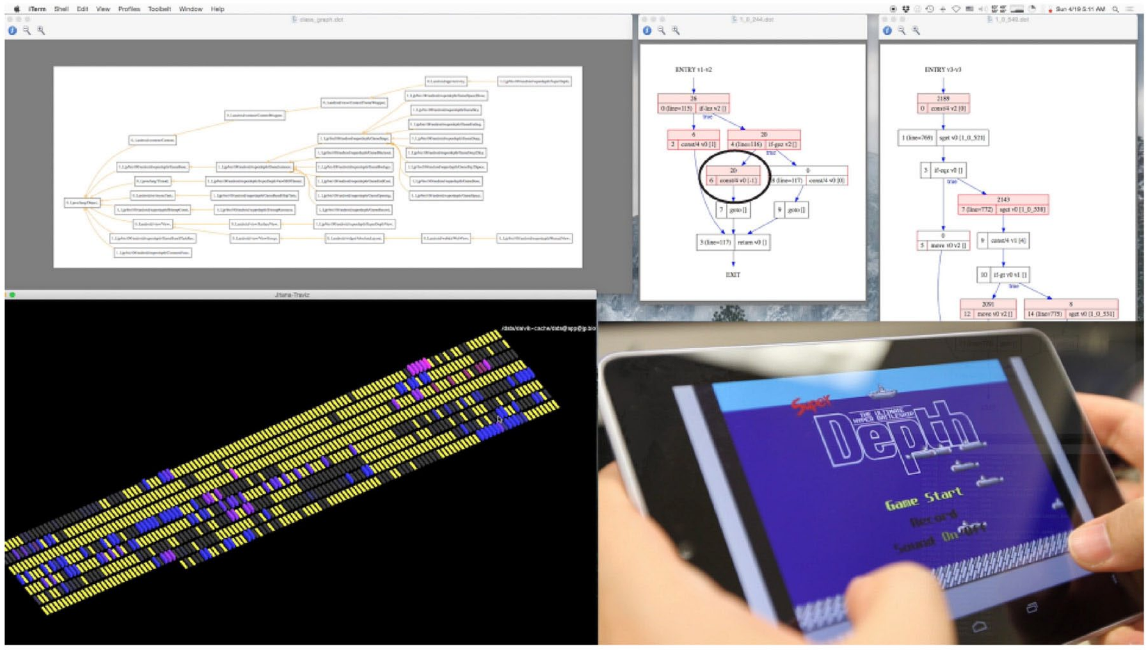

Fig. 9. In-situ visualization with TRAVIS.

that corresponds to a conditional statement and so far, all decisions have taken the left branch. These counters are continuously updated.

The bottom left quadrant shows the output of TRAVIS. Each small rectangle on what looks like a "keyboard" in the figure represents a basic block. On a color display (of this paper or of the output of TRAVIS), black rectangles indicate basic blocks that have not yet been executed, blue rectangles indicate "hot" basic blocks (i.e., basic blocks that have been executed more than five times), magenta rectangles indicate basic blocks that are currently being executed, and yellow rectangles indicate basic blocks that have been executed fewer than five times. (On a blackand- white printout, the colors range from dark gray to light gray with two intermediate shades.) The video clip that the images have been captured from is available at https://www.youtube.com/watch?v=sPdrLdIKDx4 .

By using TRAVIS, security analysts can obtain real-time feedback with respect to the quality of their dynamic analysis. For example, applying JITANA in conjunction with the dynamic analysis used by Google's Cloudbased security analysis would provide additional real-time data to show how much code is covered by the current vetting effort via dynamic analysis. If an analyst finds that the code coverage percentage is still too low for the analysis to be meaningful, the analyst can decide to take additional steps to perform better risk assessment. For example, the analysis may favor the result generated by static analysis over that generated by dynamic analysis or add additional time to execute the app and perform dynamic analysis. This information is provided without dynamic analysis overhead.

4.4. Detecting and analyzing dynamically loaded code

In this scenario, we show how the dynamic analysis component of JITANA can be used to capture dynamically loaded code and how such code can be continuously analyzed by the static analysis component of JITANA. We illustrate such a usage scenario by utilizing JITANA to capture Dex files dynamically loaded by *Facebook*.

4.4.1. Procedure

We extended TRAVIS's capability to pull loaded Dex files from a device for JITANA's use in performing analyses. With this capability, we can easily detect dynamically loaded code and then use CLVM to load these classes to perform static analysis. To illustrate this capability, we employed *Facebook*. As previously mentioned in Section 2, *Facebook* loads additional classes at runtime. To execute the app, we first tried to use Monkey but it became stuck on the login screen. As such, we asked our last author to use *Facebook* for 5 min to provide sufficient time for it to run. We then tried to capture those dynamically loaded classes as we ran the app.

4.4.2. Results

In apps that dynamically load additional code, applying static analysis alone is ineffective as dynamically loaded code is often not available at static analysis time. For example, were we to apply JITANA to perform static analysis of *Facebook*, it would be able to analyze only 3918 classes containing 18,985 methods. However, by capturing three additional Dex files at runtime, Jitana ultimately analyzed 23,621 classes containing 130,428 methods. In other words, we would miss nearly 75% of the actual loaded classes. It is also worth noting that a newer version of the Facebook app may create even more than three additional Dex files.

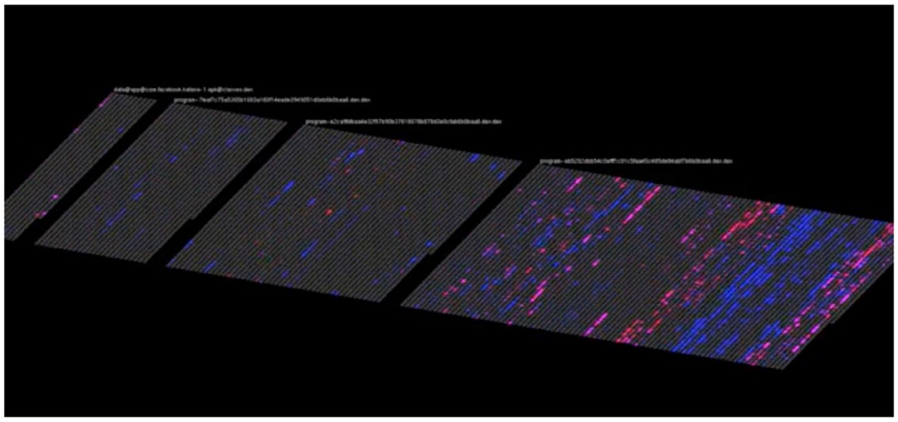

Fig. 10. Illustration of Facebook loading four Dex files.

In terms of time, JITANA required 2.11 s to analyze the initial 3918 classes to generate control-flow and data-flow information. With dynamic analysis, JITANA was able to analyze these 23,621 classes in 12.88 s. In comparison, SOOT takes 181.35 s to produce control-flow and data-flow information for classes discovered during static analysis alone.

We also used TRAVIS to visualize code coverage on the additional Dex files. **Fig. 10** illustrates the additional Dex files. When we compare Fig. 9 with 10, we see that there is only one Dex file loaded in *Super- Depth* (Fig. 9, because there is only one cluster of rectangles. However, Fig. 10 shows four clusters of rectangles; each cluster represents a loaded Dex file. The actual video used in the figure is available at https://www.youtube.com/watch?v=L0ngqsrTOHU .

With respect to security analysis, we envision that JITANA can be useful for detecting and analyzing a new breed of malicious apps that hide their malicious behaviors by dynamically loading code at runtime. There are several samples of this type of malware such as *Obad* [11,34], *Charger* [32], *Shedun* [26], and *FakeInstaller* [30] that have already been exposed. For example, *Shedun* uses a dynamically loaded component to leak sensitive information through the *Messenger* app. It uses the *Dyn- LoadService* component to load a malicious class from an external JAR file hidden in the APK. It then uses the classloader to load a Dex file contained in the JAR file. Its malicious behavior is to use intent to steal user's sensitive information. When we used JITANA to execute a version of Shedun, we were able to capture the malicious class file. We also observed the new class file being loaded at runtime via TRAVIS. When we analyzed the newly loaded class, we also found that it used sensitive APIs that appear in the widely used *SuSi* list [27], indicating that the app is maliciously behaving.

4.5. Threats to validity

The primary threat to external validity in this investigation involves the object programs utilized. In our first scenario, we employ complex real-world apps including *Facebook, Instagram, Twitter, Pandora*, and *Pokémon Go*. This allows us to apply our system to real-world scenarios, representative of those that engineers and analysts face. The same is also true for our second scenario wherein we use common apps that are packaged as part of an Android distribution in addition to real-world apps used in the first scenario. In our third scenario, we need to perform dynamic analysis and compare the results with EMMA. To do so, we need to use apps that have test suites that achieve good code coverage. This prevented us from using commercial apps, because generating inputs for those apps is non-trivial. For example, random event sequence generation tools such as Monkey tend to fail to exercise complex apps due to their needs for specific user inputs such as log-in information, accepting terms of use, and so forth. As such, we turned to opensource apps that had unit test cases. For the last scenario, we needed to find apps that load additional Dex files at runtime. Because Facebook is widely popular and its mechanism for loading additional Dex files is complex, we chose it.

A second threat to external validity involves the representativeness of our techniques for identifying IAC connections, providing code coverage information, and detecting dynamically loaded code. With respect to identifying IACs, we attempted to use the state of the art approach, IC3, to produce IAC analysis results to use in our study. However, IC3 was unable to analyze several real-world apps. As such, we used our IAC analysis implementation to provide the apps that APKCombiner needs to combine. As reported, APKCombiner was not able to combine any of these real-world apps. For the apps that IC3 and JITANA can analyze for IACs, we also compare the results to ensure that for the types of IAC connection that we can support, our results do not contain any false negatives. (False positives are expected as our analysis is not as precise as that of IC3.) With respect to code coverage, we can use only existing tools that work directly on Dex code.

This is because in real-world settings, analysts may not have access to the source code of apps that they are trying to analyze. As such, we chose EMMA. However, EMMA cannot perform continuous real-time analysis so while its runtime overhead is low, it needs to finish executing an app before it can compute the code coverage.

The primary threat to internal validity on this work involves potential errors in the implementations of JITANA and the infrastructure used to run the baseline tools and JITANA. To limit these, we extensively validated all of our tool components and scripts.

The primary threat to construct validity in this work relates to the fact that we study efficiency measures relative to applications of JITANA, but do not yet assess whether the approach helps engineers and security analysts address dependability and security concerns more quickly than current approaches.

5. Discussion

One interesting observation from our investigation of Scenarios 1 and 2 is that the amount of time required by JITANA to perform text processing for IAC analysis is negligible. Parsing an app for intents, intent filters, and content providers represents the first step in the IAC analysis process. Existing approaches such as EPICC, IC3, and AMANDROID use the commonly available Android APKTOOL to parse these XML files for analysis. Our experience has shown that APKTOOL is inefficient because its goal is to provide human-comprehensible results. Instead, we developed our own tool to directly parse the desired information which is typically stored in binary format. This allows JITANA to perform string processing with low overhead. Comparing the string processing times between APKTOOL and JITANA, we find that JITANA was more than 30 times faster.

With respect to precision, JITANA currently yields IAC analysis results that often contain more false positives than results produced by EPICC, and by IC3, which is an improvement over EPICC. This is because a specific request can be mapped to multiple intent filters. EPICC overcomes this problem by employing the IDE framework [28] to reduce false positives. IC3 further improves the precision of EPICC by employing composite constant propagation to reduce the number of identified intent filters by including URI information (see Section 6 for more information).

On the other hand, both epicc and ic3 are ICC analysis tools, so in order for them to be applicable in the IAC context, the use of apkcombiner is necessary. As we have shown, apkcombiner is not scalable or robust, so techniques based on it would have limited applicability. jitana does not suffer from this scalability issue. We are now extending jitana to support interprocedural dataflow analysis [2] and more complex constant propagation [21] so that we can increase the precision of our IAC analysis.

We further investigated the impact of false positives in our results and found that in most cases, jitana uncovered real connecting edges as well as false positive edges between pairs of apps. Therefore, the final results, with respect to app connectivity, are still accurate. However, we also found some instances in which false positives led jitana to mistakenly identify connections between some pairs of apps. For the time being, if higher precision can be sacrificed for higher efficiency and scalability, jitana can already be used to determine IAC connections.

With respect to code coverage, being able to assess test input quality in real time can provide analysts and engineers with timely information that would allow them to make decisions more quickly with respect to ending the process. Instead of using time as the main criterion for setting an at-speed security vetting budget, code coverage can also be used to determine whether vetting has been sufficient. In addition, our apprach can also keep track of execution "hotness" by counting how many times each basic block has been executed. As such, the result generated by our approach can be used to guide an input generation process to try to reach hard-to-reach blocks. If an input can be generated quickly, the input generation and code coverage measurement can be done in a continuous feedback loop, rendering testing and security vetting more efficient.

With respect to capturing dynamically loaded code, the efficiency of our approach is ideally suited to analyzing malware that exploits dynamic code loading and reflection mechanisms in Android to deliver malicious behaviors at runtime. Because our approach can capture such code and incrementally annotate the new analysis result, it allows vetting for this type of malware to be done more effectively and efficiently.

Finally, as noted in Section 4.3.2, imagine a company that allows its employees to bring their own devices to work. It is conceivable that there would be a team of analysts who need to vet devices before admitting them to the company's network or installing the company's apps. When a device is submitted for vetting, an analyst may need to observe how some of the key apps or services are used by other existing apps on the device. Perhaps there is a malicious app that can compromise data sent through a common service. A recent study [33] indicates that a smart-mobile device, on average, contains about 95 apps. Based on the performance reported in Table 4, an analyst can use JITANA to perform analysis very quickly as part of an employee check in. Efficient analysis such as this would also allow security checks to be performed more frequently.

6. Related work

Research efforts that are related to this work include program analysis frameworks for Android and detection of IAC connections. Over the past few years, there have been several approaches introduced for analyzing systems for ICCs [17,21,22,38]. EPICC is a program analysis framework that uses interprocedural data-flow and points-to analysis to uncover ICC connection points. On average, ICC analysis times for real-world apps range from 38 s for apps with fewer than 400 classes to 144 s for apps with 1500 classes or more. Our approach, however, can analyze an app with approximately 400 classes in less than 2 s and an app with approximately 200 classes in about 11 s.

IC3 [21] is an improvement over EPICC. It conducts URI analysis for Content Providers and Multi-Valued Composite (MVC) constant propagation. In MVC constant propagation analyzers try to target complex objects that may have multiple fields. AMANDROID [38] performs flow and context sensitive points-to analysis. This can help with the construction of precise interprocedural control flow graphs (ICFGs) and interprocedural data-flow graphs (IDFGs). The approach builds a data dependence graph (DDG) for theapp from an IDFG. ICCTA goes a step further by taking the ICC information and then performing taint analysis to detect possible malicious flows across components [17]. In terms of performance, IC3 can analyze a corpus of 460 real-world Android apps (downloaded in 2013) for ICC connections in about 18 h – an average of 140 s per app. Because our framework does not yet support MVC we cannot compare our performance with that of IC3, so we leave this evaluation as future work.

ic3 has also been used as the base system for further optimization through probabilistic models. Octeau et al. [20] apply probabilistic models to further resolve intent and intent filter matching to reduce false positives. First, IC3 is used to perform ICC analysis. Intent resolution is then used to further reduce ICC connections. The author then conducted an evaluation using a collection of 11,267 apps and malware samples. They did not report the time required by IC3 to initially analyze these apps, but the intent resolution time was 43 min [20]. Note that the intent resolution process is orthogonal to that of IC3 and can also be used to increase the precision of ICC results produced by jitana.

Taint analysis has been used to verify IAC connections and detect potentially suspicious connections. One such example is provided by iccta [17]. iccta is used to support IAC analysis. epicc or ic3 is first used to perform ICC analysis on each app (Step 1) and analyze for the connections that can occur across apps (Step 2). apkcombiner is then used to combine the connected apps so that soot can reconstruct the analysis graphs and flowdroid [2] can perform taint analysis on the resulting app [16] (Step 3). didfail [12] was introduced at about the same time as iccta. It also uses flowdroid for taint analysis and epicc for ICC and IAC detection. However, didfail has been created specifically to detect connections among a set of apps, so instead of combining apps in a pairwise fashion, intra-app analysis produces a set of outputs for each application that include a manifest file, epicc output, and flowdroid output. It then analyzes these output files to uncover IACs. Again, these approaches perform their initial analysis within the boundary of an app.

Our work differs from current IAC analysis approaches several ways. First, while our approach also performs analysis of each app in a collection, we perform these *simultaneously*. A typically approach needs to perform intra-app analysis *n* times for a collection consisting of *n* apps. Our approach performs its analysis just once, on all *n* apps. Second, by doing this, we preserve analysis graphs generated during the process so that richer analysis techniques such as points-to analysis can be conducted on our analysis graphs. As such, our approach is more efficient in the context of analyzing interacting apps than current approaches such as iccta that require the previously mentioned three steps. This is because: 1) Our intra app analysis is performed in parallel instead of sequentially; 2) data propagation can be done across apps naturally instead of needing to create some forms of outputs that must be composed for IAC analysis; 3) our approach eliminates the analysis graph reconstruction process altogether. As shown in this article, this last step in the current state-of-the-art approach to combine apps is inefficient and non-scalable.

SEALANT combines static analysis with runtime monitoring to prevent attacks through vulnerable IAC channels. Static analysis is used to identify vulnerable channels. Once identified, runtime monitoring is used to monitor activities in those channels to detect and prevent attacks [13]. In this work, static analysis is performed using IC3 [20] to identify IAC and COVERT [3] to perform compositional analysis to detect ICC vulnerabilities. The authors report that analyzing an app requires approximately 79 s. The majority of time (78 s) is used for architectural extraction. This information can be reused if the same app is analyzed again. In the case of at-speed analysis, an analyst needs to dynamically consider apps in a device so it is likely that architectural extraction is needed for each app in a device and the information may not be reused. Our work, on the other hand, can detect IAC channels in a device with 129 channels in about 20 min. This includes the time to pull apps from the device.

In terms of code coverage, EMMA is a widely used code coverage tool for Java and it now works for Android [29]. It supports both targeted unit testing and random testing using Monkey. The biggest difference between JITANA and EMMA is that EMMA requires instrumentation of the application. For example, to use EMMA with random testing (e.g., using Monkey to generate inputs) we first need to create an instrumenting module to instrument the app. By working directly with the VM, JITANA can observe runtime events without instrumentation.

7. Conclusion

We have presented JITANA, a hybrid program analysis framework for Android that has been designed to overcome limitations in existing program analysis approaches. We have shown that JITANA facilitates the analysis of Android apps for inter-app communications, an analysis that can help engineers and security analysts diagnose and address faults and vulnerabilities related to inter-app interactions. The results also show that JITANA is more efficient than a state-of-the-art approach for analyzing inter-app connections, and far more robust and scalable than that approach. We also have shown that JITANA is very efficient for performing hybrid analysis; it can generate real-time code coverage information that can be helpful for assessing dynamic analysis quality. JITANA can also be used to perform continuous analysis when an application dynamically loads code. In this situation, static analysis alone is inadequate.

There are many activities that we plan to conduct to improve JITANA's performance and capabilities. First, we are working on implementing an on-line IDE analysis framework so that we can improve the precision of our IAC analysis as well as provide a foundation by which other researchers can develop analysis approaches. Second, because JITANA generates all analysis graphs in BGL compliance form, we plan to develop approaches to persist prior analysis results and offload analysis tasks to high-performance clusters. Third, currently, JITANA does not analyze native code; therefore, one possible extension is to create an approach to convert ARM binary code to BGL compliance graphs. Such graphs can then be appended to JITANA graphs so that analysis can flow from managed domains to native domains and vice versa. Fourth, we are working on extending our Analysis Controller to be able to retrieve objects in addtion to classes. Our goal is to be able to retrive GUI objects that can help with creating more complete GUI models that can support more effective event sequence generation. Fifth, we are using JITANA to detect a new breed of malware that hides malicious behaviors through dynamic code loading. This paper provides a glimpse into what JITANA can do in this regard. Our future work will extend JITANA to become a complete malware analysis tool that can detect attack surfaces based on this type of threats. These are just a few ideas that we have for JITANA. JITANA is publicly available for download at: http://cse.unl.edu/~ytsutano/jitana/

Acknowledgments — This material is based on research sponsored by DARPA and Maryland Procurement Office under agreement numbers FA8750-14-2- 0053 and H98230-14-C-0140, respectively. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies or the U.S. Government.

Supplementary material is attached to the archive record for this article.

References

[1] S. Acharya, Google Removes 13 Android Apps from Play Store Infected with Brain Test Malware, 2016, http://www.ibtimes.co.uk/google-removes-13-android-appsplay-store-infected-brain-test-malware-1537049

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, (2014), pp. 259–269.

[3] H. Bagheri, A. Sadeghi, J. Garcia, S. Malek, COVERT: compositional analysis of android inter-app permission leakage, IEEE Trans. Softw. Eng. 41 (9) (2015) 866–886.

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

[5] Boost.org, Boost C++ library, 2018, http://www.boost.org/doc/libs/develop/libs/graph/doc/

[6] J. Dean, D. Grove, C. Chambers, Optimization of object-oriented programs using static class hierarchy analysis, Proceedings of the 9th European Conference on Object-Oriented Programming, Aarhus, Denmark, (1995), pp. 77–101.

[7] Google, Android security: 2016 year in review, 2017, https://source.android.com/security/reports/Google_Android_Security_2016_Report_Final.pdf

[8] M. Gouline, Code coverage on android with JaCoCo, 2015, https://blog.gouline.net/code-coverage-on-android-with-jacoco-92ec90c9355e

[9] Graphviz, Graphviz - graph visualization software, 2018, http://graphviz.org

[10] U. Ismail, Incremental Call Graph Construction for the Eclipse IDE, Technical Report, University of Waterloo, 2009.

[11] Kaspersky Lab, The most sophisticated android trojan, 2013, http://www.securelist.com/en/blog/8106/The_most_sophisticated_Android_Trojan

[12] W. Klieber, L. Flynn, A. Bhosale, L. Jia, L. Bauer, Android taint flow analysis for app sets, Proceedings of the ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis, (2014), pp. 1–6.

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

[14] O. Lhoták, L. Hendren, Scaling Java points-to analysis using SPARK, Proceedings of the 12th International Conference on Compiler Construction, (2003), pp. 153–169.

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

[16] L. Li, A. Bartel, T.F. Bissyandé, J. Klein, Y.L. Traon, Apkcombiner: combining multiple android apps to support inter-app analysis, Proceedings of the IFIP TC 11 International Conference, (2015), pp. 513–527.

[17] L. Li, A. Bartel, J. Klein, Y.L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, P. McDaniel, I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis, CoRR abs/1404.7431 (2014).

[18] T. Lindholm, F. Yellin, Java Virtual Machine Specification, 2nd, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[19] E. Messmer, Black Hat demo: Google Bouncer can be beaten, 2012, http://www. networkworld.com/news/2012/072312-black-hat-google-bouncer-261048.html.

[20] D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein, Y. Le Traon, Combining static analysis with probabilistic models to enable market-scale android inter-component analysis, Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (2016), pp. 469–484.

[21] D. Octeau, D. Luchaup, M. Dering, J. Somesh, P. McDaniel, Composite constant propagation: application to Android inter-component communication analysis, Proceedings of the 2015 International Conference on Software Engineering, (2015), pp. 77–88.

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

[23] Oracle Corp., Oracle Solaris Studio 12.4: performance analyzer, 2015, https://docs.oracle.com/cd/E37069_01/html/E37079/gkcct.html

[24] Oracle Corp., Java Debug Wire Protocol (JDWP), 2018, https://docs.oracle.com/javase/7/docs/technotes/guides/jpda/jdwp-spec.html

[25] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, G. Vigna, Execute this! analyzing unsafe and malicious dynamic code loading in android applications, 21st Annual Network and Distributed System Security Symposium, NDSS, San Diego, California, USA, (2014), pp. 1–16.

[26] P. Ponomariov, Shedun: adware/malware family threatening your Android device, 2015, https://blog.avira.com/shedun/

[27] S. Rasthofer, S. Arzt, E. Bodden, A machine-learning approach for classifying and categorizing android sources and sinks, 21st Annual Network and Distributed System Security Symposium, NDSS, San Diego, California, USA, (2014), pp. 1–15.

[28] T. Reps, S. Horwitz, M. Sagiv, Precise interprocedural dataflow analysis via graph reachability, Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming language, POPL ’95, New York, NY, USA, (1995), pp. 49–61.

[29] V. Roubtsov, EMMA: a free java code coverage tool, 2006, http://emma.sourceforge.net

[30] F. Ruiz, ‘fakeinstaller’ leads the attack on android phones, 2012, https://securingtomorrow.mcafee.com/mcafee-labs/fakeinstaller-leads-the-attack-on-android-phones/

[31] M. Schwartz, Google Play exploits bypass malware checks, 2012, http://www.informationweek.com/security/application-security/google-play-exploits-bypass-malware-chec/240001691

[32] J. Security, Deep analysis of android ransom charger, 2017, https://joesecurity.org/blog/4211295284540909643

[33] The Next Web, Android users have an average of 95 apps installed on their phones, according to Yahoo Aviate data, 2014, http://thenextweb.com/apps/2014/08/26/android-users-average-95-apps-installed-phones-according-yahoo-aviate-data/

[34] E. Tinaztepe, D. Kurt, A. Gulec, Android obad, 2013, (COMODO, Tech. Rep.).

[35] Y. Tsutano, S. Bachala, W. Srisa-an, G. Rothermel, J. Dinh, An efficient, robust, and scalable approach for analyzing interacting android apps, Proceedings of the International Conference on Software Engineering, Buenos Aires, Argentina, (2017), pp. 324–334.

[36] R. Vallée-Rai. Soot: A java bytecode optimization framework (Master’s thesis), McGill University, 2000.

[37] A. von Rhein, T. Berger, N.S. Johansson, M.M. Hardœ, S. Apel, Lifting Inter-App Data-Flow Analysis to Large App Sets, Technical Report, University of Passau, 2015.

[38] F.P. Wei, S. Roy, X. Ou, R. Song, Amandroid: a precise and general inter-component data flow analysis framework for security vetting of Android apps, Proceedings of the ACM Conference on Computer and Communications Security, (2014), pp. 1329–1341.

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