2017 IEEE/ACM 39th International Conference on Software Engineering

An Efficient, Robust, and Scalable Approach for Analyzing Interacting Android Apps

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
Email: {ytsutano, sbachala, witty, grother, jdinh}@cse.unl.edu

Abstract—**When multiple apps on an Android platform interact, faults and security vulnerabilities can occur. Software engineers need to be able to analyze interacting apps to detect such problems. Current approaches for performing such analyses, however, do not scale to the numbers of apps that may need to be considered, and thus, are impractical for application to real-world scenarios. In this paper, we introduce JITANA, a program analysis framework designed to analyze multiple Android apps simultaneously. By using a classloader-based approach instead of a compiler-based approach such as SOOT, JITANA is able to simultaneously analyze large numbers of interacting apps, perform on-demand analysis of large libraries, and effectively analyze dynamically generated code. Empirical studies of JITANA show that it is substantially more efficient than a state-of-the-art approach, and that it can effectively and efficiently analyze complex apps including Facebook, Pokémon Go, and Pandora that the state-of-the-art approach cannot handle.**

I. INTRODUCTION

Smart-mobile application platforms such as Android provide software engineers with IDEs that help them develop GUIs and generate skeleton code, and a powerful application framework that they can use to quickly create Java applications (apps) that run on Android devices. In this way, Google, as the Android provider, can collaborate with external developers to build apps with different functionalities, which in turn increases the value of Android. For such reasons, Android is currently the most widely adopted smart-mobile platform in the world.

To ensure the dependability and security of Android apps, engineers and security analysts need to be able to isolate faults and security vulnerabilities in these apps. While such faults and vulnerabilities can arise from various sources, many occur due to interactions between apps, and these can come to exist in various ways. For example, in collusion attacks, multiple apps are deliberately altered to work together to perform malicious acts [1], [2]. A recent study by McAfee Labs reports that over 5,000 versions of 21 Android apps contained colluding code capable of data exfiltration, file inspection, fake SMS messages, and other malicious activity [3]. A recent study by Memon and Anwar lists collusion as the next major mobile malware threat [4]. Our proposed framework, JITANA, is designed to effectively and efficiently address this problem.

As another example, Android apps can interact by exchanging messages [5], [6], [7], and when any of the apps involved in interactions is updated, failures can occur. (A recent study notes that 23% of Android apps behave differently after a platform update [8].) These types of faults and security vulnerabilities create the need for program analysis frameworks that are capable of analyzing multiple apps so that distributed interactions can be discovered. Unfortunately, most widely used program analysis tools for Android operate primarily on individual apps, so they are not directly capable of detecting faults and vulnerabilities that involve interactions among multiple apps. Nonetheless, researchers, engineers, and analysts attempting to address such issues have created functional techniques based on these existing tools [9], [10], [11], [12], [13].

In one technique used to analyze multiple interacting apps, Li et al. [14] employ EPICC [15], APKCOMBINER [14], and FlowDroid [16] in a four-step process. First, two candidate apps are selected. Second, APKCOMBINER is used to merge the two selected apps into a single app. The goal of this step is to reduce the IAC problem to an inter-component communication (ICC) problem that existing ICC analysis tools can solve. Third, the merged app is analyzed using EPICC, an ICC analysis tool that was state-of-the-art at that time. EPICC can be used in this context because all IAC connections have been converted to ICC connections in the merged app. Finally, FlowDroid is used to perform taint analysis on the merged app to identify possible malicious ICC connections.

While approaches such as the foregoing can operate on small-scale scenarios, they are often impractical for real-world use [17]. Section II provides an example that demonstrates that the foregoing (state-of-the-art) technique is unable to analyze a collection of apps with IACs when their interactions become just modestly complex. This is because the merging process (Step 2) is inefficient, fragile, and non-scalable. As such, the technique has no chance of scaling up to analyze even a handful of real-world apps for faults and vulnerabilities.

The goal of this work is to overcome the sources of inefficiencies in prior approaches by rethinking the design of program analysis frameworks in general. Specifically, instead of designing a framework that analyzes one app at a time, we design a framework that can efficiently and simultaneously analyze multiple apps while maintaining all of the information needed to support rich analysis techniques.

In this paper, we introduce JITANA (derived from "JIT analysis"), a program analysis framework for Android that is capable of analyzing multiple apps simultaneously with native support for the DEX instruction set. Unlike SOOT [18], a commonly used program analysis framework for Java and Android, JITANA is not designed like a compiler framework. Instead, it operates in a fashion similar to that of a *classloader* used in any Java or Android virtual machine. Engineers can load parts of an app, an entire app, or multiple apps for static analysis without decomposing or merging apps. JITANA also supports analysis of shared system classes and third-party library classes. Since JITANA can analyze all loaded apps simultaneously, it can generate important *program analysis graphs* such as data-flow graphs (DFGs) and interprocedural control-flow graphs (ICFGs), that span multiple apps.

The static class loading process JITANA uses has further advantages. First, it naturally resolves potential naming conflicts (e.g., two classes having the same name) in the same way in which a real classloader would. Second, it continues to maintain clear separations between apps, so it preserves *runtime properties* that might otherwise be lost in the merge process (e.g., two apps in Android can be two separate processes so their address spaces are not implicitly shared).

Because JITANA does not modify apps through instrumentation or merging, all apps continue to run in their usual fashion. In contrast, an app generated by merging multiple apps using APKCOMBINER is not guaranteed to run [14], so testing it for correctness may not be possible. Further, when many apps are merged into one, the resulting app can be large, rendering the use of precise analysis techniques more difficult.

To evaluate JITANA, we use three experimental scenarios to address three Research Questions (RQs).

- RQ1: How efficient is JITANA?
- RQ2: How robust is JITANA?
- RQ3: How scalable is JITANA?

In our evaluations, we compare JITANA with the approach (described above) that merges multiple apps. Our evaluations show that JITANA is substantially more efficient than the merging-based approach even in relatively simple analysis scenarios, and far more robust in analyzing non-trivial groups of real-world applications. JITANA also scales to effectively analyze non-trivial real devices using different numbers of installed apps – cases that the merging-based approach cannot handle.

In the next section, we provide a motivating example. We then introduce JITANA in Section III and evaluate it in Section IV. Section V provides additional insights into our empirical results. Section VI highlights prior research efforts related to our work, and Section VII concludes.

II. MOTIVATION

As noted in Section I, 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, IC3 [19] or SOOT. Li et al. [14] 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 [20] 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. [21] first generate models of components and then combine them to perform analysis; this approach has been used to detect permission leakage [21]. 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 [14] we described in Section I. 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 [15] or general program analysis frameworks such as SOOT [18]. 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 since it is less likely for a user to have all 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 [22]. 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 it 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.

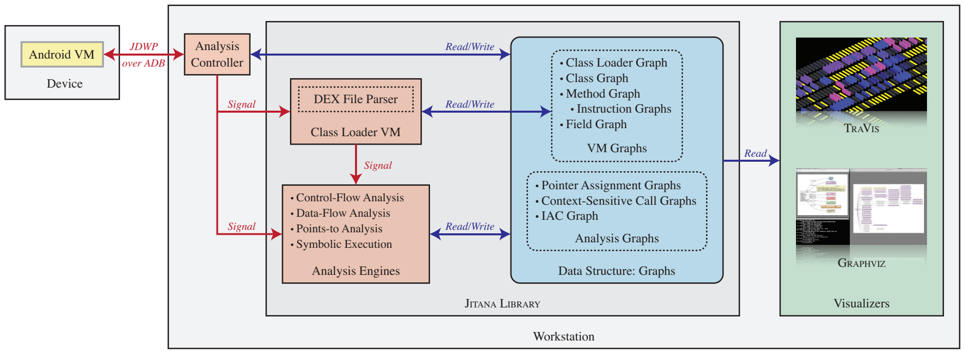


Fig. 1: Architecture of JITANA

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 needs to be merged, Li et al. [14] introduced their hybrid approach. While this approach can analyze a small number of connected apps, we show in Section IV that it faces scalability and robustness issues when it is used to merge a large number of apps or complex apps.

III. JITANA: A FRAMEWORK FOR ANALYZING INTERACTING ANDROID APPS

To address the problems just noted, and to provide a new program analysis framework for Android that can efficiently and scalably analyze multiple apps at the same time while preserving analysis graphs, we now introduce JITANA,[1] a framework for supporting both static and dynamic program analysis techniques. In this work, our main focus is static program analysis but during our subsequent discussion of analysis results we also show how JITANA can perform a dynamic analysis to deal with dynamically generated code. This renders static analysis more effective, because code generated at runtime can also be included in the analysis.

To render our static program analysis more efficient, we employ a classloader-based analysis approach instead of the traditional compiler-based analysis approach used by tools such as SOOT, which focus their analysis effort within the boundary of an application [23]. SOOT first loads the entire code of an app and performs analyses that include 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. [14] 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. Moreover, compiler-based approaches often face difficulties handling large libraries because entire codebases must be analyzed. For example, AMANDROID, a framework introduced by Wei et al. [24], 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.

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

Figure 1 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* or 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*). 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 (see Section V for a use-case of this feature that we encountered in this study).

The next major component in the framework is the *Class-Loader VM* (*CLVM*). We implemented *CLVM* based on the Java Virtual Machine Specification [25]. 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 statically load code based on the same algorithm. By using the *CLVM* to load classes as part of our static analysis framework, the following benefits ensue:

1) The explicit relationship between a classloader and classes that it loads creates a clear encapsulation of all classes in an application. This property enables us to load multiple apps at the same time while maintaining clear separation among apps. 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.
2) Because our CLVM operates in the same way as an actual classloader, the dynamic analysis component of JITANA can handle dynamically generated code. For

Algorithm 1: Class Loading 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      $C \longleftarrow \mathbf{null}$;
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 $L \neq \mathbf{null}$;
11     return $\langle L, C \rangle$;
```

example, after installation on a device, we discovered that *Facebook* loads multiple additional DEX files dynamically. Our framework simply pulls these DEX files from the device directly for analysis.

3) The use of a CLVM also provides an efficient way to analyze large libraries. Traditional compiler-based 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, by default, analyzes classes from third party libraries as well as system classes.

Once classes are loaded, the system generates a set of VM graphs that include *classloader graphs*, *class graphs*, *method graphs*, *instruction graphs*, and *field 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 EPICC-based IAC graph. The framework can run side-by-side with existing visualization tools such as GRAPHVIZ, an open-source graph visualization tool. Next we describe these graphs in details.

A. Supported Graphs

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

To achieve *efficiency*, every graph used in JITANA models appropriate graph concepts defined in the Boost Graph Library (BGL),[2] a de facto generic C++ graph library. This means that implementations of highly optimized generic graph algorithms

TABLE I: 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

are already available for use by applications on the graphs defined in JITANA without modifications. We also implement JITANA in C++, which supports the use of the generic programming paradigm with templates. This paradigm separates algorithms and data structures by defining what is called a *concept*, a description of both syntactic and semantic requirements for one or more types [26]. An algorithm operating on a concept needs to be implemented only once; the same implementation can then be reused for any concrete type that is a model of the concept. Generated graphs can be persisted for future reuse. Analyses can also be performed on any machines on which the BGL library is installed. The C++ template instantiation mechanism coupled with compiler optimizations has been shown to generate code that is as efficient as hand-tuned FORTRAN [27].

Unlike JITANA, most existing tools do not use explicit graph types for data structures. For example, SOOT and LLVM [28] follow the traditional object-oriented approach: an object holds pointers to other objects to imply relationships that implicitly form a graph data structure. This means that algorithms implemented in these platforms are strongly tied to a tool's design details, not to a graph concept. As a consequence, even a simple depth-first search algorithm, for example, must be implemented each time it is needed, and when a new tool arrives, a library of algorithms must be rewritten.

Table I 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[3] so that analysis engines can add edges of any type. Examples of some of these graphs rendered with GRAPHVIZ are shown in Figure 2.

Figure 2(a) displays a *classloader graph* for a case in which JITANA analyzes four apps simultaneously. Each class loader is assigned a unique ID (integers in the upper left corners) so

[2] http://www.boost.org/doc/libs/develop/libs/graph/doc/

[3] http://www.boost.org/doc/libs/develop/doc/html/boost_typeerasure.html

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;[4] therefore, the names of DEX files have no meaning. *ClassLoader 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).

Figure 2(b) displays a *class graph* that 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;`).

Figure 2(c) displays a *method graph* that shows relationships among several methods within a set of analyzed apps. Nodes represent methods, and edges indicate whether method calls are *direct* or *virtual*. The numbers in the upper left corners of the nodes indicate the apps to which the methods belong.

Not depicted in the figure are instruction graphs and field graphs. JITANA produces an *instruction graph* for each method that includes both control-flow and data-flow information. The data-flow information is derived via reachability analysis performed on virtual registers. *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.

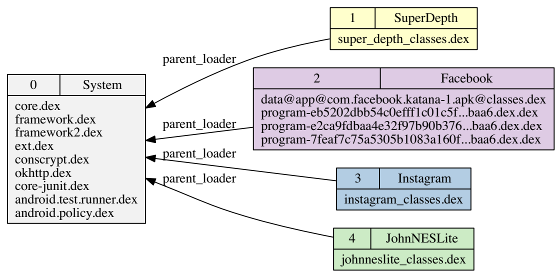
B. Supported Analysis Engines

Currently, JITANA supports traditional interprocedural control-flow, data-flow, and points-to analysis graphs [18]. In Java, most function calls are made 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 [29], a points-to analysis framework in SOOT [18].

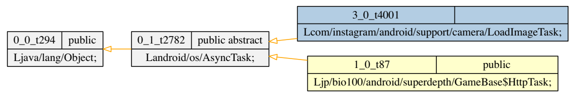
JITANA's *interprocedural control-flow analysis* includes both direct call edges and virtual call edges in a method graph as shown in Figure 2(c). However, the actual target of a virtual call edge cannot be accurately computed without consulting the *virtual dispatch tables* (*vtables*) that are used to support late-binding features such as inheritance. As such, our analysis is not sound because it ignores the vtables. Further, our analysis is incomplete because it ignores reflection. These two issues are common among static analysis tools for Java.

In terms of *data-flow analysis*, JITANA supports reaching definitions analysis, which is used to generate def-use pairs. The monotone data-flow 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.

[4]https://www.facebook.com/notes/facebook-engineering/under-the-hood-dalvik-patch-for-facebook-for-android/10151345597798920

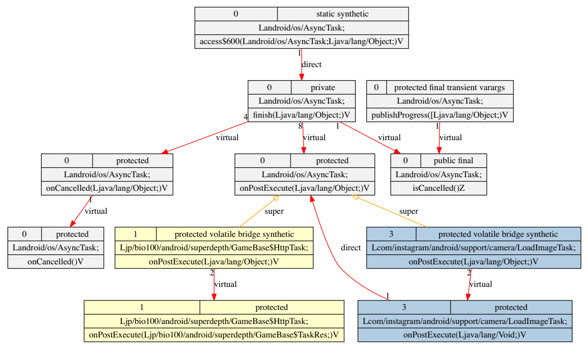


(a) Classloader graph.



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

(b) Class graph (subgraph).



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
1
2
virtual
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

(c) Method graph (subgraph).

Fig. 2: Illustrations of Various VM Graphs

The *IAC analysis* used by JITANA is similar to the approach used by EPICC to initially uncover possible ICC connections. In the first step, it searches for potential connections (exit and entry points) between components and apps; this is done by analyzing the code and manifest files. Because JITANA analyzes a collection of apps 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, 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. 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. Currently, JITANA does not support interprocedural dataflow analysis. However, once we implement it in JITANA, dataflow analysis can be performed across these IAC edges to increase analysis precision.

IV. Empirical Evaluation

As noted in Section I, to evaluate JITANA we empirically studied its efficiency, its robustness, and its scalability. We now present each of these studies in turn. We conducted all three of our studies on a computer with 2.4 GHz Core 2 Duo and 16 GB DDR3 RAM running Apple OS X (El Capitan).

A. RQ1: How Efficient is JITANA?

1) Study Setup: To address this research question we compare the performance of JITANA with the approach of Li et al. [14], described in Section II. As a reminder, their approach randomly selects two apps and uses APKCOMBINER to merge them. The approach then utilizes SOOT to produce analysis graphs (e.g., ICFG, DFG) so that more powerful analysis approaches such as ICC analysis (using EPICC) or taint analysis (using FLOWDROID) can be applied.

Work by Li et al. reported that APKCOMBINER was able to successfully combine over 2,600 pairs of real Android apps [14]. We initially considered using these apps for RQ1. However, the authors provided no information about these apps (except for *Edabah Evaluation* and *ClipStore*) so it was not possible 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, we were unable to use them in our study. Next, we collected the most recent top 10 apps (August 2016) from three different categories in Google Play as possible experiment objects (Table IV provides a complete list). We then attempted to use APKCOMBINER to merge these 30 apps. Because APKCOMBINER merges two apps at a time, we simply tried to find pairs of apps in each category that it could merge. Unfortunately, APKCOMBINER was not able to merge any pair of apps in any of our three categories. 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.

Ultimately, we resorted to using 26 apps that we discovered that APKCOMBINER can operate on. Three of these are part of the *DroidBench* benchmark suite and had previously been used to evaluate APKCOMBINER [14]; we created the other 23 apps ourselves, with the goal of *explicitly representing different types of IACs*. The apps we used are small so we can manually identify IAC connections that exist among them. We also know each connection type (e.g., explicit intent, implicit intent, broadcast receiver, and content provider). In all, there are 14 pairs of connections among the 26 apps. We then verified the IAC information by using EPICC and IC3. We refer to this hybrid approach as *Oracle + APKCombiner + Soot* or OAS. In addition to representing various connection types, these small apps have the further advantage of being able to be processed by both OAS and JITANA, allowing us to compare the two for efficiency.[5]

[5]These benchmark apps and additional information about them can be downloaded from: http://cse.unl.edu/~ytsutano/jitana/.

We conducted this study using three scenarios. In Study Scenario 1, we used APKCOMBINER to merge each of the 14 pairs of connected apps and then used SOOT to analyze each pair of merged apps to construct the basic program analysis graphs that include CFGs, DFGs, and ICFGs. We also used JITANA to analyze each pair of apps simultaneously.

In Study Scenario 2, we increased the number of connected apps to three. In this case, we used only the *DroidBench* apps, which are already connected in two ways. We applied APKCOMBINER, in turn, to create two pairs of interconnected apps (i.e., a_{01} and then a_{12}; note that a_0 and a_2 are not connected), while we applied JITANA to the set of three apps simultaneously.

In practice, a system under analysis for vulnerabilities might host various combinations of apps, and analysts might need to consider all of these combinations. Study Scenario 3 reproduces this situation. We drew seven independent random samples of apps from our benchmark set, ensuring that each sample had a (randomly chosen) number of apps ranging from four to 18. We also ensured that each resulting group of apps contained at least four apps that had IAC connections. We also attempted to combine and analyze all 26 apps.

Because OAS cannot directly analyze an entire group of apps, and instead analyzes pairs of apps, we again used the information we possessed to determine the IAC connections among apps in that group. We then selected pairs of connected apps for merging by APKCOMBINER. For example, one group of six apps included four that are connected as two pairs; we used the APKCOMBINER to merge these two pairs. Then, we used SOOT to analyze the two merged apps, generating CFGs, DFGs, and ICFGs. To assess the efficiency of the approach, we measured the time required to merge the pairs of apps and then generate analysis graphs for the merged pairs.

Currently, IAC analysis in JITANA identifies connections through explicit intents, implicit intents, and broadcast receivers. Basically, JITANA identifies the potential IAC requests for each app, and analyzes intent filters to identify IAC entry points for each app. We also implemented an analysis to identify IAC connections through a content provider. Given the foregoing, we also evaluated two performance aspects of JITANA in Scenario 3. First, we wished to observe the costs of the two approaches subsequent to the generation of IAC connections. This requires the two approaches to operate on the same sets of connections. Thus, for JITANA, we begin with the same groupings and IAC information that had been determined for use with OAS. In this case, however, JITANA then simultaneously analyzes all the apps in a group instead of just the apps identified by the oracle as connected. This results in the generation of the same IAC graph, as provided, and the same analysis information as OAS.

Second, we also let JITANA compute and construct IAC graphs on its own. In this case, each reported time includes the additional time required to compute IAC connections and construct the IAC graph based on implicit and explicit intents and the time required to simultaneously analyze all the apps. To assess the efficiency of JITANA in this case, we measured

TABLE II: RQ1: Analysis Data for OAS and JITANA for Pairs of Apps P0–P13 (Scenario 1) and DroidBench (Scenario 2)

	Reported Time (sec)														
	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	DroidBench
OAS	66.38	66.04	55.23	55.73	72.15	65.29	13.10	13.13	69.44	71.34	56.47	55.43	67.94	66.61	111.90
JITANA	4.52	4.82	2.41	2.62	4.41	4.36	3.26	2.86	4.53	4.58	4.65	4.83	2.66	2.56	3.75

the time required to analyze all of the apps in each group, produce the IAC graph for each group, and simultaneously analyze apps. For Scenario 1, time does not include IAC analysis since the apps are known to be connected. For Scenarios 2 and 3, times are reported both with and without IAC analysis.

2) Results: Table II reports our results relative to the first two scenarios in which we investigated RQ1. The table lists the times (in seconds) required by OAS and JITANA to analyze pairs of apps in Scenario 1 (Columns "P0"–"P13") and the three connected apps from *DroidBench* in Scenario 2 (Column "DroidBench").

In the case of Scenario 1, in all cases, the times required by JITANA were substantially less than the times required by OAS. The performance speed-ups ranged from 4 times to 26 times with an average of 16.

In the case of Scenario 2, we used APKCOMBINER to merge pairs of apps. Each merged app is then analyzed by SOOT. The reported time is the sum of the analyses required over the pairs of apps analyzed. In this scenario, JITANA achieved a performance speed-up of 30 times.

Where Scenario 3 is concerned, Table III reports the times taken by both systems to analyze larger groups of apps. Each row presents data on a particular configuration of apps used to evaluate OAS and JITANA. The first column in the table reports the number of apps in a configuration. The second column reports the number of pairs of connected apps in the configuration; this indicates the number of pairs of apps that OAS needs to combine. The third column reports the time required by OAS to combine the pairs of apps and for SOOT to analyze each resulting merged apps. The fourth column reports the time required by JITANA to analyze all the apps in each configuration. The fifth column includes the time required by JITANA to analyze all the apps in each configuration and compute the IACs among them.

TABLE III: RQ1: Analysis Data for OAS and JITANA (Scenario 3)

# of Apps (MB)	# of Connected Apps (MB)	OAS (sec)	Jitana (sec)	Jitana + IAC (sec)
4 (1.9)	4 (1.9)	79.26	7.82	7.95
5 (1.8)	5 (1.8)	151.40	6.46	6.47
6 (2.7)	4 (1.9)	93.11	9.30	9.47
9 (3.9)	6 (2.9)	203.46	11.83	11.92
12 (4.7)	10 (3.9)	216.57	15.80	15.99
15 (6.3)	14 (4.8)	413.08	17.53	17.54
18 (7.3)	16 (6.3)	468.32	21.72	21.90
26 (10.1)	26 (10.1)	794.27	33.48	33.90

As the data shows, with or without IAC, JITANA achieved performance speed-ups ranging from 10 times to 24 times. When we used JITANA to perform IAC analysis, it incurred negligible overhead.

B. RQ2: How Robust is JITANA?

1) Study Setup: To answer this research question we used real-world applications available for free download from the Google Play Store (the same set we had initially attempted to use to answer our first research question, as noted in Section IV-A). Our selected apps were all on the top 100 chart during August 2016. 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. First, we chose 10 apps from the "social network" category including *Facebook*, *Snapchat*, and *Instagram*, 10 apps from the "games" category including *Pokémon Go*, *Suicide Squad*, and *NBA Live*. The last 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 IV lists all of the apps in each category.

TABLE IV: Apps Downloaded from Google Play for RQ2

Category	Listing of Apps
Social Network	Facebook, Snapchat, Instagram, Pinterest, Twitter ooVoo, TextNow, 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

As previously mentioned (Section IV-A), APKCOMBINER was unable to combine any of these real-world apps in a pair-wise fashion. As such, we could not use it to produce any merged apps for ICC analysis. Next, we turned our focus to EPICC and IC3 and attempted to use them to search for entry/exit points in each of these large apps so that we can compare the results produced by these systems with those produced by JITANA. Unfortunately, both EPICC and IC3 were unable to return results for these large apps in the three categories. We also encountered a few instances in which both systems simply crashed. One possible reason for their failures involves failures during code retargeting, which is a problem that has been previously reported [15]. 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) [15], [19], [30], 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 processed.

We next used JITANA to simultaneously analyze all ten apps in each category to produce analysis graphs (the same contexts as those used in RQ1), compute IAC connections, and generate the IAC graph for each category. We measured the time required to complete this process. Unlike in RQ1, we did not provide IAC information prior to analysis.

2) Results: Table V reports results relative to RQ2. 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. For example, the first row of the social network category includes two apps: *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.

TABLE V: RQ2: 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

As the data show, JITANA successfully analyzed between two and ten apps in each of the three categories. It also processed large amounts of code, ranging from 22 MB to over 450 MB. The number of classes loaded and analyzed by JITANA exceeded 50,000 classes in several cases. The analysis times range from 27 seconds to 176 seconds. Note that these times also include the time required to compute IACs.

C. RQ3: How Scalable is JITANA?

1) Study Setup: To answer this research question 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 analyzes all of the apps on each device to build analysis contexts, compute the IAC connections among the apps, and generate the IAC graph.

In conducting this study, we are considering 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 may need to vet employee's devices to detect whether there are any connections that may allow existing apps 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 "at speed" as employees check in their devices.

We configured our three devices to have 99 apps (only 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 those on the first and second devices, respectively. These additional apps include the 30 apps used to investigate RQ2 (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 baseline approach.

2) Results: Table VI reports our results relative to RQ3. Columns 2–4 provide data on the device configuration 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.

TABLE VI: RQ3: Analysis Data for JITANA, Using Three Devices with Different Numbers of Apps

ID	# of Apps	Total Size (MB)	# of Classes ($\times$1K)	# of Methods ($\times$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

As the data shows, JITANA was able to analyze a device with 129 apps in about 19 minutes (Device 3). 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 minutes and 11 minutes, respectively. For Device 3, JITANA loads and analyzes over 320,000 classes, which amounts to about 1.35 GB of code. It is also worth noting that each app on a device has a tendency to connect with another 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.

D. Threats to Validity

The primary threat to external validity in this study involves the object programs utilized. In our first study, we have studied small benchmark programs to provide an equal chance for both OAS and JITANA to run successfully so that we can compare the efficiencies of the two systems; however, the inability of OAS to handle larger cases mandated this. We also employ complex real-world apps including *Facebook*, *Instagram*, *Twitter*, *Pandora*, and *Pokémon Go*. This is to apply our system to real-world scenarios, representative of those that engineers and analysts are facing.

A second threat involves the representativeness of our techniques for identifying IAC connections. In the case of our small benchmark apps, we were able to identify all the IAC connections that exist among the apps. For larger apps, we attempted to use the state of the art approach, IC3, to first 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 OAS needs to combine. 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.

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

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

V. DISCUSSION

One interesting observation from Table III is that the amount of time needed 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 a sophisticated interprocedural dataflow analysis technique [31] 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 VI 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 working to extend JITANA to also support interprocedural dataflow analysis [16] and more complex constant propagation [19] 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. When we analyzed five apps and 12 apps simultaneously, JITANA mistakenly identified a pair of apps in each case as having an IAC connection. For the time being, if higher precision can be sacrificed for higher efficiency and scalability, JITANA can already be used to determine IAC connections.

In RQ2 and RQ3, we also observed an interesting strategy that apps such as *Facebook* employ. To reduce the size of the downloaded APK, *Facebook* generates additional classes at runtime. These classes appear in multiple additional DEX files. To capture them, we utilized the dynamic analysis capability of JITANA to monitor classes being loaded. In all, JITANA ended up analyzing 23,621 classes while SOOT analyzed only 6,350 classes in *Facebook*. The times required to analyze *Facebook* for SOOT and JITANA were 181.35 and 22.52 seconds, respectively. This motivates the need for an efficient hybrid program analysis system to capture these additional classes, because static analysis alone, including JITANA in static analysis mode only, would have missed them.

Finally, as first noted in Section IV-C2, imagine a company that allows its employees to bring their own devices to work. It is conceivable that there is a team of analysts that 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 [32] indicates that a smart-mobile device, on average, contains about 95 apps. Based on the performance reported in Table VI, 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.

VI. RELATED WORK

In this section, we discuss other research efforts that are related to this work. These efforts include program analysis frameworks for Android and detection of IAC connections.

Over the past few years, there have been several approaches introduced to analyze systems for ICCs [2], [15], [19], [24].

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 a real-world apps range from 38 seconds for apps with less than 400 to 144 seconds for apps with 1,500 classes or more. Our approach, however, can analyze an app with approximately 400 classes in less than two seconds and an app with approximately 200 classes in about 11 seconds.

IC3 [19] 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 [24] 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) of the app 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 [2]. In terms of performance, IC3 can analyze a corpus of 460 real-world Android apps (downloaded in 2013) for ICC connections in about 18 hours – an average of 140 seconds per app. Because our framework does not yet support MVC we cannot compare our performance with IC3; however, 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. [30] 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. Octeau et al. conducted an evaluation using a collection of 11,267 apps and malware samples [33]. They did not report the time required by IC3 to initially analyze these apps, but the intent resolution time was 43 minutes. Note that the intent resolution process is orthogonal to 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 [2]. 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 [16] can perform taint analysis on the resulting app [14] (Step 3). DIDFAIL [22] 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 the current IAC analysis approaches in the following 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 done in parallel instead of sequentially; 2) data propagation can be done across apps naturally instead of having 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 paper, this last step in the current state-of-the-art approach to combine apps is inefficient and non-scalable.

VII. CONCLUSION

We have presented JITANA, a program analysis framework for Android that is capable of analyzing multiple apps simultaneously. We show that JITANA facilitates the analysis of Android apps for inter-app communications, an analysis that can help engineers and security analysis diagnose and address faults and vulnerabilities related to inter-app interactions. We 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.

There are many activities that we plan to conduct to improve JITANA's performance and capabilities. First, we are creating an interprocedural data-flow 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. Because JITANA generates all analysis graphs in BGL compliance forms, we plan to develop approaches to persist prior analysis results and off-load analysis tasks to high-performance clusters. Currently, JITANA does not analyze native code. 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 domain to native domain and vice versa. These are just a few ideas that we have for JITANA. JITANA is publicly available for download at: http://cse.unl.edu/~ytsutano/jitana/.

ACKNOWLEDGMENT

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.

References

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

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

[3] McAfee Labs, "McAfee Labs Report Reveals New Mobile App Collusion Threats," 2016, https://www.mcafee.com/us/about/news/2016/q2/20160614-01.aspx.

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

[5] J. Garcia, D. Popescu, G. Safi, W. G. J. Halfond, and N. Medvidovic, "Identifying message flow in distributed event-based systems," in *Proceedings of the ACM Symposium on Foundations of Software Engineering*, 2013, pp. 367–377.

[6] "Apple breaks new iphones with terrible software update," http://www.slate.com/blogs/future_tense/2014/09/24/apple_ios_8_0_1_software_update_major_bugs_hit_iphone_6_6_plus.html, 2014.

[7] "YouTube API change: some older devices can't update to new app," http://hexus.net/ce/news/audio-visual/82570-youtube-api-change-older-devices-update-new-app/, 2014.

[8] V.-V. Helppi, "What Every App Developer Should Know About Android," http://www.smashingmagazine.com/2014/10/02/what-every-app-developer-should-know-about-android/, October 2014.

[9] W. G. J. Halfond and A. Orso, "Improving test case generation for web applications using automated interface discovery," in *Proceedings of the ACM Symposium on The Foundations of Software Engineering*, 2007, pp. 145–154.

[10] K. R. Jayaram and P. Eugster, "Program analysis for event-based distributed systems," in *Proceedings of the ACM International Conference on Distributed Event-based System*, 2011, pp. 113–124.

[11] R. Purandare, J. Darsie, S. G. Elbaum, and M. B. Dwyer, "Extracting conditional component dependence for distributed robotic systems," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012, pp. 1533–1540.

[12] G. Safi, A. Shahbazian, W. G. J. Halfond, and N. Medvidovic, "Detecting event anomalies in event-based systems," in *Proceedings of the ACM Symposium on Foundations of Software Engineering*, 2015, pp. 25–37.

[13] W. G. J. Halfond and A. Orso, "Automated identification of parameter mismatches in web applications," in *Proceedings of the ACM Symposium on Foundations of Software Engineering*, 2008, pp. 181–191.

[14] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Apkcombiner: Combining multiple android apps to support inter-app analysis," in *Proceedings of the IFIP TC 11 International Conference*, 2015, pp. 513–527.

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

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

[17] P. Woollacott, "Threat Information Overload is Overwhelming Security Analysts," http://thevarguy.com/network-security-and-data-protection-software-solutions/051915/threat-information-overload-overwhelming-sec, May 2015.

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

[19] D. Octeau, D. Luchaup, M. Dering, J. Somesh, and P. McDaniel, "Composite constant propagation: Application to Android inter-component communication analysis," in *Proceedings of the 2015 International Conference on Software Engineering*, 2015, pp. 77–88.

[20] A. von Rhein, T. Berger, N. S. Johansson, M. M. Hard, and S. Apel, "Lifting Inter-App Data-Flow Analysis to Large App Sets," University of Passau, Technical Report MP-1504, September 2015.

[21] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "COVERT: compositional analysis of android inter-app permission leakage," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 866–886, 2015.

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

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

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

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

[26] A. Stepanov and P. McJones, *Elements of Programming*. Upper Saddle River, NJ: Addison-Wesley, 2009.

[27] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine, "Concepts: Linguistic support for generic programming in C++," in *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, October 2006, pp. 291–310.

[28] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004, pp. 75–.

[29] O. Lhoták and L. Hendren, "Scaling Java points-to analysis using SPARK," in *Proceedings of the 12th International Conference on Compiler Construction*, April 2003, pp. 153–169.

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

[31] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming language*, ser. POPL '95, New York, NY, USA, 1995, pp. 49–61.

[32] 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/.

[33] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2012, pp. 95–109.