ELEGANT: Towards Effective Location of Fragmentation-Induced Compatibility Issues for Android Apps

Conference Paper · December 2018

DOI: 10.1109/APSEC.2018.00042

Chang Xu
Nanjing University
127 PUBLICATIONS **1,659** CITATIONS

Lili Wei
The Hong Kong University of Science and Technology
14 PUBLICATIONS **200** CITATIONS

Jue Wang
Nanjing University
7 PUBLICATIONS **41** CITATIONS

Jun Ma
Nanjing University
18 PUBLICATIONS **170** CITATIONS

ELEGANT: Towards Effective Location of Fragmentation-Induced Compatibility Issues for Android Apps

Cong Li[†⋆], Chang Xu[†∗%], Lili Wei[‡⋄], Jue Wang[†⋆], Jun Ma[†∗], Jian Lü[†∗]
[†]*State Key Lab for Novel Software Tech. and Dept. of Comp. Sci. and Tech., Nanjing University, Nanjing, China*
[‡]*Dept. of Comp. Science and Engineering, The Hong Kong Univ. of Science and Technology, Hong Kong, China*
[⋆]{leetsong.lc, juewang591}@gmail.com, [∗]{changxu, majun, lj}@nju.edu.cn, [⋄]lweiae@cse.ust.hk

Abstract—**Android fragmentation is a double-edged sword of the Android ecosystem. On the one hand, it promotes Android's prevalence. On the other hand, the numerous combinations of various system versions, customized features, system drivers, and device models make it infeasible, if not impossible, for developers to exhaustively test their apps for potential compatibility issues. Previous research has proposed promising techniques for detecting these issues. However, they suffer from severe false positive problems due to their lack of third-party library detection or imprecise program analysis. In this paper, we present ELEGANT, an automated tool to effectively detect and locate fragmentation-induced compatibility issues for Android apps. ELEGANT exploits whitelist-enhanced or obfuscation-insensitive techniques to detect and alleviate the impact of third-party libraries on the analysis precision, and uses a three-step static detection algorithm to increase the precision of its program analysis. We experimentally evaluated ELEGANT with 22 real-world popular Android apps. The experimental results confirmed ELEGANT's effectiveness on detecting and locating Android fragmentation-induced compatibility issues, as well as realizing an impressive reduction on false positives by around 70%.**

Index Terms—**Android fragmentation, compatibility issues, testing**

I. Introduction

Android gains popularity rapidly in recent years. In Q1 of 2011, Android became the most popular mobile operating system with 36.4% market share [1]. Since then, more and more OEMs (short for original equipment manufacturers) have joined Android. Based on the open-source original Android system, deeply customized operating systems (e.g., Oxygen OS) and mobile devices (e.g., Samsung Galaxy) have been developed and produced. By Q1 of 2018, its market share has grown to 85.9% [1]. However, customized operating systems and mobile devices of Android also bring heavy fragmentation, which in turn brings severe compatibility issues (we follow previous research [2] and name such issues *fragmentation-induced compatibility issues*, or *FIC issues* for short in the rest of this paper), i.e., one app functions normally on some devices while malfunctions or even crashes on others. By far, there are more than 20 customized Android operating systems [3] developed based on 10 different major versions of original Android operating system, running on more than 24,000 distinct device models [4]. The numerous combinations of system versions, customized features, system drivers, and device models make it infeasible, if not impossible, for developers to exhaustively test their apps for potential FIC issues. This further affects the qualities of these apps.

To handle FIC issues, much industrial and academic research has been conducted. Some research focuses on investigating and alleviating Android fragmentation. Han et al. found the empirical evidence on whether and where the fragmentation specifically exists within an Android project, and proposed an approach to examining fragmentation within Android [5]. Packages like `android.support.*` and projects like Treble have been introduced to help developers develop apps compatible with former Android versions such to alleviate version fragmentation [6]. Other research focuses on characterizing and detecting compatibility issues brought by fragmentation. Wei et al. summarized common patterns of FIC issues to an API-Context Pair Model and developed an automated tool named FicFinder to detect these FIC issues based on the model [2].

Although existing research makes many efforts, we found that they suffer from severe false positive problems. To determine the causes of such problems, we inspected existing research and found that: (1) a large number of issues reported by existing techniques occur in third-party libraries, which are beyond the developers' control, (2) static analyses of proposed techniques suffer from low precision when identifying FIC issues, (3) some issues cannot be detected due to the inadequacy of existing issue patterns and database, and (4) issue reports generated by proposed techniques contain inadequate information for developers to locate all occurring sites of detected FIC issues.

To deal with these problems, we present ELEGANT, an automated tool to effectively detect and locate fragmentation-induced compatibility issues based on the API-Context Pair Model [2]. In general, ELEGANT is a two-phase tool including a *Preprocessing Phase* and a *Locating Phase*. Specifically, in the Preprocessing Phase, ELEGANT isolates[1] the third-

[%] Chang Xu is the corresponding author.

[1] We use term "third-party library isolation" here to represent "third-party library elimination", because ELEGANT does not physically eliminated third-party libraries within the APK file.

party libraries within an app using either a whitelist-enhanced or an obfuscation-insensitive technique to deal with the first problem. In the Locating Phase, ELEGANT leverages a *three-step static detection algorithm* to increase the precision of static analysis: (1) in the first step, which we call the *Detection Step*, ELEGANT constructs a *call site tree* of the app under test (or AUT for short) to represent all potential *call path*s of a FIC issue, (2) in the second step, which we call the *Validation Step*, ELEGANT prunes the constructed call site tree to improve the precision of our analysis and eliminate the false positive call paths, and (3) in the last step, which we call the *Generation Step*, ELEGANT generates an issue report using the call site tree to provide detailed information for locating occurring sites of the issues. And this solves the second problem. To deal with the third one, ELEGANT models FIC issues using an improved API-Context Pair Model [2] and expands the default API-Context pair database of FicFinder [2]. With the call site tree constructed in the Locating Phase, ELEGANT is able to report adequate information for developers to locate occurring sites of validated issues, and this solves the last problem.

We experimentally evaluated ELEGANT with 22 real-world popular Android apps. Our results show that ELEGANT effectively detected and located real FIC issues within these apps with 70% less false positives compared with FicFinder [2].

We summary our contributions in this paper as follows:

- We proposed a two-phase approach to detecting and locating FIC issues within Android apps based on third-party library isolation and a three-step static detection algorithm.
- We implemented our prototype tool named ELEGANT[2] and evaluated it using real-world Android apps. ELEGANT effectively detected and located FIC issues within these apps with less false positives compared with an existing approach FicFinder [2].

The rest of the paper is organized as follows. Section II presents the proposed two-phase approach especially the three-step static detection algorithm. The overall architecture and implementation details of ELEGANT are elaborated on in Section III. Section IV presents our evaluation results. Section V elaborates on the limitations and our future work. Related work is discussed in Section VI, and finally Section VII concludes this paper.

II. Proposed Approach

This section presents our proposed approach. The overview of the approach is presented in Section II-A, and technical details are presented in Section II-B and II-C.

A. Overview

ELEGANT adopts a two-phase approach to detecting and locating FIC issues. The two-phase approach works as follows:

i. **Preprocessing Phase.** In this phase, the AUT is pruned for its contained third-party libraries, and transformed to an intermediate representation which integrates the AUT's call graph, control flow graph, inter-procedure program dependency graph (or *inter-PDG* for short), manifest information, etc..
ii. **Locating Phase.** Taking as input the intermediate representation, this phase adopts a three-step static detection algorithm to detect and locate FIC issues, and generate a detailed issue report.

B. Preprocessing Phase: Third-Party Library Isolation

Third-party libraries are widely used in Android apps. However, FIC issues within them are out of app developers' control. Hence reporting an issue occurring in a third-party library does more bothering than help to app developers. This motivates us to isolate third-party libraries from the AUT in the Preprocessing Phase.

In ELEGANT, the Preprocessing Phase by default uses a whitelist-enhanced technique to detect and isolate third-party libraries. We collect a list of 174 popular Android as well as Java libraries that are frequently used in Android app development. To accelerate searching, these libraries are ranked by a heuristic use-frequency score. The score is calculated as follows:

$$score(lib) = \alpha \cdot watch(lib) + \beta \cdot fork(lib) + \gamma \cdot star(lib)$$

where $watch(lib)$, $fork(lib)$, $star(lib)$ are the number of watches, forks, and stars of the library in GitHub [7], respectively, and $\alpha + \beta + \gamma = 1.0$. Following the user habits in GitHub, we eventually set the weight α to 0.15, β to 0.35 and γ to 0.5. For those libraries not hosted in GitHub, a heuristic score is applied to each of them according to its popularity.

The whitelist-enhanced technique is fast and accurate. However, considering that some apps use code obfuscation to change package names which results in the inability of the whitelist-enhanced technique to identify third-party libraries within these apps, an obfuscation-insensitive technique is needed. The Preprocessing Phase adopts an existing technique named LibScout [8] to isolate obfuscated third-party libraries. LibScout first extracts a code obfuscation-insensitive profile for each library and the AUT, then calculates a similarity score between each pair of the extracted profiles and its database to determine the third-party libraries.

C. Locating Phase: Three-Step Static Detection Algorithm

In this section, we present our three-step static detection algorithm used in the Locating Phase for detecting and locating FIC issues. The Locating Phase utilizes an API-Context Pair Model, which we adopt from FicFinder [2] and expand, to locate FIC issues. Thus, before presenting our three-step static detection algorithm, we elaborate on the API-Context Pair Model and our expansion.

1) Expansion of API-Context Pair Model: To automatically detect FIC issues, Wei et al. found that they are often triggered in a specific software and/or hardware context by improperly invoking some specific Android APIs. They summarized these common patterns to an API-Context Pair Model which is an representation of FIC issues. An API-Context pair is a pair of

[2] ELEGANT is available in https://github.com/Leetsong/ELEGANT/

Algorithm 1: Overall Algorithm

```
input : $db$, a database of API-Context pairs;
        $cg$, a call graph;
        $pdg$, an inter-PDG
output: $r$, an issue report
1 $r \leftarrow$ Map: API-Context pair $\mapsto$ list of call paths
2 foreach API-Context pair $acp \in db$ do
3     $cst \leftarrow$ Detect($acp.api, acp.context, cg$)
4     $cst \leftarrow$ Validate($cst, acp, pdg$)
5     $cpl \leftarrow$ Generate($cst.root$) // the $root$ property
          represents the root node of a call site tree
6     addToMap($r, acp, cpl$)
7 return $r$
```

an API (which they call an *issue-inducing API*) and a context (which they call an *issue-triggering context*), of which the API is represented using its standard definition, and the context is comprised of three types of conditions, i.e., the software environment, hardware environment, and API usage. They collected 25 API-Context pairs as a database, and their tool named FicFinder detects FIC issues based on this database [2]. However, we found that some issues cannot be detected using the default database of FicFinder [2]. Therefore, we expanded the database by adding 41 new API-Context pairs.

2) Overall Three-Step Static Detection Algorithm: The Locating Phase adopts a three-step static detection algorithm to detect and locate FIC issues, and generate a detailed report. For each API-Context pair in our expanded database, the algorithm runs the following three steps sequentially to accomplish different tasks:

i. **Detection Step**: in this step, ELEGANT searches for all potential *call paths* of a potential FIC issue, and constructs a *call site tree*.
ii. **Validation Step**: in this step, ELEGANT prunes the call site tree using program slicing technique [9] to eliminate false positive call paths.
iii. **Generation Step**: in this step, ELEGANT generates an issue report which contains detailed information for locating each validated issue.

Algorithm 1 presents the overall algorithm. It takes as inputs three parameters: (1) an API-Context pair database db, (2) a call graph cg, and (3) an inter-PDG pdg. The last two parameters are constructed from the AUT, with third-party libraries isolated. The algorithm produces as output an issue report containing detailed information for locating the FIC issues. Above all, an empty issue report r is constructed (Line 1). The report is a map mapping from API-Context pairs to their generated list of call paths. In this report, each API-Context pair represents a FIC issue, and its corresponding call paths are detailed information on how to locate it. Then for each API-Context pair acp in db (Line 2), firstly, a *call site tree* is constructed by invoking Detect (Line 3) with acp and cg; secondly, the tree cst is pruned by invoking Validate with acp and pdg (Line 4); and thirdly, by invoking Generate on the pruned call site tree cst, the call paths list cpl which contains detailed information for locating the issue represented by acp is generated (Line 5). Finally, the pair (acp, cpl) which represents both an issue and its detailed information is added to the report r (Line 6) which is the output of this algorithm (Line 7).

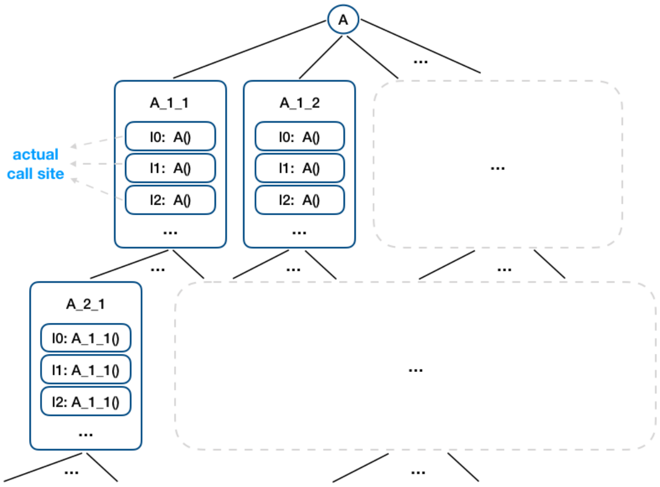

Fig. 1: Call Site Tree

The three steps are described in detail one by one next.

3) Detection Step: The Detection Step constructs a call site tree for each identified issue. A call site tree is a tree, each node of which represents a method within the AUT's code. The root node of a call site tree represents the corresponding issue-inducing API, and the children of each node represent all possible callers of its corresponding method. Additionally, each node has a $callsites$ property. The $callsites$ property is a set of call sites of its parent's corresponding method that occur in this node's method. The call site tree does not take the recursions into consideration. And each path from one call site of a leaf node up to the root node forms one call path of this issue, and each call path implies that it is potential to lead to this FIC issue. For instance, Figure 1 presents a call site tree, of which the root represents an issue-inducing API A. A_1_1, A_1_2, A_1_3, $\cdots$, are methods which call API A at call sites (or line) $l0$, $l1$, $l2$, $\cdots$, within them. Suppose the height of this tree is exactly three. Then $A\_2\_1 : l2 \rightarrow A\_1\_1 : l2 \rightarrow A$, $A\_2\_1 : l2 \rightarrow A\_1\_1 : l1 \rightarrow A$, $\cdots$, are all potential call paths, and each call path implies that the FIC issue related to the issue-inducing API A is potential to be manifested along it.

Compared with existing research whose issue report contains only the exact call site of a specific issue, i.e., the first layer like $A\_1\_1/A\_1\_2/\cdots$ of our call site tree, the issue report generated by our approach provides not only the exact call sites of an issue, but also the full information of its different call paths, which contain adequate information for developers to locate this issue. Moreover, the call site tree can be utilized by our static program analysis to produce

Algorithm 2: Detect

```
input  : $api$, an issue-inducing API;
         $ctx$, an issue-triggering context;
         $cg$, a call graph
output: $cst$, a call site tree
1  $cst \leftarrow$ CallSiteTree()
2  $root \leftarrow$ CallSiteTreeNode($api$)
3  if not checkContextSatisfying($ctx$) then
4      setRoot($cst, root$)
5      return $cst$
6  $css \leftarrow$ getAllCallSites($cg, api$)
7  $callers \leftarrow$ classifyCallSites($css$)
8  foreach method $m \in callers$ do
9      $csst \leftarrow$ Detect ($m, ctx, cg$)
10     setCallSites($csst.root, callers[m].callsites$)
        // the $callsites$ property represents the call sites
        of a node
11     append($root.children, csst.root$) // the
        $children$ property represents the children node
        set of a node
12 setRoot($cst, root$)
13 return $cst$
```

more precise results. Although the static analysis on an overall program will use the information inside the call graph, it is often inadequate to obtain a precise slice for a certain variable, which is used in the follow-up steps of the algorithm. The program slicing technique [9] looks for statements that are data- or control-dependent to a specific variable in a specific statement. For example, if we would like to find the slice of a variable v within the statement $l2$ of method A_2_1, the data- and control-dependents of the variables in $l0$ of method A_1_1 are also part of slice of v. However, due to the imprecise nature of static analysis, these can be dropped. But with call site tree, even they are dropped by the slicing technique, they can be taken into considerations when node A_1_1 is traversed.

Algorithm 2 presents the process of this construction step. It takes as inputs an issue-inducing API api, an issue-triggering context ctx, and a call graph cg of the AUT. It produces as output a call site tree. Firstly, an empty tree cst (Line 1) and a node $root$ representing api (Line 2) are constructed. Then whether the software/hardware environment (e.g., the minimum supported API level) the AUT declared in its manifest information satisfies that of declared in ctx is checked. If not satisfied (Line 3), for example, the issue-triggering context is API level 8, whereas the AUT's minimum supported level is 18 which is newer and will not trigger the issue, the tree cst containing only $root$ (Line 4) which indicates that no issues are detected is directly returned (Line 5). Otherwise, by traversing cg, all call sites css of api are found (Line 6). Considering there can be more than one call site inside a method, by classifying it is easy to determine which call site belongs to which caller method, and this helps in the subsequent assignment. Then all call sites css are then classified into different groups according to the method they inhabit, i.e., callers of api, and these groups are assigned to $callers$ (Line 7), so that the j-th call site within the i-th caller method can be referred to by $callers[i].callsites[j]$. Next, for each caller m, the Detection Step is recursively executed to construct their corresponding call site subtree $csst$ (Line 9), and each $csst.root$ is set its corresponding call sites (Line 10) as well as appended to $root$ as one of its children node (Line 11). Finally, $root$ containing all its children node is set to cst (Line 12), which is the output of this algorithm (Line 13).

4) Validation Step: This step prunes the constructed call site tree using the program slicing technique [9]. Given that some app developers have already been aware of some FIC issues, and written compatible code to adapt different context, e.g., an app developer can use `Build.VERSION.SDK_INT` to check the Android API level and write different code to adapt different levels to evade these issues. In practice, there are several patterns developers frequently use to fix these issues[3], and heuristic pattern matching technique enables to determine the existence of these patterns. Once they are determined to exist, these issues are considered as fixed, and it is not suitable for us to report all call paths of the constructed call site tree from the Detection Step. From the perspective of the call site tree, if any compatible code is written in some nodes' corresponding methods, call paths related to these nodes are considered false positives, which will be pruned.

Moreover, we found that some API-Context pairs are semantics related, and cannot be validated with static analyses, e.g., program slicing technique [9]. For instance:

- `AlarmManager.set()`: According to the official documents, "beginning in API 19, the trigger time passed to this method is treated as inexact: the alarm will not be delivered before this time, but may be deferred and delivered some time later" [10]. With different Android API levels, this API sets alarms at different time, either exact or not. Static program analysis cannot deduce whether the app developer would like to use an exact time or not.
- `AsyncTask.execute()`: According to the official documents, "this function schedules the task on a queue for a single background thread or pool of threads depending on the platform version" [11]. With different Android API levels, this API behaves differently. It is impossible to deduce the app developer's intention of whether to use it in parallel or not via static program analysis, either.

Static program analyses are not capable of deducing the semantics related properties described above. Hence once the corresponding API-Context pairs are used improperly, false negatives will be produced. Although impossible to deduce, developers should be warned about these APIs' hidden traps, and the related call site tree should not be pruned. Thus, we introduced an mechanism called *Important Fields* to the API-Context Pair Model. With it, the context of API-Context

[3] Collected patterns are in https://github.com/Leetsong/ELEGANT/issues/12.

Algorithm 3: Validate

```
input  : $cst$, a call site tree;
         $acp$, an API-Context pair;
         $pdg$, an inter-PDG
output: $cst$, a pruned call site tree
1  if $acp.important\_fields$ satisfied then
2      return $cst$;
3  $q \leftarrow$ Queue ($cst.root$)
4  while $q \neq \emptyset$ do
5      $n \leftarrow$ deQueue($q$)
6      foreach child node $c \in node.children$ do
7          enQueue($q, c$)
8      foreach call site $cs \in n.callsites$ do
9          $slice \leftarrow$ backwardSlicing($cs, pdg$)
10         foreach statement $s \in slice$ do
11             if $s$ can fix this issue then
12                 delCallSite($n, cs$)
13                 break
14     if $n.callsites = \emptyset$ then
15         while hasNoSiblings($n$) and
            $cst.root \neq n.parent$ do
16             $n \leftarrow n.parent$ // the $parent$ property
                represents the parent node of this node
17         delNode($cst, n$)
18 return $cst$
```

Pair Model is extended with a new condition called important fields, and it can be formalized in the following context-free grammar:

$$\begin{aligned} &Context \rightarrow Condition \mid Condition \wedge Context \\ &Condition \rightarrow Software\_env \mid Hardware\_env \\ &\qquad\qquad\quad \mid API\ usage \mid Important\ fields \end{aligned}$$

The important fields specify constraints on software environment, hardware environment and API usage. If an API-Context pair uses this mechanism, the three-step static detection algorithm will skip the Validation (2nd) Step and directly step into the Generation (3rd) Step if any issues are actually detected in the Detection (1st) Step. Taking `AlarmManager.set()` as an example,

$$\begin{aligned} \textbf{API}: &\ AlarmManager.set() \\ \textbf{Context}: &\ min\_API\_level = 19 \\ &\ \wedge\ Important\_fields = [\text{``}min\_API\_level\text{''}] \end{aligned}$$

in the Detection Step, the AUT will be examined to determine whether the software environment $min\_API\_level = 19$ is satisfied. Given that it uses the *Important Fields* mechanism, and its $Important\_fields$ is "min_API_level", if the constraints are satisfied, the Validation Step will be skipped.

Algorithm 4: Generate

```
input  : $csn$, a call site tree node
output: $paths$, a list of call paths of $csn$
1  $paths \leftarrow$ an empty array
2  foreach child node $c \in csn.children$ do
3      $cpl \leftarrow$ Generate($c$)
4      if $cpl = \emptyset$ then
5          foreach call site $s \in c.callsites$ do
6              append($paths$, Array($s$))
7      else
8          foreach call site $s \in c.callsites$ do
9              foreach call path $cp \in cpl$ do
10                 $p \leftarrow$ concat($cp, s$)
11                 append($paths, p$)
12 return $paths$
```

If the detected issue's corresponding API-Context pair has no important fields or the constraints are not satisfied, its call site tree is examined to determine whether each call site of each call path is fixed. Once a call site is determined fixed by the app developer, its belonged call paths will be pruned. Algorithm 3 presents the process of this pruning step. It takes as inputs a call site tree cst, an API-Context pair acp, and an inter-PDG pdg. It produces as output a pruned call site tree. A breadth first search is adopted in this step. Firstly, for those API-Context pairs who use *Important Fields* mechanism, the important fields will be examined to determine whether satisfied like the checking process of the example in last paragraph, and if so (Line 1), this pruning step is skipped and the unpruned cst is directly returned (Line 2). Otherwise, a node queue q containing only $root$ is constructed (Line 3). Whenever q is not empty (Line 4), the head n of it is popped (Line 5), and all its children are appended to q (Line 6-7). Then for each call site cs of each node n (Line 8), the static backward program slice $slice$ of cs over pdg is obtained using program slicing technique [9] (Line 9). For each statement s of $slice$ (Line 10), once it is determined that it can fix the FIC issue (Line 11), the call site cs is deleted from node n (Line 12). Hence all the call paths related to cs are in result deleted. Furthermore, once all call sites of node n are deleted (Line 14), n contributes nothing to the call paths and therefore is also deleted (Line 17). This deleting process goes along each call path from bottom up to the root node and all its ancestor nodes that satisfy this condition (Line 14) are deleted (Line 14-17). Finally, the pruned call site tree cst is returned (Line 18). Once the returned call site tree cst contains only a root, this issue is a false positive and is immediately dropped.

5) Generation Step: This step generates the validated call paths of a call site tree. As aforementioned, each call path validated in last step potentially leads to its corresponding FIC issue, so generating it to developers is necessary. Algorithm 4 presents the process of this generation step. It takes as input

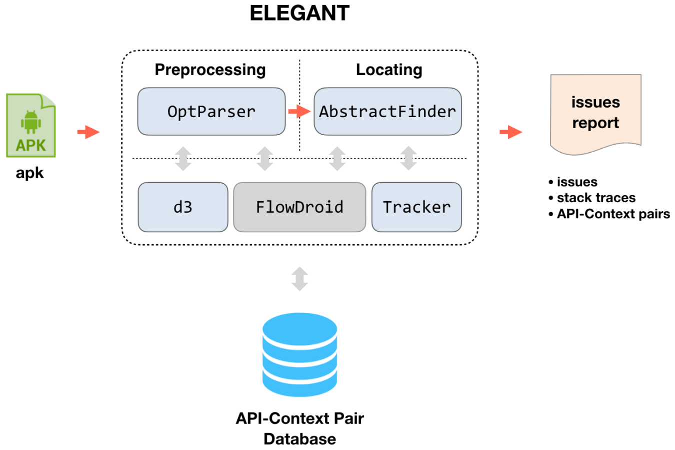

Fig. 2: ELEGANT Architecture

a call site tree node csn, and produces as output a list of call paths of its input csn. For csn, each of its child nodes such as c is traversed (Line 2), and the list of call paths from the leaf nodes up to c are generated as cpl (Line 3). Then each call site s of node c is concatenated to each recursively generated call path cp (Line 5-6 and 10), such to generate call paths of csn (Line 6 and 11). Finally these paths $paths$ are returned as the detailed information of this issue (Line 12).

III. ELEGANT Architecture

This section presents the overall architecture and implementation details of ELEGANT. Figure 2 presents the work- and data-flow of ELEGANT. It takes as input an apk file, and produces as output an issue report that contains adequate information for locating FIC issues. Specifically, ELEGANT, on top of `d3`, `Tracker` and `FlowDroid`, is composed of an `OptParser`, and an `AbstractFinder`. ELEGANT works as follows:

i. When an apk file f is sent to ELEGANT, `OptParser` isolates the third-party libraries, and transforms it to an intermediate representation im_f with the help of `d3` and `FlowDroid`. This accomplishes the Preprocessing Phase of our proposed approach.
ii. Taking as input im_f, `AbstractFinder` examines it to locate FIC issues within it, and then emits those validated with the help of `Tracker`. This accomplishes the Locating Phase of our proposed approach.

A. FlowDroid

An Android app is composed of many event handlers, with no specifically designed main entry like `public static void main(String[] args)` in plain Java. As a result, it is a difficult task to construct a control flow graph and perform static analyses such as data flow analysis on it. FlowDroid [12], developed by A. Steven on top of Soot [13], accomplishes it using specifically designated sources and sinks. It is designed as a context-, flow-, field-, object-sensitive and lifecycle-aware static taint analysis tool for Android apps. In ELEGANT, FlowDroid [12] is used as a module `FlowDroid` to assist `OptParser` to construct the intermediate representation im_f of the apk f, including the call graph, the control flow graph, the inter-PDG, the manifest information, etc.

B. d3

The `d3`, short for detecting 3rd-party libraries, is the module that assists `OptParser` to detect and isolate third-party libraries. The `d3` module by default adopts the whitelist-enhanced technique we presented in Section II-B to isolate third-party libraries. Moreover, the code obfuscation-insensitive technique is also available in ELEGANT. To choose an appropriate technique, we compared LibScout [8], LibRadar [14], as well as LibD [15], and finally set down to LibScout [8] based on the following considerations:

- LibRadar [14] and LibD [15] are techniques based on code clustering. These techniques suppose that the third-party libraries are widely used, and cannot detect libraries that are used rarely.
- It is difficult for us to extend the database of LibRadar [14] and LibD [15] when a new library is found, because the databases of them are trained with powerful servers, taking tens or even hundreds of hours. However, it is easy to extract the profile of any newly found libraries using LibScout [8] and add them to its database.

To offer convenience for those who want the issues to be overall reported no matter it is in third-party libraries or not, a switch to disable this isolation step is available in ELEGANT.

C. Tracker

The `Tracker` is a component that is implemented as a message deliverer based on the PubSub pattern, like Redis PubSub [16], which provides convenience to developers interested in extending our implementation. With the help of the `Tracker`, any types of messages can be delivered to any components in ELEGANT. And ELEGANT uses the `Tracker` mainly for emitting the validated issues.

D. OptParser

The `OptParser` accomplishes the Preprocessing Phase. It is a component that parses options and preprocesses the apk file. With the help of `d3`, the `OptParser` isolates all third-party libraries from the apk f. With the help of `FlowDroid`, the `OptParser` transforms the raw binary apk file f to an intermediate code representation Jimple [17], and further collects the f's manifest information, constructs the call graph, the control flow graph, the inter-PDG, and finally integrates them all into the intermediate representation im_f.

E. AbstractFinder

The `AbstractFinder` accomplishes the Locating Phase. With the help of our expanded API-Context pair database, the `AbstractFinder` uses the three-step static detection algorithm to detect FIC issues, validate their call site trees, and generate the issue report.

TABLE I: App Information

ID	App	Category	Revision
1	1Sheeld[1,2] [18]	Tools	b49c98a[1];ebac88f[2]
2	AnkiDroid[1,2] [19]	Education	dd654b6[1];3e007be[2]
3	AntennaPod[1,2] [20]	Media & Video	6f15660[1];9b20aea[2]
4	A.S.Keyboard[1,2] [21]	Tools	d0be248[1];561e67e[2]
5	BankDroid[1,2] [22]	Finance	f491574[1];a1bf663[2]
6	BitcoinWallet[2] [23]	Finance	8c35288[2]
7	BraveBrowser[2] [24]	Personalization	792217a[2]
8	c:geo[2] [25]	Entertainment	2056b66[2]
9	ChatSecure[2] [26]	Communication	e09c5a8[2]
10	ConnectBot[1,2] [27]	Communication	49712a1[1];49001ec[2]
11	Conversations[1,2] [28]	Communication	1a073ca[1];f58c173[2]
12	iNaturalist[2] [29]	Education	c129bdf[2]
13	IrssiNotifier[1,2] [30]	Communication	ad68bc3[1];7cebdc6[2]
14	K-9 Mail[1,2] [31]	Communication	74c6e76[1];83fc4c8[2]
15	Kore[1,2] [32]	Media & Video	0b73228[1];e6470a1[2]
16	O.G.Tracker[2] [33]	Travel & Local	b817c98[2]
17	OpenVPN[2] [34]	Communication	b3a7f45[2]
18	OwnCloud[2] [35]	Productivity	e188407[2]
19	Pac.Droid[1] [36]	Communication	1090758[1]
20	QKSMS[1,2] [37]	Communication	73b3ec4[1];8233220[2]
21	Transdroid[1,2] [38]	Tools	28786b0[1];22670e9[2]
22	WordPress[1] [39]	Social	efda4c9[1]

[1] Apps/Revisions used in the comparison experiments.
[2] Apps/Revisions used in the separate experiment.

IV. Evaluation

We evaluated ELEGANT with real-world Android apps. Our evaluation aims to answer the following three research questions:

- **RQ1:** (Effectiveness of ELEGANT) Is the two-phase approach used by ELEGANT effective in detecting more issues with a lower false positive rate?
- **RQ2:** (Effectiveness of Third-Party Library Isolation) Is the third-party library isolation effective in further reducing the false positive rate?
- **RQ3:** (Effectiveness of Expansion) Is the expansion of the API-Context pair database effective in detecting more issues with a lower false positive rate?

A. Experiment Setup

Our experiments were conducted on macOS High Sierra 10.13.4, running on a MacBook Air (2014), with a 1.4GHz Core i5 CPU, and 4GB memory.

To answer the research questions, we conducted four experiments including three comparison experiments with FicFinder [2] using 14 real-world Android apps used to evaluate FicFinder with the *same revisions*[4], and one separate experiment using another set of 20 real-world Android apps with the *newest source-code-available revisions*. The apps' information, including their names, categories, and revisions used in our experiments, are listed in Table I.

Specifically, to answer the first question, i.e., to evaluate whether the two-phase approach used by ELEGANT is effective, we conducted a comparison experiment between ELEGANT and FicFinder with 14 apps originally used to evaluate FicFinder [2]. To further demonstrate the effectiveness of ELEGANT even on modern apps, we intentionally selected 20 real-world Android apps with the *newest source-code-available revisions* to exercise evaluation. To answer the second question, i.e., to evaluate whether our third-party library isolation is effective, we conducted a comparison experiment using ELEGANT with the module `d3` enabled and disabled. For the last one, i.e., to evaluate whether the expansion of our API-Context pair database is effective, we conducted a comparison experiment between the default API-Context pair database (of size 25) used by FicFinder [2] and our expanded database (of size 66). This experiment was conducted using FicFinder [2].

The experimental results of each AUT are presented in the form of "a/b" or "a/b/c", where "a" indicates the number of validated APIs, "b" indicates the number of validated exact call sites, and "c" the number of validated call paths. The reported issues are classified into either true positives or false positives. An issue will be classified as a false positive if all compatible code that satisfy its context in API-Context Pair Model can be found in the source code of the AUT. For example, supposing an issue is related to API `Activity.onKeyDown()` whose model is,

$$\begin{aligned} \textbf{API}: &\ Activity.onKeyDown() \\ \textbf{Context}: &\ min\_API\_level = 16 \\ &\wedge\ bad\_devices = [\text{“}LG\text{”}, \text{“}LGE\text{”}] \end{aligned}$$

if (1) code that adapt API level less than 16 and greater than 16, and (2) code that adapt LG/LGE/other devices, can both be found, this issue is classified as a false positive. In the result table, we use "ALL" to represent the total number of issues reported, "TP" to represent the number of true positives, and "FP" the false positives. To show effectiveness, we compare the number of false positives, and then discuss the improvement of our work. In addition, to reduce errors of manual checking, we cross-validated all these results.

B. RQ1: Effectiveness of ELEGANT

The first experimental results are listed in Table II. The results show that:

i. ELEGANT reported issues $32/133/514$, including $27/106/407$ TPs and $5/27/107$ FPs. The rate of FP is $0.156/0.203/0.208$.
ii. FicFinder reported issues $51/196/-$, including $24/107/-$ TPs and $27/89/-$FPs. The rate of FP is $0.529/0.454/-$.

These results confirm that ELEGANT is more effective than FicFinder: although ELEGANT reports less issues, TPs reported are much more and FPs much less, and the rate of FP decreases from $0.529/0.454/-$ to $0.156/0.203/0.208$, which makes the performance increase by $70.5\%/54.2\%/-$.

To further demonstrate the effectiveness of ELEGANT even on modern apps, we intentionally selected 20 real-world apps with the *newest source-code-available revisions* as listed in

[4] Some apps' revisions are not source-code-available at the time the experiments were conducted, so we abandon such apps

TABLE II: Effectiveness of ELEGANT - 1[#]

App	ELEGANT			FicFinder [2]		
	ALL	TP	FP	ALL	TP	FP
1Sheeld	3/12/13[*]	3/12/13	0/0/0	2/4/-[+]	2/4/-	0/0/-
BankDroid	4/12/20	3/11/13	1/1/7	3/4/-	2/3/-	1/1/-
AnkiDroid	5/25/243	3/15/165	2/10/78	8/28/-	1/3/-	7/25/-
AntennaPod	3/16/35	2/14/33	1/2/2	4/16/-	1/2/-	3/14/-
A.S.Keyboard	1/15/42	1/15/42	0/0/0	4/11/-	3/10/-	1/1/-
ConnectBot	1/2/3	1/2/3	0/0/0	0/0/-	0/0/-	0/0/-
Conversations	2/3/42	2/3/42	0/0/0	4/7/-	3/4/-	1/3/-
IrssiNotifier	1/5/21	1/5/21	0/0/0	7/21/-	0/0/-	7/21/-
K-9 Mail	5/6/13	5/6/13	0/0/0	4/11/-	4/10/-	0/1/-
Kore	1/11/30	1/11/30	0/0/0	2/13/-	1/11/-	1/2/-
Pac.Droid	0/0/0	0/0/0	0/0/0	1/2/-	1/2/-	0/0/-
QKSMS	2/2/9	2/2/9	0/0/0	4/8/-	3/6/-	1/2/-
Transdroid	1/4/4	1/4/4	0/0/0	3/14/-	1/5/-	2/9/-
WordPress	3/20/39	2/6/19	1/14/20	5/57/-	2/38/-	3/19/-
Total	32/133/514	27/106/407	5/27/107	51/196/-	24/107/-	27/89/-
FP-Rate	0.156/0.203/0.208			0.529/0.454/-		

[#] The experiment uses apps/revisions superscritped with ^{1} in Table I.

[%] The experiment uses apps/revisions superscritped with ^{2} in Table I.

[*] The experimental results of each AUT are represented in the form of “a/b/c”, where “a” indicates the number of validated APIs, “b” indicates the number of validated exact call sites, and “c” the number of validated call paths.

[+] FicFinder cannot report call paths, “-” is used instead.

TABLE III: Effectiveness of ELEGANT - 2[%]

APP	ALL	TP	FP
1Sheeld	1/1/1[*]	1/1/1	0/0/0
AnkiDroid	5/16/72	2/6/50	3/10/22
AntennaPod	1/2/3	0/0/0	1/2/3
A.S.Keyboard	0/0/0	0/0/0	0/0/0
BankDroid	4/9/17	3/8/10	1/1/7
BitcoinWallet	0/0/0	0/0/0	0/0/0
c:geo	0/0/0	0/0/0	0/0/0
ChatSecure	3/6/7	2/5/6	1/1/1
ConnectBot	1/1/1	1/1/1	0/0/0
Conversations	1/21/23	1/21/23	0/0/0
iNaturalist	0/0/0	0/0/0	0/0/0
IrssiNotifier	1/5/7	1/5/7	0/0/0
K-9Mail	4/4/7	3/3/6	1/1/1
Kore	1/5/11	1/5/11	0/0/0
LinkBubble	1/1/1	0/0/0	1/1/1
O.G.Tracker	2/8/10	2/8/10	0/0/0
OpenVPN	0/0/0	0/0/0	0/0/0
OwnCloud	1/1/27	1/1/27	0/0/0
QKSMS	1/1/11	1/1/11	0/0/0
Transdroid	0/0/0	0/0/0	0/0/0
Total	27/81/198	19/65/163	8/16/35
FP-Rate	0.296/0.198/0.177		

Table I and conducted a separate experiment. ELEGANT was configured to enable `d3` and use our expanded database. The results are presented in Table III. As shown, ELEGANT reported 27/81/198 issues, with TPs 19/65/163, and FPs 8/16/35. The rate of FP is 0.296/0.198/0.177. These results further demonstrate the effectiveness of ELEGANT.

With such results, we can answer **RQ1**.

> **Answer to RQ1:** The two-phase approach used by ELEGANT is effective in detecting more issues with a lower false positive rate.

C. RQ2: Effectiveness of Third-Party Library Isolation

Table IV presents the results:

i. Without `d3`, ELEGANT reported issues 37/143/551, including 27/106/407 TPs and 10/37/144 FPs. The rate of FP is 0.270/0.259/0.261.
ii. With `d3`, ELEGANT reported issues 32/133/514, including 27/106/407 TPs and 5/27/107 FPs. The rate of FP is 0.156/0.203/0.208.

These results show that ELEGANT is more effective when module `d3` is enabled: with `d3`, ELEGANT can isolate third-party libraries and reduce the FPs: the false positive rate decreases by 42.22%/21.62%/20.31%. Hence we can answer **RQ2**.

> **Answer to RQ2:** Our third-party library isolation is effective in further reducing the false positive rate.

D. RQ3: Effectiveness of Expansion

The results are presented in Table V:

i. With the default database, FicFinder reported 31/87 FIC issues, with TPs 8/33 and FPs 23/54. The rate of FP is 0.742/0.621.
ii. With our expanded database, FicFinder reported 51/196 issues, with TPs 24/107, and FPs 27/89. The rate of FP is 0.529/0.454.

These results show that the performance of FicFinder has improved:

i. In the reported issues, TPs increase by 200.00%(from 8 to 24)/224.2%(from 33 to 107).
ii. Within the additionally reported issues, only 4/35 are considered as FPs. The rate of FP decreases from 0.742/0.621 to 0.529/0.454, by 28.71%/26.89%.

After further investigation, we found that the reason why the performance is improved is that, our expanded database contains some error-prone API-Context pairs to which, however, developers always attach less importance, e.g., `Resources.getDrawable()`, `Activity.onKeyDown()`, `ContentValues.put()`. Hence we get the answer to **RQ3**.

> **Answer to RQ3:** Our expansion of API-Context pair database is effective in detecting more issues with a lower false positive rate.

V. Limitations and Future Work

Evolving of API-Context Pair Database. Android ecosystem is evolving very quickly. The API-Context pair database has to evolve with Android, or it will be out of date and lose its effectiveness. However, it is always difficult and time-consuming for us to extract API-Context pairs from various bug reports. And an automated approach is in urgent need.

TABLE IV: Effectiveness of Third-Party Library Isolation[#]

App	Enable d3			Disable d3		
	ALL	TP	FP	ALL	TP	FP
1Sheeld	3/12/13[*]	3/12/13	0/0/0	3/13/16	3/12/13	0/1/3
BankDroid	4/12/20	3/11/13	1/1/7	4/12/20	3/11/13	1/1/7
AnkiDroid	5/25/243	3/15/165	2/10/78	5/25/262	3/15/165	2/10/97
AntennaPod	3/16/35	2/14/33	1/2/2	4/17/36	2/14/33	2/3/3
A.S.Keyboard	1/15/42	1/15/42	0/0/0	1/15/42	1/15/42	0/0/0
ConnectBot	1/2/3	1/2/3	0/0/0	1/2/3	1/2/3	0/0/0
Conversations	2/3/42	2/3/42	0/0/0	2/3/42	2/3/42	0/0/0
IrssiNotifier	1/5/21	1/5/21	0/0/0	4/8/26	1/5/21	3/3/5
K-9 Mail	5/6/13	5/6/13	0/0/0	5/7/18	5/6/13	0/1/5
Kore	1/11/30	1/11/30	0/0/0	2/12/31	1/11/30	1/1/1
Pac.Droid	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
QKSMS	2/2/9	2/2/9	0/0/0	2/2/9	2/2/9	0/0/0
Transdroid	1/4/4	1/4/4	0/0/0	1/4/4	1/4/4	0/0/0
WordPress	3/20/39	2/6/19	1/14/20	3/23/42	2/6/19	1/17/23
Total	32/133/514	27/106/407	5/27/107	37/143/551	27/106/407	10/37/144
FP-Rate	0.156/0.203/0.208			0.270/0.259/0/261		

[#] The experiment uses apps/revisions superscritped with ^{1} in Table I.
[*] The experimental results of each AUT are represented in the form of “a/b/c”, where “a” indicates the number of validated APIs, “b” indicates the number of validated exact call sites, and “c” the number of validated call paths.

TABLE V: Effectiveness of Expansion[#]

App	Default			Expanded		
	ALL	TP	FP	ALL	TP	FP
1Sheeld	1/3[*]	1/3	0/0	2/4	2/4	0/0
BankDroid	1/1	0/0	1/1	3/4	2/3	1/1
AnkiDroid	4/19	0/0	4/19	8/28	1/3	7/25
AntennaPod	2/4	1/2	1/2	4/16	1/2	3/14
A.S.Keyboard	3/4	2/3	1/1	4/11	3/10	1/1
ConnectBot	0/0	0/0	0/0	0/0	0/0	0/0
Conversations	3/3	2/2	1/1	4/7	3/4	1/3
IrssiNotifier	5/9	0/0	5/9	7/21	0/0	7/21
K-9 Mail	2/2	2/2	0/0	4/11	4/10	0/1
Kore	1/11	1/11	0/0	2/13	1/11	1/2
Pac.Droid	1/2	1/2	0/0	1/2	1/2	0/0
QKSMS	3/4	2/3	1/1	4/8	3/6	1/2
Transdroid	2/6	1/5	1/1	3/14	1/5	2/9
WordPress	3/19	0/0	3/19	5/57	2/38	3/19
Total	31/87	8/33	23/54	51/196	24/107	27/89
FP-Rate	0.742/0.621			0.529/0.454		

[#] The experiment uses apps/revisions superscritped with ^{1} in Table I.
[*] The experimental results of each AUT are represented in the form of “a/b”, where “a” indicates the number of validated APIs, “b” indicates the number of validated exact call sites.

Thus, we plan to develop an automated technique to extract API-Context pairs from various Android compatibility issues.

Expansion of Third-Party Libraries. As Android evolves, more and more third-party libraries will be developed and used. By far, we have to manually add the newly found and developed third-party libraries to our `d3` module so that the module can continuously and effectively work. This forms a threat to ELEGANT.

Drawbacks of API-Context Pair Model. As aforementioned, API-Context Pair Model is incapable of modeling semantics related APIs. Although we add an *Important Fields* mechanism to it, it is still weak. And this is where the false positives are produced. Thus, we are planning to improve the API-Context Pair Model to make it more powerful in dealing with semantics related issues.

VI. Related Work

Android fragmentation and compatibility issues have been regarded as very important problems among researchers and developers for a long time. Many researchers and engineers have investigated into these problems. We discuss some of them in this section.

In 2011, Android Compatibility program, led by Google, was released [40]. It defines technical details of the Android platform and provides tools for OEMs to ensure that apps function normally on a variety of devices. Later in 2012, Han et al. found the empirical evidence on the existence of fragmentation within the Android project via analyzing the bug reports related to HTC and Motorola [5]. They also proposed an approach to examining fragmentation within Android systems [5]. In 2013, McDonnell et al. investigated the impact of API evolution on software ecosystems. They confirmed that the client adoption was not keeping pace with API evolution, and further found that the API updates are more defect-prone than other types of changes in the client code [41]. These findings indicate the impact of API evolution to the compatibility issues. Hence in 2014, starting from Android 5.0, packages like `android.support.*` have been introduced to help developers develop apps compatible with APIs of previous versions. Ham et al. also proposed a compatibility test system to handle Android fragmentation problems [42]. Later in 2016, Wei et al. conducted an empirical study on Android fragmentation. They investigated into the compatibility issues brought by it, and summarized common patterns of them to an API-Context Pair Model. Based on the model, they proposed an automated tool named FicFinder to detect these FIC issues [2]. And project Treble, included within Android

8.0 in 2017, was also proposed to make it easier, faster, and less costly for OEMs to update devices to a new version of Android such to alleviate version fragmentation [6]. In 2018, Wei et al. conducted a larger empirical study on FIC issues to provide a more comprehensive understanding on them [43].

VII. Conclusion

In this paper, we presented ELEGANT, an automated two-phase tool to effectively detect and locate FIC issues. Based on our expanded API-Context pair database of size 66, it first isolates third-party libraries within an app in its Preprocessing Phase, then adopts a three-step static detection algorithm to detect and locate FIC issues in its Locating Phase. Our evaluation of ELEGANT involves three comparison experiments using 14 real-world Android apps and one separate experiment using 20 real-world Android apps, and the results confirm the effectiveness of ELEGANT on detecting and locating these issues for Android apps.

Acknowledgement This work is supported in part by National Basic Research 973 Program (Grant #2015CB352202), National Natural Science Foundation (Grants #61690204, #61472174) of China, and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

[1] Global market share held by the leading smartphone operating systems in sales to end users from 1st quarter 2009 to 1st quarter 2018. [Online]. Available: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

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

[3] List of custom Android distributions. [Online]. Available: https://en.wikipedia.org/wiki/List_of_custom_Android_distributions

[4] Android Fragmentation (August 2015). [Online]. Available: https://opensignal.com/reports/2015/08/android-fragmentation/

[5] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, "Understanding android fragmentation with topic analysis of vendor-specific bugs," in *2012 19th Working Conference on Reverse Engineering*, Oct 2012, pp. 83–92.

[6] Treble. [Online]. Available: https://source.android.com/devices/architecture/treble

[7] GitHub. [Online]. Available: https://github.com

[8] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 356–367. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978333

[9] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449. [Online]. Available: http://dl.acm.org/citation.cfm?id=800078.802557

[10] Android Docs, AlarmManager. [Online]. Available: https://developer.android.com/reference/android/app/AlarmManager.html

[11] Android Docs, AsyncTask. [Online]. Available: https://developer.android.com/reference/android/os/AsyncTask.html

[12] 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 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 259–269. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594299

[13] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, pp. 13–. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008

[14] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: Fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 653–656. [Online]. Available: http://doi.acm.org/10.1145/2889160.2889178

[15] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: Scalable and precise third-party library detection in android markets," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 335–346. [Online]. Available: https://doi.org/10.1109/ICSE.2017.38

[16] Redis Pub/Sub. [Online]. Available: https://redis.io/topics/pubsub

[17] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations," 1998.

[18] 1Sheeld. [Online]. Available: https://github.com/Integreight/1Sheeld-Android-App.git

[19] AnkiDroid. [Online]. Available: https://github.com/ankidroid/Anki-Android.git

[20] AntennaPod. [Online]. Available: https://github.com/AntennaPod/AntennaPod.git

[21] AnySoftKeyboard. [Online]. Available: https://github.com/AnySoftKeyboard/AnySoftKeyboard.git

[22] BankDroid. [Online]. Available: https://github.com/liato/android-bankdroid.git

[23] BitcoinWallet. [Online]. Available: https://github.com/bitcoin-wallet/bitcoin-wallet

[24] BraveBrowser. [Online]. Available: https://github.com/brave/browser-android.

[25] cgeo. [Online]. Available: https://github.com/cgeo/cgeo

[26] ChatSecure. [Online]. Available: https://github.com/guardianproject/ChatSecureAndroid

[27] ConnectBot. [Online]. Available: https://github.com/connectbot/connectbot.git

[28] Conversations. [Online]. Available: https://github.com/siacs/Conversations.git

[29] iNaturalist. [Online]. Available: https://github.com/inaturalist/iNaturalistAndroid

[30] IrssiNotifier. [Online]. Available: https://github.com/murgo/IrssiNotifier.git

[31] K-9 Mail. [Online]. Available: https://github.com/k9mail/k-9.git

[32] Kore. [Online]. Available: https://github.com/xbmc/Kore.git

[33] OpenGPSTracker. [Online]. Available: https://github.com/rcgroot/open-gpstracker

[34] OpenVPN. [Online]. Available: https://github.com/OpenVPN/openvpn

[35] OwnCloud. [Online]. Available: https://github.com/hypery2k/owncloud

[36] PactrackDroid. [Online]. Available: https://github.com/firetech/PactrackDroid.git

[37] QKSMS. [Online]. Available: https://github.com/moezbhatti/qksms.git

[38] Transdroid. [Online]. Available: https://github.com/erickok/transdroid.git

[39] WordPress. [Online]. Available: https://github.com/wordpress-mobile/WordPress-Android.git

[40] Android Compatibility Program. [Online]. Available: https://source.android.com/compatibility/

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

[42] H. K. Ham and Y. B. Park, "Designing knowledge base mobile application compatibility test system for android fragmentation," *International Journal of Software Engineering and Its Applications*, vol. 8, no. 1, pp. 303–314, jan 2014. [Online]. Available: https://doi.org/10.14257%2Fijseia.2014.8.1.26

[43] L. Wei, Y. Liu, S. C. Cheung, H. Huang, and X. Liu, "Understanding and detecting fragmentation-induced compatibility issues for android apps," *IEEE Transactions on Software Engineering*, 2018.