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 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 investigated into, and proposed promising techniques for, detecting these issues, which, however, can 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-sensitive 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 heavy compatibility issues (we follow previous research [2] and name such issues fragmentation-induced compatibility issues, or compatibility 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]. As a result, 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 compatibility issues, and this further impacts the qualities of these apps.

Therefore, to handle fragmentation-induced compatibility issues, much industrial and academic research has been conducted. Some research focuses on investigating and alleviating Android fragmentation. D. 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. L. Wei et al. summarized common patterns of the compatibility issues to an API-Context Pair Model and developed an automated tool named FieFinder to detect these fragmentation-induced compatibility 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 a third-party library, which is beyond the developers’ control, and (2) static analyses of proposed techniques lack precision when identifying fragmentation-induced compatibility issues. In addition to false positive problems, we also found that: (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 compatibility 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]. Technically, ELEGANT is a two-phase tool including a Preprocessing Phase and a Locating Phase. Detailedly, ELEGANT isolates the third-party libraries within an app...
using either a whitelist-enhanced or an obfuscation-sensitive technique in the Preprocessing Phase to deal with the first problem. For the second one, ELEGANT uses a three-step static detection algorithm in the Locating Phase 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 to represent all potential stack traces of an compatibility 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 stack traces, 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. To deal with the third one, ELEGANT models fragmentation-induced compatibility 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 fragmentation-induced compatibility 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 fragmentation-induced compatibility issues within Android apps based on third-party library isolation and a three-step static detection algorithm.
- We implemented our prototype tool named ELEGANT and evaluated it using real-world Android apps. ELEGANT effectively detected and located compatibility issues within these apps with less false positives compared with existing approaches.

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. Related work is discussed in Section V, and finally Section VI 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, respectively.

A. Overview

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

i. Preprocessing Phase. In this phase, the app under test is pruned for its contained third-party libraries, and transformed to an intermediate representation which integrates the app’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 fragmentation-induced compatibility issues, and generate a detailed issue report.

B. Preprocessing Phase: Third-Party Library Isolation

Third-party libraries are widely used in Android apps. The compatibility 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 app under test 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 \(\alpha\) to 0.15, \(\beta\) to 0.35 and \(\gamma\) 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-sensitive 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-sensitive profile for each library and the app under test, 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 fragmentation-induced compatibility issues. The Locating Phase utilizes an API-Context Pair Model, which we adopt from FicFinder [2] and expand, to locate the fragmentation-induced compatibility 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 fragmentation-induced compatibility issues, L. 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
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 $r \leftarrow \text{Map}: \text{API-Context pair} \mapsto \text{list of stack traces}$
2 foreach API-Context pair $acp \in db$
3   $cst \leftarrow \text{Detect}(acp.api, acp.context, cg)$
4   $cst \leftarrow \text{Validate}(cst, acp, pdg)$
5   $stl \leftarrow \text{Generate}(cst, root)$
6   $\text{addToMap}(r, acp, stl)$
7 return $r$

2) Overall Three-Step Static Detection Algorithm: The Locating Phase adopts a three-step static detection algorithm to detect and locate fragmentation-induced compatibility 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 stack traces of a potential fragmentation-induced compatibility issue, and constructs a call site tree. The three steps are described detailedly one by one next.

ii. **Validation Step**: in this step, ELEGANT prunes the call site tree using program slicing technique [9] to eliminate false positive stack traces.

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 app under test, with third-party libraries isolated. The algorithm produces as output an issue report containing detailed information for locating the fragmentation-induced compatibility 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 stack traces. In this report, each API-Context pair represents a compatibility issue, and its corresponding stack traces 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 stack traces list $stl$ which contains detailed information for locating the issue represented by $acp$ is generated (Line 5). Finally, the pair $(acp, stl)$ 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).

The three steps are described detailedly 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 app’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 stack trace of this issue, and each stack trace implies that it may lead to this fragmentation-induced compatibility issue. For instance, Figure 1 presents a call site tree, of which the root represent 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 stack traces, and each stack trace implies that the fragmentation-induced compatibility issue related to the issue-inducing API $A$ may 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
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 ← CallSiteTree()
2 root ← CallSiteTreeNode(api)
3 if current context not satisfy ctx then
   4         cst.root ← root
   5         return cst
6 css ← getCallSites(cg, api)
7 callers ← classifyCallSites(css)
8 foreach method m ∈ callers do
   9         csst ← Detect(m, ctx, cg)
10        csst.root.callsites ← callers[m].callsites
11        append(root.children, csst.root)
12 cst.root ← root
13 return cst
```

report generated by our approach provides not only the exact call sites of an issue, but also the full information of its different stack traces, 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 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 f2 of method A_2_1, the data- or 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 ignored. But with call site tree, even they are ignored 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 app under test. 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 current context satisfies ctx is detected. If not satisfied (Line 3), for example, the issue-triggering context is Android version 4.0, whereas the current Android version is 5.0 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). Then all call sites css are classified into different groups according to the methods 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 as 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 attached 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 attached 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 fragmentation-induced compatibility issues, and written compatible code to adapt different context, e.g., an app developer can use Build.VERSION.SDK_INT to check the Android version and write different code to adapt different versions, it is not suitable for us to report all stack traces 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, stack traces 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 analysis, 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 platform versions (a.k.a., 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 platform versions, 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 Pair Model is extended with a new condition called important fields, and it can be formalized in the following context-free grammar:

```
Context → Condition | Condition ∧ Context
Condition → Software_env | Hardware_env | API usage | Important fields
```
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

if acp.important_fields satisfied then
  return cst;
q ← Queue (cst.root)
while q != ∅ do
  n ← dequeue(q)
  foreach child node c ∈ node.children do
    enqueue(q, c)
  foreach call site cs ∈ n.callsites do
    slice ← backwardSlicing(cs, pdg)
    foreach statement s ∈ slice do
      if s can fix this issue then
        delCallSite(n, cs)
        break
  if n.callsites = ∅ then
    while hasNoSiblings(n) and
         cst.root ≠ n.parent do
      n ← n.parent
    delNode(cst, n)
return cst
```

Algorithm 4: Generate

```
input : csn, a call site tree node
output: traces, a list of stack traces of csn

traces ← an empty array
foreach child node c ∈ csn.children do
  sts ← Generate(c)
  foreach subtrace st ∈ sts do
    s ← concat (st, n.method)
    append (traces, s)
return traces
```

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,

**API:** `AlarmManager.set()`

**Context:** `min_API_level = 19 ∧ Important_fields = ["min_API_level"]`

in the Detection Step, the app under test 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 is satisfied, the Validation Step will be skipped.

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 stack trace is fixed. Once a call site is determined fixed by the app developer, its belonged stack traces 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, 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 avoid the fragmentation-induced compatibility issue (Line 11), the call site `cs` is deleted from node `n` (Line 12), hence all the stack traces related to `cs` are in result deleted. Furthermore, once all call sites of node `n` are deleted (Line 14), `n` contributes nothing to the stack traces and therefore is also deleted (Line 17). This deleting process goes along each stack trace 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 it has been eliminated by this step.

5) Generation Step: This step generates the validated stack traces of a call site tree. As aforementioned, each stack trace may lead to its corresponding fragmentation-induced compatibility issue. Algorithm 4 presents the process of this generation step. It takes as input a call site tree node `csn`, and produces as output a list of stack traces of its input `csn`. For `csn`, each of its child node such as `c` is traversed (Line 2), and the stack traces from the leaf nodes up to `c` are generated as `sts` (Line 3). Then each caller node `n`'s (Line 5) corresponding method `n.method` of current node `csn` is concatenated to each recursively generated stack trace `st` (Line 6) such to generate stack traces of `csn` (Line 7). Finally these traces `traces` are returned as the detailed information of this issue (Line 8).

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 in-
is used as a module FlowDroid analysis tool for Android apps. In ELEGANT, FlowDroid flow-, field-, object-sensitive and lifecycle-aware static taint designated sources and sinks. It is designed as a context-, Steven on top of Soot [13], accomplishes it using specifically data flow analysis on it. FlowDroid [12], developed by A. a control flow graph and perform static analyses such as
plain Java. As a result, it is a difficult task to construct
im
phase of the analysis process as a separate module,
e.g., a statistics module, which is extremely loosely coupled,
providing convenience to developers interested in extending
our implementation.
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
characterized 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.

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-sensitive
technique is also available in ELEGANT. To choose an appro-
priate 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 databases LibRadar [14]
and LibD [15] uses 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]. With the help of the Tracker, any types of
messages can be delivered to any components in ELEGANT.
ELEGANT uses the Tracker mainly for emitting the vali-
dated issues. In fact, there are various manners to deliver
issues, e.g., as a direct return value of AbstractFinder.
However, we chose eventually to use the PubSub pattern be-
cause as a frequently used design pattern, it is more extensible
and flexible. Via this pattern, any types of messages besides the
validated issues, e.g., the elapsed time or stats used in analysis,
can be emitted at any time. One component needs only to
register to its interested types of messages and masks all the
others. With the help of the Tracker, we care nothing but
how to deliver the messages, and can extract anything else out
of the main logic of the analysis process as a separate module,
e.g., a statistics module, which is extremely loosely coupled,
providing convenience to developers interested in extending
our implementation.

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 app’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 compatibility issues, validate their call site
trees, and generate the issue report.
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?

### IV. EVALUATION

We selected another 20 real-world Android apps as listed in Table I to evaluate FicFinder [2]. To further demonstrate the effectiveness of ELEGANT, we additionally selected another 20 real-world Android apps to evaluate it. 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].

#### B. RQ1: Effectiveness of ELEGANT

The first experimental results are listed in Table II. The experimental results of each app under test 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 stack traces. 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 app under test. For example, supposing an issue is related to API `Activity.onKeyDown()` whose model is

```plaintext
API: Activity.onKeyDown()
Context: min_API_level = 16 ∧ bad_devices = ["LG", "LGE"]
```

if (1) code that adapt API level less than 16 and greater than 16, and (2) code that adapt LG/LGE/others 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. These symbols and classification principles remain the same in the follow-up experiments and result tables, i.e., experiments for RQ2, RQ3, and Table III to Table V.

The results show that:

i. ELEGANT reported issues 32/133/514, including 27/106/407 true positives and 5/27/107 false positives. The false positive rate is 0.156/0.203/0.208.

ii. FicFinder reported issues 51/196/−, including 24/107/− true positives and 27/89/− false positives. The false positive rate is 0.529/0.454/−.

These results confirm that ELEGANT is more effective than FicFinder: although ELEGANT reports less issues, true positives reported are much more and false positives much less, and the false positive rate 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. We selected another 20 real-world apps as listed in 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 true positives 19/65/163, and false positives 8/16/35. The false positive rate 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 true positives and 10/37/144 false positives. The false positive rate is 0.270/0.259/0.261.

ii. With d3, ELEGANT reported issues 32/133/514, including 27/106/407 true positives and 5/27/107 false positives. The false positive rate 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 false positives: 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 fragmentation-induced compatibility issues, with true positives 8/33 and false positives 23/54. The false positive rate is 0.742/0.621.

ii. With our expanded database, FicFinder reported 51/196 issues, with true positives 24/107, and false positives 27/89. The false positive rate is 0.529/0.454.
### TABLE IV

**Effectiveness of Third-Party Library Isolation**

<table>
<thead>
<tr>
<th>App</th>
<th>Enable d3</th>
<th>Disable d3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ALL</td>
<td>TP</td>
</tr>
<tr>
<td>1Sheeld</td>
<td>3/12/13*</td>
<td>3/12/13</td>
</tr>
<tr>
<td>BankDroid</td>
<td>4/12/20</td>
<td>3/11/13</td>
</tr>
<tr>
<td>AntennaPod</td>
<td>3/16/35</td>
<td>2/14/33</td>
</tr>
<tr>
<td>A.S. Keyboard</td>
<td>1/15/42</td>
<td>1/15/42</td>
</tr>
<tr>
<td>ConnectBot</td>
<td>1/2/3</td>
<td>1/2/3</td>
</tr>
<tr>
<td>Conversations</td>
<td>2/3/42</td>
<td>2/3/42</td>
</tr>
<tr>
<td>IrssiNotifier</td>
<td>1/5/21</td>
<td>1/5/21</td>
</tr>
<tr>
<td>K-9 Mail</td>
<td>5/6/13</td>
<td>5/6/13</td>
</tr>
<tr>
<td>Kore</td>
<td>1/11/30</td>
<td>1/11/30</td>
</tr>
<tr>
<td>Pac.Droid</td>
<td>0/0/0</td>
<td>0/0/0</td>
</tr>
<tr>
<td>QKSMS</td>
<td>2/2/9</td>
<td>2/2/9</td>
</tr>
<tr>
<td>Transdroid</td>
<td>1/4/4</td>
<td>1/4/4</td>
</tr>
<tr>
<td>WordPress</td>
<td>3/20/39</td>
<td>2/6/19</td>
</tr>
<tr>
<td><strong>FP-Rate</strong></td>
<td>0.156/0.203/0.208</td>
<td>0.270/0.259/0.261</td>
</tr>
</tbody>
</table>

* The experimental results of each app under test 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 stack traces.

### TABLE V

**Effectiveness of Expansion**

<table>
<thead>
<tr>
<th>App</th>
<th>Default Expanded</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ALL</td>
</tr>
<tr>
<td>1Sheeld</td>
<td>1/3</td>
</tr>
<tr>
<td>BankDroid</td>
<td>1/1</td>
</tr>
<tr>
<td>AnkiDroid</td>
<td>4/19</td>
</tr>
<tr>
<td>AntennaPod</td>
<td>2/4</td>
</tr>
<tr>
<td>A.S. Keyboard</td>
<td>3/4</td>
</tr>
<tr>
<td>ConnectBot</td>
<td>0/0</td>
</tr>
<tr>
<td>Conversations</td>
<td>3/3</td>
</tr>
<tr>
<td>IrssiNotifier</td>
<td>5/9</td>
</tr>
<tr>
<td>K-9 Mail</td>
<td>2/2</td>
</tr>
<tr>
<td>Kore</td>
<td>1/11</td>
</tr>
<tr>
<td>Pac.Droid</td>
<td>1/2</td>
</tr>
<tr>
<td>QKSMS</td>
<td>3/4</td>
</tr>
<tr>
<td>Transdroid</td>
<td>2/6</td>
</tr>
<tr>
<td>WordPress</td>
<td>3/19</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>31/87</td>
</tr>
<tr>
<td><strong>FP-Rate</strong></td>
<td>0.472/0.621</td>
</tr>
</tbody>
</table>

* The experimental results of each app under test 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.

These results show that the performance of FicFinder has improved:

i. In the reported issues, true positives increase by 200.00% (from 8 to 24)/224.2% (from 33 to 107).

ii. Within the reported issues, only 4/53 are considered as false positives. The false positives rate 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 that have always been ignored by developers, e.g., `Resources.getDrawable()`, `Activity.onKeyDown()`, `ContentValues.put()`.

Hence we get the answer to RQ3.

### V. RELATED WORK

Android fragmentation and compatibility issue 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, D. 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, T. 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, L. 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 fragmentation-induced compatibility 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].

VI. CONCLUSION

In this paper, we presented ELEGANT, an automated two-phase tool to effectively detect and locate fragmentation-induced compatibility issues. Based on our expanded API-Context pair database with 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 the fragmentation-induced compatibility issues in the 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 demonstrate the effectiveness of ELEGANT on detecting and locating these issues for Android apps.

In future, we plan to further improve the API-Context Pair Model to make it more powerful in dealing with semantics related issues. We also plan to propose an automated technique to extract API-Context pairs from various Android compatibility issues.