DDLDroid: Efficiently Detecting Data Loss Issues in Android Apps

Conference Paper · July 2023

DOI: 10.1145/3597926.3598089

Wei Song
Nanjing University of Science and Technology
98 PUBLICATIONS **1,236** CITATIONS

DDLDroid: Efficiently Detecting Data Loss Issues in Android Apps

Yuhao Zhou
School of Computer Science and Engineering
Nanjing University of Science and Technology
Nanjing, China
zhouyuhao423@163.com

Wei Song*
School of Computer Science and Engineering
Nanjing University of Science and Technology
Nanjing, China
wsong@njust.edu.cn

ABSTRACT

Data loss issues in Android apps triggered by activity restart or app relaunch significantly reduce the user experience and undermine the app quality. While data loss detection has received much attention, the state-of-the-art techniques still miss many data loss issues due to the inaccuracy of the static analysis or the low coverage of the dynamic exploration. To this end, we present DDLDroid, a static analysis approach and an open-source tool, to systematically and efficiently detect data loss issues based on the data flow analysis. DDLDroid is bootstrapped by a saving-restoring bipartite graph which correlates variables that need saving to the corresponding variables that need restoring according to their carrier widgets. The missed or broken saving or restoring data flows lead to data loss issues. The experimental evaluation on 66 Android apps demonstrates the effectiveness and efficiency of our approach: DDLDroid successfully detects 302 true data loss issues in 73 minutes, 180 of which are previously unknown.

CCS CONCEPTS

• **Software and its engineering → Software defect analysis**.

KEYWORDS

Android apps, data loss, data flow analysis, bug detection

ACM Reference Format:
Yuhao Zhou and Wei Song. 2023. DDLDroid: Efficiently Detecting Data Loss Issues in Android Apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598089

1 INTRODUCTION

As one of the four fundamental components of Android, *activity* plays a critical role in human-app interaction, because it implements the graphical interfaces which allow users to not only browse the content provided by apps but also input some data to apps [15, 19, 21, 34]. An activity usually consists of widgets, which can be imagined as the views of an app's data and functionality that are accessible right from the user's screen [5]. Due to the event-driven nature of apps, activities of an running app are frequently destructed and then recreated in many scenarios (*e.g.*, screen rotation, system theme switching). During this procedure, if some data relevant to GUI display is not saved and then restored, data loss issues will occur, that is, inconsistent GUI display before and after the activity reconstruction [10, 28]. The data loss issues trigger non-crashing functional bugs [28, 29, 31] and significantly diminish the user experience (*e.g.*, the users have to enter some inputs again that has been entered before), and may even incur more severe outcomes (*e.g.*, app crashes) [1, 12, 16, 20, 22, 24, 26, 33].

There are two scenarios that require the reconstruction of an activity [12]: 1) activity restart and 2) app relaunch. Activity restart is caused by the configuration changes [9] (*e.g.*, screen rotation), which makes the app experience a *pause-stop-destroy-recreate* lifecycle transition. App relaunch is caused by the system-initiated process death [8], which happens when an app is placed in the background (*e.g.*, when a call is answered) but the system cannot keep the app process in memory for a longer time [7, 27]. Since both scenarios bring a series of activity's lifecycle transitions [4], the relevant lifecycle callbacks should contain the code to maintain the interface data appropriately not only in the memory but also in the persistent storage. Although Google provides guides [10] to help avoid data loss issues, developers may inevitably make mistakes due to the complexity of apps.

Data loss issues have received much attention [1, 12, 16, 20, 22, 24, 26, 33]. As two representatives of static analysis work on data loss, KREfinder [26] focuses on finding incorrect data saving and restoring that lead to potential data loss issues, whereas LiveDroid [12] aims to identify the data that need to be saved, and then generate code patches for fixing data loss issues. However, both suffer from the data over-saving problem (thus many false positives), because many variables and GUI properties that do not affect the widget display are also considered to be saved in their approaches. Moreover, both rely on access paths to connect variables that need saving to those that need restoring, which is too restricted. Thus, they also have many false negatives. More work uses dynamic techniques to detect or fix data loss issues [1, 14, 16, 20, 22, 24, 33]. They either insert some events (*e.g.*, screen rotation) in the available test cases [1, 16, 20, 22, 24, 33], or automatically explore the app by executing event sequences to trigger data loss scenarios [14]. Although these dynamic techniques spend much time on exploration, the activity coverage is not high, thus many false negatives.

To address the limitations of existing approaches, we present DDLDroid, a static analysis which can effectively and efficiently detect data loss issues in Android apps. Based on the callbacks of activity lifecycle and those handling user events, DDLDroid first identifies two sets of program variables (the set of variables that

*Corresponding author.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598089

need saving and the set of variables that need restoring). We observe that while the data is macroscopic from the perspective of users (*e.g.*, the *scores* of an counting app; a *popup* for deletion confirmation), the data display depends on some concrete variables in the code. For example, the value of a `String` variable determines what the *scores* shows in the screen. According to this observation, DDLDroid analyzes the widget API invocations, and according to the arguments or the returned value, it then correlates such program variables to the relevant mutable widget properties, avoiding many false positives in data loss detection. The widget properties immutable to users (*e.g.*, the text property of a `TextView` for the title setting is never changed) are not considered by DDLDroid. With this, a variable that needs saving is connected to another variable that needs restoring if they correlates to the same widget property (that is, the value of the former should be saved and then used as input for the latter). Therefore, DDLDroid can report a data loss issue when the data is not completely maintained (either the variable that needs saving or restoring is missing). For the data whose variables that need saving and restoring are both identified, DDLDroid further checks the correctness of the saving and restoring data flows. If either the data flow is broken, a data loss issue is also reported. DDLDroid considers the data flows in the scenarios of activity restart and app relaunch, avoiding many false negatives.

Applied to a dataset of 66 real-world Android apps, DDLDroid totally finds 302 true data loss issues and only 27 false positives, achieving an accuracy of 91.79%. In addition, there exist 108 reported data loss issues that are not confirmed provisionally because we are not familiar with the apps and have difficulties in manually finding the target states for validation. Even if we conservatively categorize all the unconfirmed data loss issues as false positives, DDLDroid can still achieve an accuracy of 69.11%. Compared with the most recent static technique LiveDroid [12], DDLDroid detects 716.32% more real data loss issues. In contrast to the recent dynamic detectors, DDLDroid detects 104.05% and 77.65% more real data loss issues than DLD [24] and iFixDataloss [14] do, respectively. The average analysis time of DDLDroid is only 66 seconds, which is much efficient than iFixDataloss and DLD; the latter often spend much time (*e.g.*, one hour) in exploring the app.

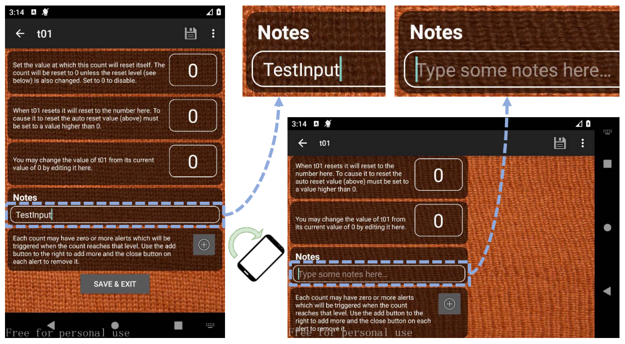
In a nutshell, our contributions are highlighted as follows:

(1) We present DDLDroid, an approach for effectively and efficiently detecting data loss issues in Android apps.
(2) Based on Soot [2] and FlowDroid [3], we implement DDLDroid as a tool and make the tool open-source.
(3) We apply DDLDroid to a dataset of 66 real-world Android apps and the experimental results corroborate the effectiveness and efficiency of DDLDroid.

The remainder of this article is organized as follows. Section 2 employs a real data loss issue to motivate our work. Section 3 presents our approach DDLDroid. Section 4 conducts the experimental evaluation and comparison. Section 5 reviews the related work and Section 6 concludes the paper.

2 A MOTIVATING EXAMPLE

In this section, we motivate our work through a data loss issue in the app *BeeCount*. Figure 1 shows the data loss issue of the widget *enw.project_name* (wrapped with a dashed box in the figure) for



Set the value at which this count will reset itself. The count will be reset to 0 unless the reset level (see below) is also changed. Set to 0 to disable.

When t01 resets it will reset to the number here. To cause it to reset the auto reset value (above) must be set to a value higher than 0.

You may change the value of t01 from its current value of 0 by editing it here.

Each count may have zero or more alerts which will be triggered when the count reaches that level. Use the add button to the right to add more and the close button on each alert to remove it.

Figure 1: The loss of `EditText` in the app *BeeCount*.

```
public class CountOptionsActivity extends AppCompatActivity
    implements OnSharedPreferenceChangeListener {
    EditTitleWidget enw;
+   private String settingNotes;
+   private boolean settingNotesCheck;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_count_options);
        static_widget_area =
            (LinearLayout) findViewById(R.id.static_widget_area);
+       settingNotes =
+           savedInstanceState.getString("settingNotes");
+       if(settingNotes != null) {
+           settingNotesCheck = true;
+       }
    }
    @Override
    protected void onResume() {
        enw = new EditTitleWidget(this,null);
        enw.setProjectName(count.notes);
        enw.setWidgetTitle(getString(R.string.notesHere));
        enw.setHint(getString(R.string.notesHint));
        static_widget_area.addView(enw);
+       if(settingNotesCheck) {
+           enw.setProjectName(settingNotes);
+       }
    }
    @Override
    protected void onSaveInstanceState(Bundle outState) {
+       outState.putString("settingNotes", enw.getProjectName());
    }
}

public class EditTitleWidget extends LinearLayout {
    EditText project_name;
    public EditTitleWidget(Context context, AttributeSet attrs) {
        project_name = (EditText) findViewById(R.id.projectName);
    }
    public void setProjectName(String name) {
        project_name.setText(name);
    }
    public String getProjectName() {
        return project_name.getText().toString();
    }
}
```

Figure 2: Source code excerpt from *BeeCount*.

inputting "Notes" in `CountOptionsActivity` when the user rotates the phone. Figure 2 illustrates the source code related to this data loss issue. It follows from the code that the app does not save the input value of this widget before the activity restarts or the app relaunches. Instead, the app creates a new `EditTitleWidget` at

Line ⑳ and adds it to the screen every time when the user re-enters this activity. Consequently, the previous input is lost, which reduces the user experience and undermines the app quality.

The fix for this data loss issue by the developers is presented with sign "+" in Figure 2 as well (*i.e.*, Lines ④-⑤, ⑫-⑯, ㉕-㉗, and ㉛). Since the callback `onSaveInstanceState()` is invoked by the system before the activity destroys, the input value obtained by `enw.getProjectName()` can be successfully saved into the variable *outState* of `Bundle` type at Line ㉛. When the activity recreates, the saved value is restored at Line ⑫ and then set to the new-created input box *enw* at Line ㉖. With these steps, the previous input for "Notes" is recovered correctly.

3 DDLDROID

In this section, we elaborate on how DDLDroid is designed. Figure 3 presents the workflow of DDLDroid, which consists of four steps. 1) It first extracts necessary files from the APK package. 2) Based on the Jimple code obtained in the first step, it identifies the data variables that need saving and restoring by constructing a bipartite graph. 3) According to the identified data variables, it then constructs the data flows of saving and restoring to determine whether the data is properly handled. 4) Based on the bipartite graph and the data flows of data saving and restoring, it finally outputs the found data loss issues. For each detected data loss issue, DDLDroid also reports the relevant diagnostic information. The following sub-sections discuss these four steps in more details.

3.1 Pretreatment

The pre-processing step is to obtain the Jimple code and resource files required for the subsequent analysis. We decompile the APK file with Soot [2] (FlowDroid [3]) to generate the Jimple code and the call graph of the code. Jimple code is a kind of IR, which is a kind of three-address code that facilitates the analysis of Java programs. The call graph is generated by FlowDroid with its default configured algorithm (*i.e.*, SPARK). The resource files extracted by apktool [32] are helpful in constructing the bipartite graph.

3.2 Saving-Restoring Bipartite Graph Construction

To detect data loss, we first need to identify the interface data that needs saving and restoring. Since the specific content of data display on UI depends on some variables in the code, the crux is to identify such variables. Specifically, we identify the variables that need restoring from the callbacks that initialize UI and the variables that need saving from the callbacks that handle user events and activity destroy, respectively. Such callbacks and their responsibilities are listed in Table 1. Since these two kinds of callbacks occur at different stages of app's lifecycle and they do not interfere with each other, the variables that need restoring and saving constitute two independent sets. Moreover, there is a connection between the two variable sets because the values of some variables that need saving will be assigned to certain variables that need restoring. Thus, the two independent sets of variables and the connection between them consist in a bipartite graph (cf. Definition 1).

Table 1: Callbacks and Their Responsibilities

Callbacks	Responsibility
`onClick()`, `onLongClick()`, `onContextClick()`, `onDrag()`, `onTouch()`, `onHover()`, `onKey()`, `onFocusChange()`, `onShow()`, `onLocationChanged()`, `onSystemUiVisibilityChange()`, `onDismiss()`, `onScrollChange()`, `onOptionsItemSelected()`	user interactions
`onCreate()`, `onRestart()`, `onStart()`, `onResume()`	activity initilization
`onPause()`, `onStop()`, `onDestroy()`	activity destroy

Table 2: Mostly-used Widgets and Their APIs

Widget Type	Superclass or Interface	API
`Text`	`android.widget.TextView`	`setText()`
	`android.widget.EditText`	`setText()` `getText()`
`Dialog`	`android.app.Dialog` `android.app.Dialog$Builder` `android.app.AlertDialog$Builder` `android.support.v7.app.AlertDialog` `android.support.v7.app.AlertDialog$Builder` `android.support.v7.preference.DialogPreference` `androidx.appcompat.app.AlertDialog` `androidx.appcompat.app.AlertDialog$Builder` `android.preference.DialogPreference` `android.content.DialogInterface.On*`	`show()`
`Drawer`	`androidx.drawerlayout.widget.DrawerLayout`	`openDrawer()` `closeDrawer()`

Definition 1 (**Saving-Restoring Bipartite Graph, SRBG**). *Given an app whose variable set is $\mathcal{V}$, a saving-restoring bipartite graph of the app is an undirected bipartite graph $SRBG = (V, E)$ where:*

(1) *$V = S \cup R$ is the vertex set such that $S = \{s_i | \ s_i \in \mathcal{V}\}$ is the set of variables that need saving and $R = \{r_j | \ r_j \in \mathcal{V}\}$ is the set of variables that need restoring, and $S \cap R = \emptyset$.*

(2) *$E = \{(s_i, r_j) | \ s_i \in S, r_j \in R\}$ is the edge set such that for $\forall$ $(s_i, r_j) \in E$, it represents that the value of the variable s_i should be saved and then passed to the variable r_j for restoring.*

In the following part, we discuss how to obtain the bipartite graph, to facilitate data loss detection.

Observation 1. *The mutable data is displayed and updated via specific widget API invocations, and a data loss issue corresponds to the inconsistent display of a widget before and after the activity restart or app relaunch.*

Vertex determination. According to Observation 1, the existence of a variable that needs saving or restoring in an activity depends on whether there exists an invocation to a specific API of a widget (cf. Table 2). Such APIs are responsible for opening or closing widgets (*i.e.*, dialogs, drawers), and for setting or getting widget (*i.e.*, texts) properties. If the invocation is in the callback that deals with activity initialization, it refers to an operation of UI recovery, during which the corresponding variable should be restored. If the invocation is in the callback that handles user-interaction event or activity destroy, it indicates that the UI will be modified and the corresponding variable should be saved.

Algorithm 1 outlines how DDLDroid obtains R and S based on the call graph. For each activity, DDLDroid extracts the callbacks of different categories (Lines ③-⑥) from the code and the layout

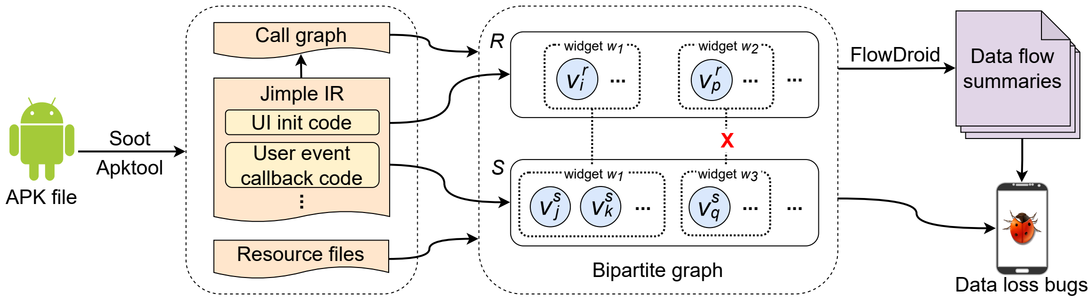


Figure 3: Workflow of DDLDroid.

Algorithm 1: Determine variable sets R and S

```
   Input: Activity set $ACT$, specific widget API set $W$ in Table 2
   Output: Variable set $R$, variable set $S$
1  $R \leftarrow \emptyset, S \leftarrow \emptyset$
2  for each activity $act \in ACT$ do
3      $lr \leftarrow getLayoutResources(act)$
4      $CU \leftarrow getCallbacksOfUserInteraction(act, lr)$
5      $CI \leftarrow getCallbacksOfActivityInitialization(act)$
6      $CD \leftarrow getCallbacksOfActivityDestroy(act)$
7      for each callback $c \in CI$ do
           // follow the lifecycle order: from onCreate() to onResume()
8          $R' \leftarrow DFS\_search(c)$
9          $R \leftarrow R \cup R'$
10     for each callback $c \in (CU \cup CD)$ do
           // follow the lifecycle order: from onPause() to onDestroy()
11         $S' \leftarrow DFS\_search(c)$
12         $S \leftarrow S \cup S'$
13 return $R, S$
14 Function $DFS\_search(f)$
15 $V \leftarrow \emptyset$, $body \leftarrow getMethodBody(f)$
16 for each statement $s \in body$ involving invocation do
17     $callee \leftarrow getCallee(s)$
18     if $isWidgetAPI(callee, W)$ then
19         $v_{wid} \leftarrow identifyVariableFromContext(s)$
           // $v_{wid}$ denotes a variable $v$ whose carrier widget is $wid$
20         $V \leftarrow append(V, v_{wid})$
21     else
22         $V \leftarrow V \cup DFS\_search(callee)$
23 return $V$
```

file. It then searches for the variables that need restoring (Lines ⑦-⑨). For each callback c that handles UI initialization (*i.e.*, restoring data and then recovering app state as the same before destruction), DDLDroid traverses its code. If an invocation s of a relevant widget API in Table 2 is located, it extracts the variable that needs restoring from the arguments or the returned value of s, or from the condition variable in the `if` statement which controls s (Lines ⑱-⑳). Otherwise, the traversal jumps to the callee, and jumps back to go on traversing the caller when the traversal of callee ends (Lines ㉑-㉒). The identification of variables that need saving is similar (Lines ⑩-⑫). The only difference is that we start searching for variables that need saving from user-interaction callbacks and system callbacks that deal with activity destroy. These callbacks are responsible for dealing with the modified mutable data (*i.e.*, updating mutable data according to user interactions and saving data before activity destroy).

Example 1 (Continuation of the motivating example in Section 2). *In the fixed version, when DDLDroid encounters the*

```
1  public class EditHostActivity extends ActionBarActivity {
2      public static final String KEY_SHOW_CANCEL_DIALOG
3          = "showCancelDialog";
4      @Override
5      protected void onCreate(Bundle savedInstanceState) {
6          super.onCreate(savedInstanceState);
7          setContentView(R.layout.host_edit);
8          if(savedInstanceState != null) {
9             if(savedInstanceState.getBoolean(KEY_SHOW_CANCEL_DIALOG))
10               showCancelDialog();
11         }
12     }
13     @Override
14     protected void onSaveInstanceState(Bundle outState) {
15         outState.putBoolean(KEY_SHOW_CANCEL_DIALOG,
16            (cancelDialog != null && cancelDialog.isShowing()));
17     }
18     private void showCancelDialog() {
19         // settings for cancelDialog;
20         cancelDialog.show();
21     }
22 }
```

Figure 4: Source code excerpt from *Port Knocker*.

invocation of `setProjectName()` *at Line ㉖, it turns to traverse the callee function and identifies the call site that invokes* `setText()` *at Line ㊵. From this API invocation, the local variable (i.e.,* name*) that needs restoring is found and put into R. Additionally, when DDLDroid encounters the dialog API invocation at Line ⑳ in Figure 4, from the statement (Line ⑨) on which Line ⑩ depends, an anonymous variable that needs restoring is also identified and put into R.*

Edge determination. We create an edge between two variables, $s_i \in S$ and $r_j \in R$, when the value of s_i should be saved and then used for restoring r_j. However, determining the edge set E of a SRBG suffers from the following two challenges:

(1) *Heterogeneity.* Since s_i and r_j may be different local variables in various callbacks, their names are usually heterogeneous such that we cannot determine an edge by simply judging whether their names (access paths) are the same.
(2) *Many-to-many relationship.* Since we may extract many variables from different callbacks and the values of some variables in S can be used for restoring some variables in R, the relationship between R and S is m: n (cf. Figure 5(a)).

Observation 2. *Each variable that needs restoring or saving has a widget as its carrier for data display.*

According to Observation 2, instead of matching variables based on the access paths (like `this.account.user.addr`) as adopted

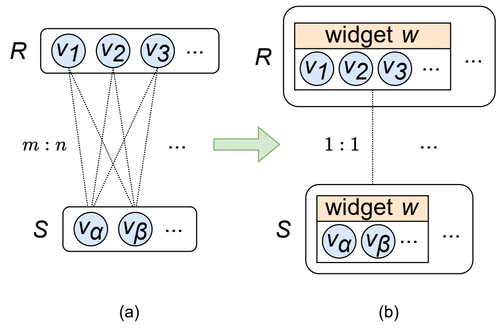


Figure 5: Partitioning variable sets and correlating variable partitions according to carrier widgets.

in [12], we determine an edge if the two variables are associated to the same widget. For each variable, we use an attribute tuple (*type*, *feature*) to identify its carrier (Line ⑲ in Algorithm 1). The attribute *type* refers to the type of the widget (*e.g.*, `TextView`, `Dialog`, `DrawerLayout`). The attribute *feature* is used to distinguish different widgets of the same type. There are two categories of *feature*s:

(1) The first category is the corresponding activity field variable for the widget. When an activity creates, `findViewById()` is invoked such that one field variable is bound to a unique widget in the whole lifecycle of the activity. Thus, if two widgets have the same field variable, they are the same widget.
(2) The second category is a series of widget-setting statements for the widget. For example, a `Dialog` is created if a series of setting APIs (the `set*()` methods of the dialog classes listed in Table 2) are invoked. We extract all these setting statements from the code where the widget is created. If all settings of two widgets are the same, they are regarded to be the same widget.

Observation 3. *Each widget has one property for content display that is related to all variables whose carriers are the widget, and saving all mutable program variables and widget properties may lead to over-saving, because many of them are irrelevant to UI display.*

According to Observation 3, saving either the corresponding program variables or the related widget property suffices for data restoring. Since those irrelevant program variables and widget properties are excluded by DDLDroid, and thus, many false positives can be avoided.

Observation 4. *The vertexes in S fall into some sub-sets based on their carrier widgets, and so does R. Then, edges can only exist between a sub-set of S and a sub-set of R, where the variables in the two sub-sets have the same carrier widget.*

Based on Observation 4, we significantly reduce the connections between vertexes in R and S by clustering the variables into some sub-sets, as illustrated in Figure 5(b). For example, for a widget with m variables that need restoring and n variables that need saving, if we do not group them into two sub-sets in R and S, respectively, it then needs to analyze the correctness of restoring and saving operations for $m \times n$ times. However, after grouping variables, it only needs to analyze the correctness of restoring and saving operations for $m + n$ times. In our implementation, a cluster corresponds to a map where the KEY is the attribute tuple (*type*, *feature*) that uniquely represents a widget, and the VALUE is a set of variables that need restoring or saving, whose carrier is the widget.

Table 3: All Classes Related to Storage

Class	Storage
`Androidx.lifecycle.ViewModel`	memory
`Android.content.SharedPreferences`	disk
`Android.os.Bundle`	disk
`Android.database.sqlite.SQLiteDatabase`	disk
`*OutputStream`, `*InputStream`	disk
`*Writer`, `*Reader`	disk

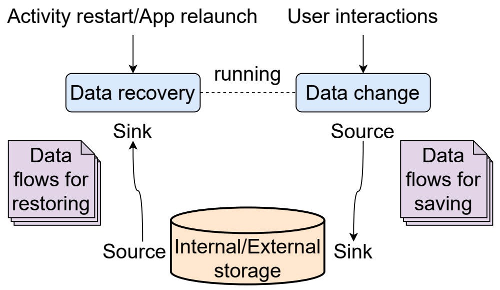


Figure 6: Data flows of saving and restoring.

Example 2 (Continuation of Example 1). *After the fix, DDLDroid identifies the carrier widgets of the variable (i.e., name) that needs restoring and the variable (anonymous at Line ㊸) that needs saving, respectively. The carriers of these two variables are both identified as the widget enw.project_name for entering "Notes". Therefore, these two variables are put into two sub-sets of R and S, respectively.*

3.3 Data Flow Analysis

In this part, we describe how to verify whether the variables in S and R are correctly saved and restored, respectively.

The correctness of saving and restoring depends on whether the variables are properly stored to and loaded from the storage spaces (cf. Table 3), respectively. Since manually enumerating all the storage-related classes from Android libraries seems infeasible, Table 3 only lists the classes that are summarized in the work of `KREFinder` [26] and supplements `ViewModel`. Specifically, the correct saving refers to the situation that the latest values of variables in S can flow into the saving statements which store the values to these storage spaces. The correct restoring refers to the situation that the latest values are loaded back from these storage spaces and assigned to the variables in R during activity initialization. Therefore, we can verify the correctness of saving and restoring with taint analysis, as illustrated in Figure 6.

Our taint analysis is inherited from FlowDroid [3], which is an open-source static taint analysis tool for Android apps to construct the data flow from a tainted *source* to a *sink*. Since constructing data flows of saving and restoring requires the capability of tracing flows between arbitrary statements, there are two ways to define the sources and sinks:

(1) Statement-based definition. We use `StatementSourceSinkDefinition` to define the sources and sinks for both the statements of initializing variables in activity initialization and the statements of updating variables in user interactions.
(2) Method-based definition. We use `MethodSourceSinkDefinition` to define the sources and sinks for the APIs of storage-related classes, the invocations of which refer to the saving and reading operations.

We discuss how to define the sources and sinks for the data flows of saving and restoring, respectively. Initially, for the data flows of restoring, the APIs of the classes listed in Table 3 that read data from the storage spaces are defined as method-based sources. Then, we locate the corresponding assignment statements of variables in R from the callbacks of activity initialization, and define them as statement-based sinks. Therefore, the data flow from such a source to such a sink indicates that the previous value can be loaded from the storage spaces and then used to restore a variable in R. Similarly, for the data flows of saving, the corresponding assignment statements of variables in S in the callbacks of handling user events and activity destroy are defined as statement-based sources, and the APIs of the classes listed in Table 3 that store data are defined as method-based sinks. Therefore, the data flow from such a source to such a sink indicates that the assigned new value of a variable in S can flow to a call site that saves data. In this way, we can analyze whether the variable values come from the expected storage spaces and whether the changed variable values can flow to the expected storage spaces.

`FlowDroid` [3] may miss some data flows of saving and restoring due to incomplete taint wrappers during backward propagation. The taint wrappers aim at providing simple rules to obtain the results of taint propagation at the call sites of library APIs without performing analysis in libraries. Although `FlowDroid` can handle most cases of taint wrappers during forward propagation, it does not provide these wrappers during backward propagation. Hence, based on the same rules that are employed during forward propagation, we add the implementation of `getAliasesForMethod()` to obtain aliases of taints when encountering the call sites of some library APIs (*e.g.*, `String.valueOf()`) during backward propagation.

Example 3 (Continuation of Example 2). *In the fixed version, both the data flow of restoring (i.e., Line ⑫⟹ ⑭⟹ ㉖⟹ ㊵) and the data flow of saving (i.e., Line ㊸⟹ ㉛) are constructed by DDLDroid. The former flow indicates that the previous input of "Notes" is restored correctly from* `Bundle` *and the latter flow indicates that the existing input can be saved correctly into* `Bundle`*. Thus, the data loss issue is no longer exists in the fixed version.*

3.4 Data Loss Report

Since data loss occurs during activity restart or app relaunch, we reveal loss issues in these two scenarios, respectively.

Observation 5. *All the internal/external storage spaces listed in Table 3 are not cleaned by the system during activity restart.*

Data loss during activity restart. According to Observation 5, regardless of whether the data is maintained by the `ViewModel` in memory or the other storage spaces in disk, a data loss issue can be determined if either of the following two situations occurs:

(1) Incomplete saving or restoring. For a variable in R, if there is no corresponding variable in S, then there must be a data loss, and vice versa. Thus, we report data loss issues due to incomplete data management based on the SRBG. Since the SRBG correlates the variables that need saving and restoring, a mismatched variable indicates a data loss.
(2) Incorrect saving or restoring. Even for the data that is managed by the saving and restoring operations, it may still be lost if the saving or restoring operations are not correctly implemented. Thus, we report a data loss issue if either of the data flows of saving and restoring is broken.

Observation 6. *The instances of* `ViewModel` *are in memory, which are also cleaned by the system during app relaunch, resulting in the data saved in them cleaned.*

Data loss during app relaunch. Based on Observation 6, even the data not lost during activity restart may be lost during app relaunch if it is maintained by `ViewModel`.

For the scenario of app relaunch, we reveal loss issues by checking whether the data can be saved to and read from the external storage. In practice, developer can maintain the data directly in the disk or use `ViewModel` as the bridge. If it is the former case, a data loss during activity restart indicates that the data is not saved to and restored from the disk. If it is the later case, a data loss during activity restart indicates that the data is not correctly maintained by `ViewModel`, and thus, the data will not be correctly saved to or read from the disk as well. In summary, no matter which means is used, the data lost during activity restart must also be lost during app relaunch. Next, we check the data that is maintained by `ViewModel` and not lost during activity restart. The data maintained by `ViewModel` is saved in certain fields in its instances. If there is such a field not saved to the disk or not read from the disk in the constructor of `ViewModel`, then the data maintained by the field will be lost during app relaunch.

Example 4 (Continuation of Example 3). *Since only one variable (i.e., name) that needs restoring is identified at Line ㊵ (reachable from Line ㉑) in the buggy version, DDLDroid reports it as an incomplete saving issue that leads to the loss of the previous input for "Notes". After the fix, both the variable that needs restoring and the variable that needs saving, as well as the corresponding saving and restoring data flows are identified by DDLDroid (as illustrated in Examples 2 and 3, respectively), which indicates that the widget enw.project_name for entering "Notes" no longer suffers from data loss.*

4 EVALUATION

In this section, we evaluate DDLDroid and compare it with the state-of-the-art data loss detectors. Our experimental evaluation aims to answer the following research questions (RQs):

- **RQ1**: How effective is DDLDroid in detecting data loss issues in Android apps?
- **RQ2**: How does the detection capability of DDLDroid compare to those of the state-of-the-art techniques?
- **RQ3**: How efficient and scalable is DDLDroid?

- **RQ4**: Are data loss issues severe in Android apps? How do developers react to the presence of data loss issues?

4.1 Experimental Setup

Comparison tools. We compare DDLDroid with LiveDroid [12], DLD [24], and iFixDataloss [14]. LiveDroid is the most recent static analyzer to reveal the data that need saving and restoring. DLD and iFixDataloss are the two state-of-the-art dynamic data loss detectors. Although KREFinder [26] is also relevant to data loss issues, it is not considered because it only analyzes the saving and restoring of all mutable fields but does not automatically flag the affected GUI widgets. To report the affected GUI widgets, KREFinder needs lots of manual work to reconstruct the triggering method invocation chains of data loss issues to confirm them.

Dataset. We evaluate DDLDroid on a set of 66 apps used in the work of iFixDataloss [14], which covers 45 apps used in the work of DLD [24], 17 apps used in the work of LiveDroid [12], and four commercial apps used in the work of iFixDataloss [14]. We download these apps from the public dataset of iFixDataloss, and perform DDLDroid and the state-of-the-art techniques on these apps for comparison. Notably, whether the presence or absence of data loss issues in this dataset is unknown in advance. We only know some detected data loss issues by the state-of-the-art tools reported in the work of iFixDataloss.

Bug confirmation. We verify the correctness of the generated results by manually validating the reported data loss issues on an Android phone. Since DDLDroid also outputs the locations of the affected GUI widgets, we can manually run the app and guide it to the designated UI state to validate the reported data loss issues in the presence of activity restart or app relaunch. Screen rotation and system theme switch (two typical configuration changes) are used to trigger activity restart. As to app relaunch, the setting of *Background process limit* in the developer options is set to “no background processes”. Consequently, the Android system will destroy the process of the tested app when we navigate to another app. When we navigate back to the tested app, the app relaunch is simulated successfully. Finally, we compare the two screenshots before and after activity restart or app relaunch to validate the data loss issue. If the app state differs in the two screenshots, the reported issue is a true positive; otherwise, it is a false positive. The manual validation also applies to the output of the competitors.

Experimental environment. Our experiment is performed on a 64-bit Windows 10 computer with 11th Gen Intel(R) Core(TM) i7-11700 @ 2.50GHz CPU and 64GB memory. The Android phone is an emulator configured with 4 processors, 8GB RAM, and Android 10. We use the default configuration for each tool. In line with the evaluation in the work of iFixDataloss, for DLD and iFixDataloss, the timeout limit of testing each app is set to be one hour.

4.2 RQ1: Effectiveness

The first three columns show the basic information (*i.e.*, app name, lines of code, and version) of these apps. The column “DDLDroid” contains five sub-columns. The sub-column “T” lists the total number of data loss issues found by DDLDroid. The sub-columns “#TP” and “#FP” report the number of true positives and false positives, respectively. The sub-column “#UC” lists the number of data loss

Table 4: Experimental Results on 66 Apps

APP	LoC	Version	DDLDroid					LiveDroid				DLD	iFix∗
			T	#TP	#FP	#UC	Time(s)	T	#TP	#FP	Time(s)	T	T
androidclient	16.8K	0.9.3	4	**2**	2	0	**35**	4	**2**	2	65	0	0
AntennaPod	17.4K	1.6.0.9	11	**11**	0	0	**29**	-	-	-	-	3	6
arXiv mobile	6.7K	2.0.27	0	0	0	0	**9**	9	3	6	12	1	**6**
BeeCount	5.9K	2.4.7	12	**10**	1	1	**7**	3	1	2	9	1	1
Book∗	70.7K	5.2.0	16	**7**	1	8	**19**	4	2	2	81	0	5
Browser	16.9K	8.9.1	6	**6**	0	0	**8**	20	0	20	21	4	1
Calendar∗	5.6K	3.14.159	14	**1**	0	13	**309**	-	-	-	-	0	-
Conversations	74.0K	1.23.3	33	**11**	0	22	**28**	-	-	-	-	0	**11**
CycleStreets	24.7K	3.5	8	3	1	4	269	166	**5**	161	**85**	3	0
Diary	3.2K	1.5	6	4	0	2	**5**	6	1	5	11	**5**	0
DNS66	3.7K	0.4.1	3	**1**	0	2	**5**	1	**1**	0	16	**1**	**1**
Document∗	0.6K	2.7.9	0	0	0	0	**6**	-	-	-	-	**2**	0
Droidshows	12.1K	7.9.7	8	5	0	3	**4**	6	1	5	11	**11**	2
Easy xkcd	16.3K	6.0.4	17	**16**	1	0	**10**	-	-	-	-	6	2
Etar Calendar	0.6K	1.0.10	2	0	0	2	**6**	0	0	0	8	**11**	0
Firefox Focus	21.2K	4	0	0	0	0	**3**	-	-	-	-	0	0
Flym	9.9K	1.4.0	4	2	0	2	**3**	5	1	4	10	**7**	3
Gadgetbridge	68.8K	0.25.1	6	**6**	0	0	319	2	0	2	**232**	2	-
Gitclub	20.8K	1	1	1	0	0	**13**	-	-	-	-	**3**	0
Glucosio	5.0K	1.4.0	15	**15**	0	0	**317**	-	-	-	-	0	0
Gnucash	37.2K	2.4beta3	9	**9**	0	0	**325**	-	-	-	-	0	5
Hourglass	0.8K	1.9	1	1	0	0	**5**	18	**2**	16	**5**	1	1
K9	110.6K	5.6	11	1	1	9	489	13	**9**	4	**85**	1	4
LeafPic	21.0K	0.6beta1	12	**8**	0	4	**31**	2	0	2	350	0	-
Loop Habit∗	22.9K	1.6.2	0	0	0	0	**9**	-	-	-	-	0	0
MALP	23.7K	1.1.0	2	2	0	0	**10**	1	0	1	14	**5**	4
MGit	7.2K	1.4.0	3	3	0	0	**7**	3	0	3	16	0	**6**
MTG Familiar	37.0K	3.5.5	19	0	5	14	**14**	2	0	2	31	14	**25**
MyDiary	17.4K	0.3.0	0	0	0	0	**14**	2	**2**	0	34	0	0
Notes	7.1K	1.0.2	8	**8**	0	0	**13**	1	1	0	16	1	5
OInotepad	6.5K	1.5.4	2	**2**	0	0	**6**	0	0	0	8	0	0
Omni Notes	23.6K	5.4.3	15	**12**	3	0	**11**	1	0	1	44	0	0
OpenTasks	22.1K	1.1.13	2	1	1	0	**17**	1	1	0	22	**3**	0
OpenVPN∗	25.9K	0.7.5	3	2	0	1	**14**	0	0	0	18	2	**10**
PassAndroid	15.1K	3.3.3	6	**4**	0	2	**18**	-	-	-	-	2	2
Periodic Table	3.6K	1.1.1	0	0	0	0	**4**	0	0	0	24	0	**1**
PGPClipper	3.2K	0.22	2	**2**	0	0	**11**	0	0	0	15	0	-
Port Knocker	3.7K	1.0.8	2	1	1	0	**13**	-	-	-	-	0	**8**
Prayer Times	34.2K	3.6.6	3	**3**	0	0	**24**	-	-	-	-	0	0
Pro Expense	6.2K	1.0beta5	0	0	0	0	**310**	-	-	-	-	**1**	0
QRStream	2.4K	1.1.4	3	3	0	0	**4**	2	0	2	89	**6**	0
QuasselDroid	15.2K	0.11.6	6	**6**	0	0	**21**	2	2	0	45	0	0
QuickLyric	16.8K	2.1	6	6	0	0	**9**	2	0	2	49	2	**8**
Remembeer	5.1K	1.3.0	2	**1**	0	1	**9**	2	**1**	1	17	0	0
RingDroid	7.8K	2.7.4	6	**6**	0	0	**5**	1	1	0	7	5	1
Rumble	26.1K	1.0.2	7	**7**	0	0	48	2	0	2	**39**	1	0
Simple Draw	3.7K	3.1.5	0	0	0	0	**10**	0	0	0	19	**4**	1
Simple File∗	7.1K	3.2.0	0	0	0	0	**5**	-	-	-	-	**4**	1
Simple Gallery	11.3K	1.5	0	0	0	0	**9**	0	0	0	21	0	**5**
Simple Soli∗	8.9K	2.0.1	1	0	1	0	**7**	3	0	3	11	**4**	-
Simpletask	27.6K	10.1.13	16	**14**	1	1	**17**	-	-	-	-	5	4
SMS Backup∗	12.7K	1.5.11	0	0	0	0	306	0	0	0	**103**	**4**	0
Syncthing	10.0K	0.9.5	17	**15**	0	2	**11**	0	0	0	31	0	0
TapeMeasure	5.8K	1.0.3	0	0	0	0	**8**	0	0	0	14	0	-
Taskbar	13.3K	3.0.3	17	**14**	0	3	**10**	0	0	0	16	0	0
Tasks	47.6K	6.0.6	8	3	0	5	**22**	-	-	-	-	11	**15**
Tickmate	6.8K	1.4.6	10	**9**	0	1	35	0	0	0	**14**	0	1
Timesheet	2.8K	1.5	8	**8**	0	0	**5**	0	0	0	**5**	0	0
Tuner	7.5K	1.33	0	0	0	0	**5**	0	0	0	15	0	**4**
Tusky	9.9K	1.0.3	4	**2**	0	2	310	1	1	0	**69**	0	1
Twidere	247.2K	3.7.3	1	0	1	0	**83**	-	-	-	-	0	0
Vespucci	100.3K	10.2	30	**24**	2	4	**33**	-	-	-	-	0	-
Vlille Checker	4.7K	4.4.0	0	0	0	0	**7**	1	0	1	23	**4**	-
Webmon	16.6K	2.6.0	10	6	4	0	325	0	0	0	**25**	2	**8**
WiFiAnalyzer	9.8K	1.9.2	13	**13**	0	0	**6**	1	0	1	12	6	11
World Clock∗	3.9K	1.8.6	6	**5**	1	0	**309**	-	-	-	-	0	1
Sum			437	**302**	27	108	4368	287	37	250	**1867**	148	170

issues that have not been confirmed at this stage. Since we validate the detected data loss issues manually based on their locations reported by DDLDroid, we suffer from unconfirmed issues if we cannot direct the app to the targeted activity state. As we are not familiar with all the functions of such apps, unconfirmed data loss issues are inevitable in the limited time. Fortunately, compared to the exploration executed by the dynamic detectors, more data loss issues, especially those in intricate usage scenarios, can be reproduced with the help of the source code of these apps.

DDLDroid reports a total of 437 data loss issues in these 66 apps. We have confirmed 302 true positives, and the screenshots of these true positives are also available with our artifacts. The details on how to reproduce these data loss issues are also available in every result report. The remaining 135 reported issues consist of 27 false positives and 108 unconfirmed issues. It is worth mentioning that the unconfirmed data loss issues are not necessarily false positives. According to the ratio (*i.e.*, 27/302 = 8.94%) of false positives to true positives in our validated issues, we believe that most (at least a large proportion) of the unconfirmed issues are true positives.

False positives. DDLDroid is generally accurate because only 27 false positives are reported. Considering the number of true positives and false positives, DDLDroid achieves an accuracy of 302/329 = 91.79%. Even when all the 108 unconfirmed reports are all false positives, the accuracy is 302/437 = 69.11%. Conservatively, the accuracy of DDLDroid ranges from 69.11% to 91.79%. The false positives are due to the following reasons.

(1) Imprecise widget extraction. Since only the mutable GUI widgets may suffer from data loss issues, DDLDroid is equipped with a strict strategy to extract the widgets for the further analysis of data saving and restoring. However, some widgets are maintained by the Android system, and thus DDLDroid does not find their saving or restoring operations. For example, in the app *World Clock*∗, the input words in the search bar are not lost because they can be maintained by a fragment back-stack [6], which is an internal stack that holds the data of fragment instances. Therefore, the Android system can steer the user to the previous state based on the saved data in this stack. We argue that a lightweight analysis in the Android libraries may alleviate these false positives.
(2) Imprecise taint analysis. Due to the imprecision of the taint analysis, some data flows of saving or restoring are not detected, leading DDLDroid to report incorrect saving or restoring. For example, in the app *Port Knocker*, DDLDroid successfully extracts the variables that need saving and restoring for a dialog. However, FlowDroid misses the data flows of saving and restoring the corresponding variables of this dialog, because of the inadequacy of the alias analysis used in Soot and the imprecision of callback invocation modeling in FlowDroid. These false positives can be mitigated by improving of the data flow analysis as we do in Section 3.3.

Unconfirmed issues. The number of unconfirmed data loss issues is 108. The reasons why we fail to confirm these data loss issues are as follows.

(1) Inaccessible functionality. There are many reasons for this, say, some activities can only be accessed in specific regions, and some app services are only provided to permitted users. For example, the app *Tusky*∗ requires an account of specific category to login and then use it. However, the application for such an account needs the approval of the vendor or the related organization. Unfortunately, our account application has not been approved yet.
(2) Difficultly-triggered methods. Not only the locations of the affected GUI widgets but also the method invocation chain are reported by DDLDroid. However, some methods can only be triggered in a complex scenario. Since we are not familiar with these apps, we find difficulties in meeting the conditions on the method invocation chain. For example, in the app *LeafPic*, there exists an unconfirmed data loss issue that is reported as missing restoring. This issue is a dialog disappearance, and the method invocation chain of opening this dialog contains an invocation of the method `setDrawerTheme()`. Unfortunately, we fail to trigger this method invocation from the UI, and thus, we cannot open the dialog to check whether it could disappear after activity restart or app relaunch.
(3) Unknown UI. For example, in the app *Flym*, two unconfirmed data loss issues of dialog disappearances are both in a `net.fred.feedex.fragment.c` fragment. Although DDLDroid reports the method invocation chains of opening these two dialogs, we cannot know which activity contains this fragment according to its class name, and hence, fail to reach the expected state.

Answer to RQ1: For the 437 data loss issues detected from the 66 apps, 302 are true positives, 27 are false positives, and the remaining 108 have not been confirmed yet. Thereby, the accuracy of DDLDroid ranges from 69.11% to 91.79%.
Implications: DDLDroid is effective in detecting data loss issues and achieves a high true positive rate.

4.3 RQ2: Comparison with the State of the Art

Table 4 also lists the detection results of LiveDroid [12], DLD [24], and iFixDataloss [14] on the 66 apps. Since the results of DLD [24] are screenshots, we filter out the duplicate data loss issues by comparing these screenshots and get the total number of true data loss issues. Based on the output information of LiveDroid and iFixDataloss, we also manually validate their results on a phone. It is worth mentioning that LiveDroid fails to analyze 20 apps due to certain runtime exceptions and iFixDataloss fails to statically construct the GUI models of eight apps. Additionally, although we use the latest version of iFixDataloss on its website, its dynamic exploration module meets some errors when testing 13 apps such that 73 data loss issues reported by it cannot be reproduced in our evaluation.

DDLDroid detects the most number (302) of true data loss issues, followed by iFixDataloss (170), DLD (148), and lastly LiveDroid (37). Regarding false positives, DDLDroid only reports 27, LiveDroid reports 250, and the other two dynamic detectors report none. Additionally, we also analyze the intersections of true data loss issues detected by DDLDroid and the three competitors. As shown in Figure 7, DDLDroid can find 14 of 37 issues detected by LiveDroid, 8

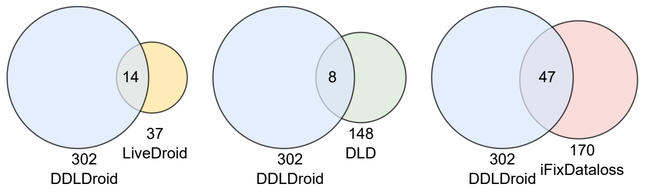


Figure 7: The intersections of the detected true data loss issues between different techniques.

of 148 issues detected by DLD, and 47 of 170 issues detected by iFixDataloss. Overall, among the 302 true data loss issues reported by DDLDroid, 253 cannot be found by the other three tools, and 180 of them are previously unknown (the rest 73 are reported in [14], but we cannot reproduce.). These results demonstrate the usefulness and advantage of our approach.

False negatives. Although DDLDroid performs better in terms of the number of true positives, it miss 220 data loss issues. This is not surprising, because the techniques involved are very different: DDLDroid is a static detector that does not use the liveness and may-modify analysis used in LiveDroid [12], whereas the other two are dynamic detectors. We argue that the false negatives are mostly due to the following reasons.

(1) Incomplete call graph. Since DDLDroid determines *R* and *S* based on the call graph, the incomplete call graph makes DDLDroid miss some methods that handle UI initialization, user interaction events, and activity destroy. For example, many invocations are `InterfaceInvoke` type in the code, and DDLDroid inquires the call graph to get the corresponding "INTERFACE edges" pointing to the callees. However, the call graph sometimes misses such edges, and thus DDLDroid aborts the analysis of these interface methods.
(2) Obfuscated code. The code obfuscation technique is widely used in the real-world Android apps, which introduces challenges for static analysis. For example, in the app *ProExpense*, two data loss issues in `MainActivity` are missed because `onOptionsItemSelected()` is obfuscated by the identifier renaming technique and thus DDLDroid fails to find and then analyze it effectively.

Answer to RQ2: DDLDroid finds 716.21%, 104.05%, and 77.65% more true data loss issues than LiveDroid, DLD, and iFixDataloss do, respectively. Moreover, DDLDroid detects 180 true data loss issues which are previously unknown.
Implications: DDLDroid detects many data loss issues that are missed by the state-of-the-art techniques, thus complementing the existing work to a large extent.

4.4 RQ3: Efficiency and Scalability

Since FlowDroid needs a lot of time for its analysis or even does not stop when it analyzes some very large apps, we set the time limit of data flow analysis to be five minutes in DDLDroid to achieve a trade-off between scalability and effectiveness. On the one hand, although we have attempted to analyze those apps that are trapped in data flow analysis for longer time (or no time limit), FlowDroid does not stop or the number of newly-detected data loss issues is very slim. On the other hand, since the largest analysis time of those apps that are not trapped in data flow analysis is only 269 seconds, which is within the time limit, we argue that our setting of time limit is reasonable.

DDLDroid spends totally 4,368 seconds in analyzing the 66 apps. The time cost on these apps ranges from three to 489 seconds. The average time cost is 66 seconds while the median time cost is only 11 seconds. It follows from Table 4 that DDLDroid mostly finishes the analysis within one minute, and that only a small number of runs (10/66 = 15.15%) are beyond 300 seconds due to reaching the time limit. As to LiveDroid [12], it spends totally 1,867 seconds in analyzing 46 apps, and the average time cost is 41 seconds. On those 46 apps, the average time cost of DDLDroid is 54 seconds, which is 13 seconds longer than that of LiveDroid. However, DDLDroid can analyze the other 20 apps that LiveDroid fails. The running time of DLD and iFixDataloss is on average one hour per app.

Confirmation cost. In case of multiple issues reported from one app, the time cost for triaging subsequent reports is getting lower due to the initial learning curve. Although we are not familiar with these apps, the manual analysis of validation typically takes five to ten minutes per data loss issue, which suggests that if the developers can apply DDLDroid on their apps, the confirmation and fixing of the data loss issues may be more efficient.

Answer to RQ3: iFixDataloss [14] and DLD [24] spend on average one hour per app in guaranteeing their effectiveness, whereas DDLDroid saves 98% time, which consumes on average only 66 seconds per app. On the 46 apps that LiveDroid successfully analyzes, DDLDroid only costs on average 13 seconds longer than LiveDroid [12] does per app.
Implications: DDLDroid scales to large apps owing to the trade-off between efficiency and effectiveness. DDLDroid is much more efficient than the state-of-the-art dynamic techniques. While the state-of-the-art static technique achieves comparable efficiency, it is not as robust as DDLDroid.

4.5 RQ4: Severity of Data Loss Issues

We analyze the 302 true positives reported by DDLDroid, and obtain the number of different categories of data loss issues, respectively. Figure 8 shows that among the 302 data loss issues, 29 (29/302 = 9.6%) cause app crashes, which is the severest consequence of data loss. Figure 9(a) presents a crash error in the app *GnuCash*. The text loss issues account for 69 (69/302 = 22.8%). In this case, users have to waste a lot of time to re-enter the previous input. Figure 9(b) exhibits the five editable text boxes with data loss issues on a password-change page in the app *Omni Notes*. The most common case is the dialog disappearance whose number is 204 (204/302 = 67.6%), which is the most likely to occur because the dialog usually plays a crucial role in handling user interactions and almost every app uses dialogs. For example, nine dialogs in the app *Conversations* suffer from loss issues and Figure 9(c) showcases one of them.

We also analyze the distribution of the 302 true data loss issues on the 66 apps. Surprisingly, 48 of the 66 apps suffer from data loss

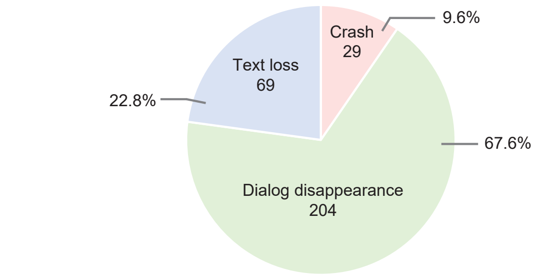
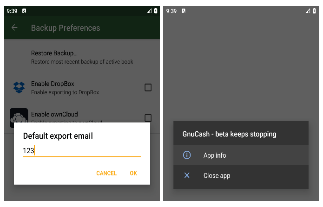
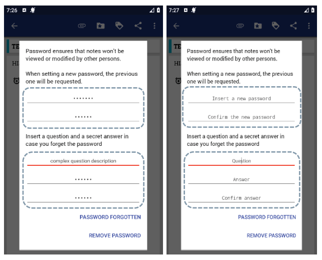


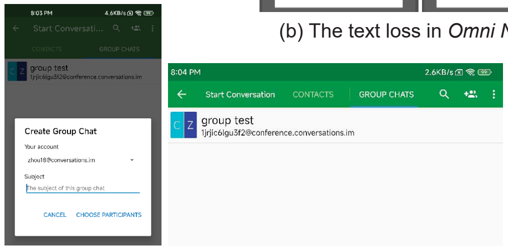
Figure 8: The distribution of data loss issues according to their consequences.



(a) The crash in *Gnucash*



(b) The text loss in *Omni Notes*



(c) The dialog disappearance in *Conversations*

Figure 9: Examples of data loss issues.

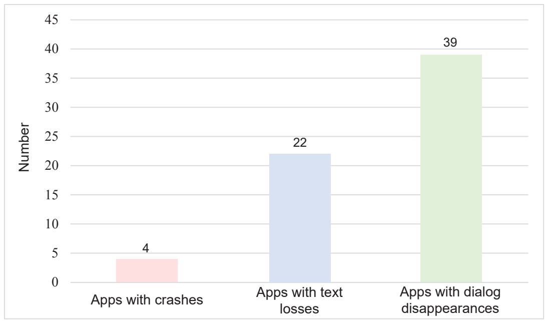


Figure 10: The distribution of data loss issues on the 66 apps.

issues. Figure 10 illustrates the number of apps affected by different categories of data loss issues in detail. Among the 48 apps with data loss issues, four (4/48 = 8.3%) apps are led to crashes, 22 (22/48 = 45.8%) apps suffer from text loss issues, and 39 (39/48 = 81.3%) apps are affected by dialog disappearances.

To investigate how developers regard these data loss issues, we have reported the previously unknown data loss issues of 20 apps whose developers are relatively active. Up to the time of the paper submission deadline, the developers of five apps have confirmed our submitted issues, which covers the three categories of data loss issues (*i.e.*, crash, dialog disappearance, and text loss). We quote some as follows: "*Anyway, it is indeed possible to save fields on rotation.*", "*Yes, it should retain state between screen rotation. I'll mark this as a minor bug*", "*Screen rotation within the New task activity is indeed not handled properly.*", etc. Therefore, data loss issues are relevant to the unconsciousness or even mistakes of developers in handling with the configuration change or the process death. More details of the developer feedback are summarized in Table 5.

Table 5: Details of Developer Feedback

App	Issue ID	Status
BeeCount	#36	Accepted
Vespucci	#2072	Accepted
AntennaPod	#6289	Accepted developer added label: Enhancement
Simpletask	#1216	Accepted developer added label: Bug
Omni Notes	#888	Accepted developer added label: Bug-Minor

Answer to RQ4: The 302 true data loss issues detected by DDLDroid are widely distributed in 48 (48/66 = 72.7%) apps. The most serious consequence of data loss is app crash, and the most common ones are non-crash data loss issues that lead to losing the existing progress of use. Moreover, the developers of five apps have confirmed our submitted data loss issues.
Implications: Data loss issues are prevalent in Android apps. In spite of that most data loss issues are not severe, the developers agree that fixing such bugs can enhance user experience and app quality.

4.6 Threats to Validity

Finally, we discuss the threats to the validity of our experiments.

Construct validity. One major threat to the validity of the experiment lies in the oracle construction which is based on our manual validation. Since we are not familiar with the apps, it is difficult for us to enter some targeted activity states from the main activity to check whether the data loss issues DDLDroid finds are true positives or false positives. The overall activity (activity state) coverage of our manual analysis is not high enough due to the complicated app functionality and our limited time. This impacts the accuracy calculation for DDLDroid. Fortunately, we have successfully confirmed 302 true positives, 27 false positives, and only 108 data loss issues DDLDroid finds are not checked. To reduce this threat, we plan to report those unconfirmed issues to the developers.

External validity. Another threat may stem from the apps used for the experimental comparison. Only 66 apps are used. Although these apps are also used in prior work [12, 14, 24], they do not necessarily reflect all programming practices in Android app development. To mitigate this threat, we plan to apply DDLDroid to more real-world Android apps.

5 RELATED WORK

5.1 Data Loss Detection

Recently, detecting data loss issues in Android apps has received much attention by the researchers and practitioners [1, 14, 16, 20, 22, 24, 26, 33]. In general, previous efforts on data loss detection fall into two categories: dynamic detection and static detection.

Dynamic detection. The lion's share of attention is focused on dynamic detection, which usually guides the app to some states and then to inject specific events to enter the scenarios that may trigger data loss issues. AppDoctor [16] and Crashscope [20] inject special events (*e.g.*, screen rotation) to the original test cases and use the new test cases to test apps. However, both approaches can only reveal the data loss issues that lead to app crashes. To this end, the subsequent work tries to find more data loss issues, not limited to those leading to crashes. Thor [1] augments existing test suites with neutral sequences of events that concern with the connectivity, the audio service, and the lifecycle of the activities, to check whether the supposedly-neutral sequences result in test failures (*e.g.*, menu or dialog disappears). Similarly, based on an available GUI model, Quantum [33] also embeds some events that may trigger data loss issues into their test cases, and then compares the GUI screenshots before and after these events are embedded for data loss confirmation. Without requiring available test cases, some techniques explore apps automatically to find data loss issues [14, 22, 24]. Both ALARic [22] and DLD [24] exploit screen rotations (which causes activity restart) to trigger data loss issues. The difference lies in that ALARic [22] employs a random test case generation technique whereas DLD [24] is based on a biased exploration strategy. Recently, based on an extracted GUI model via static analysis, iFixDataloss [14] automatically explores the app to trigger different data loss scenarios. This approach can find data loss issues during activity restart and app relaunch. While it takes a long time (*e.g.*, one hour) to explore the app, the activity coverage is still not satisfactory, and thus it may miss many data loss issues.

Since the events injected by dynamic techniques are some special cases of activity restart or app relaunch, only a part of data loss scenarios can be triggered. Thus, the data loss issues that are triggered by the other events (*e.g.*, the system theme switching) cannot be revealed. Moreover, the activity coverage of the dynamic detection approaches is not high, which inspires us to employ a static detection approach.

Static detection. There are only few static analysis techniques for data loss detection. KREFinder [26] regards all the mutable fields in an activity as the data that can be lost. Since some fields are irrelevant to the GUI display, it reports lots of false positives. To reduce the false alarms, it further spends much time validating the obtained results manually for issue confirmation. Recently, LiveDroid [12] proposes the liveness analysis and may-modify analysis to identify the necessary data that needs saving and restoring. Since it relies on access paths to connect variables that need saving to variables that need restoring, many data loss issues are missed. Additionally, it reports many variables and widgets that are not related to saving and restoring UI states. By contrast, DDLDroid extracts the variables that need restoring and saving through the widget API invocations and then checks the data flow of the restoring and saving operations, and thus it not only outputs the widgets with data loss issues but also reduces the false positives by excluding the irrelevant variables and widget properties.

5.2 Data Loss Fixing

Bug fixing for Android apps also attracts attention [11–14, 17, 18, 23, 30, 35]. In this part, we only review the related work on fixing data loss issues in Android apps [11, 12, 14, 17, 23].

Data loss mitigation. There are some approaches that aim to protect the app against data loss issues [11, 17]. For example, RuntimeDroid [11] employs an online resource loading module to update GUI automatically and a novel UI component migration technique to preserve prior user changes during configuration changes at runtime. In this way, it can help apps mitigate data loss issues during activity restart. Marvin [17] proposes a novel memory manager for Android, which transfers apps to the external storage instead of killing them to reclaim the memory when the memory runs low. Thus, it avoids data loss issues during app relaunch.

Data loss fixing. Instead of mitigating data loss scenarios, several techniques try to generate code patches to fix data loss issues [12, 14, 23]. LiveDroid [12] first identifies a large number program variables and GUI properties as the critical app data to be saved. It then adds the data saving and restoring code into the APK file to fixing potential data loss issues. However, it suffers from the over-saving problem because lots of variables and GUI properties do not need to be saved. DataLossHealer [23] dynamically fixes the data loss issues based on the Xposed Framework [25], which can intercept method invocations and change the behavior of an app using run-time hooking and code injection mechanisms. iFixDataloss [14] fixes data loss issues by preserving variable values and restoring them based on patch templates. It adds the implementation of these patch templates into the app source code. However, both iFixDataloss [14] and DataLossHealer [23] may miss many opportunities of fixing data loss issues.

6 CONCLUSIONS

Data loss issues in Android apps are closely relevant to the user experience and have received an increasing attention in recent years. If Android apps do not correctly deal with the data relevant to GUI display, data loss issues may occur. In this paper, we present a static analysis approach, DDLDroid, to detect data loss issues in Android apps. We evaluate DDLDroid on a set of 66 apps. DDLDroid successfully detects from the 66 apps a total of 302 true data loss issues, 180 of which are previously unknown. The average and median runtime overhead of DDLDroid on the 66 apps is 66 and 11 seconds, respectively. Thus, DDLDroid is more efficient than the state-of-the-art dynamic data loss detectors which usually spend much more time (*e.g.*, as much as hours) in exploring large apps.

DATA AVAILABILITY

All our artifacts are publicly available[1].

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China under Grant No. 61761136003.

[1]https://doi.org/10.5281/zenodo.7907006

REFERENCES

[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic execution of Android test suites in adverse conditions. In *Proceedings of the 24th International Symposium on Software Testing and Analysis, ISSTA'15, Baltimore, MD, USA, July 12-17*. ACM, 83–93. https://doi.org/10.1145/2771783.2771786

[2] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2017. The Soot-Based Toolchain for Analyzing Android Apps. In *Proceedings of the 4th IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE'17, Buenos Aires, Argentina, May 22-23*. IEEE, 13–24. https://doi.org/10.1109/MOBILESoft.2017.2

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14, Edinburgh, United Kingdom, June 09 - 11*. ACM, 259–269. https://doi.org/10.1145/2594291.2594299

[4] Android Developers. 2022. The Activity Lifecycle. [Online]. Available: https://developer.android.com/guide/components/activities/activity-lifecycle.

[5] Android Developers. 2022. App widgets overview. [Online]. Available: https://developer.android.google.cn/develop/ui/views/appwidgets/overview?hl=en.

[6] Android Developers. 2022. Fragments. [Online]. Available: https://developer.android.com/reference/android/app/Fragment.

[7] Android Developers. 2022. Overview of memory management. [Online]. Available: https://developer.android.com/topic/performance/memory-overview.

[8] Android Developers. 2022. Processes and Application Lifecycle. [Online]. Available: https://developer.android.google.cn/guide/components/activities/process-lifecycle.

[9] Android Developers. 2022. Runtime-changes. [Online]. Available: https://developer.android.com/guide/topics/resources/runtime-changes.

[10] Android Developers. 2022. Save UI states. [Online]. Available: https://developer.android.com/topic/libraries/architecture/saving-states.html.

[11] Umar Farooq and Zhijia Zhao. 2018. RuntimeDroid: Restarting-Free Runtime Change Handling for Android Apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'18, Munich, Germany, June 10-15*. ACM, 110–122. https://doi.org/10.1145/3210240.3210327

[12] Umar Farooq, Zhijia Zhao, Manu Sridharan, and Iulian Neamtiu. 2020. LiveDroid: identifying and preserving mobile app state in volatile runtime environments. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 160:1–160:30. https://doi.org/10.1145/3428228

[13] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-usage update for Android apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'19, Beijing, China, July 15-19*. ACM, 204–215. https://doi.org/10.1145/3293882.3330571

[14] Wunan Guo, Zhen Dong, Liwei Shen, Wei Tian, Ting Su, and Xin Peng. 2022. Detecting and fixing data loss issues in Android apps. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'22, Virtual Event, South Korea, July 18 - 22*. ACM, 605–616. https://doi.org/10.1145/3533767.3534402

[15] Yuyu He, Lei Zhang, Zhemin Yang, Yinzhi Cao, Keke Lian, Shuai Li, Wei Yang, Zhibo Zhang, Min Yang, Yuan Zhang, and Haixin Duan. 2020. TextExerciser: Feedback-driven Text Input Exercising for Android Applications. In *Proceedings of the IEEE Symposium on Security and Privacy, SP'20, San Francisco, CA, USA, May 18-21*. IEEE, 1071–1087. https://doi.org/10.1109/SP40000.2020.00071

[16] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. 2014. Efficiently, effectively detecting mobile app bugs with AppDoctor. In *Proceedings of the Ninth Eurosys Conference, EuroSys'14, Amsterdam, The Netherlands, April 13-16*. ACM, 18:1–18:15. https://doi.org/10.1145/2592798.2592813

[17] Niel Lebeck, Arvind Krishnamurthy, Henry M. Levy, and Irene Zhang. 2020. End the Senseless Killing: Improving Memory Management for Mobile Operating Systems. In *Proceedings of the USENIX Annual Technical Conference, USENIX ATC'20, July 15-17*. USENIX Association, 873–887. https://www.usenix.org/conference/atc20/presentation/lebeck

[18] Forough Mehralian, Navid Salehnamadi, and Sam Malek. 2021. Data-driven accessibility repair revisited: on the effectiveness of generating labels for icons in Android apps. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE'21, Athens, Greece, August 23-28*. ACM, 107–118. https://doi.org/10.1145/3468264.3468604

[19] Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, and Denys Poshyvanyk. 2018. Automated reporting of GUI design violations for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering, ICSE'18, Gothenburg, Sweden, May 27 - June 03*. ACM, 165–175. https://doi.org/10.1145/3180155.3180246

[20] Kevin Moran, Mario Linares Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation, ICST'16, Chicago, IL, USA, April 11-15*. IEEE Computer Society, 33–44. https://doi.org/10.1109/ICST.2016.34

[21] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse Engineering Mobile Application User Interfaces with REMAUI (T). In *Proceeding of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE'15, Lincoln, NE, USA, November 9-13*. IEEE Computer Society, 248–259. https://doi.org/10.1109/ASE.2015.32

[22] Vincenzo Riccio, Domenico Amalfitano, and Anna Rita Fasolino. 2018. Is this the lifecycle we really want?: an automated black-box testing approach for Android activities. In *Proceedings of the Companion Proceedings for the ISSTA/ECOOP Workshops, ISSTA'18, Amsterdam, Netherlands, July 16-21*. ACM, 68–77. https://doi.org/10.1145/3236454.3236490

[23] Oliviero Riganelli, Daniela Micucci, and Leonardo Mariani. 2016. Healing Data Loss Problems in Android Apps. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops'16, Ottawa, ON, Canada, October 23-27*. IEEE Computer Society, 146–152. https://doi.org/10.1109/ISSREW.2016.50

[24] Oliviero Riganelli, Simone Paolo Mottadelli, Claudio Rota, Daniela Micucci, and Leonardo Mariani. 2020. Data loss detector: automatically revealing data loss bugs in Android apps. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'20, Virtual Event, USA, July 18-22*. ACM, 141–152. https://doi.org/10.1145/3395363.3397379

[25] Rovo89. 2022. Xposed Module Repository. [Online]. Available: https://repo.xposed.info/.

[26] Zhiyong Shan, Tanzirul Azim, and Iulian Neamtiu. 2016. Finding resume and restart errors in Android applications. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'16, part of SPLASH, Amsterdam, The Netherlands, October 30 - November 4*. ACM, 864–880. https://doi.org/10.1145/2983990.2984011

[27] Wei Song, Jing Zhang, and Jeff Huang. 2019. ServDroid: detecting service usage inefficiencies in Android applications. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE'19, Tallinn, Estonia, August 26-30*. ACM, 362–373. https://doi.org/10.1145/3338906.3338950

[28] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. 2021. Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–31. https://doi.org/10.1145/3485533

[29] Jingling Sun, Ting Su, Junxin Li, Zhen Dong, Geguang Pu, Tao Xie, and Zhendong Su. 2021. Understanding and finding system setting-related defects in Android apps. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'21, Virtual Event, Denmark, July 11-17*. ACM, 204–215. https://doi.org/10.1145/3460319.3464806

[30] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing crashes in Android apps. In *Proceedings of the 40th International Conference on Software Engineering, ICSE'18, Gothenburg, Sweden, May 27 - June 03*. ACM, 187–198. https://doi.org/10.1145/3180155.3180243

[31] Jue Wang, Yanyan Jiang, Ting Su, Shaohua Li, Chang Xu, Jian Lu, and Zhendong Su. 2022. Detecting non-crashing functional bugs in Android apps via deep-state differential analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE'22, Singapore, Singapore, November 14-18*. ACM, 434–446. https://doi.org/10.1145/3540250.3549170

[32] Ryszard Wiśniewski and Connor Tumbleson. 2020. Apktool. [Online]. Available: https://ibotpeaches.github.io/Apktool/.

[33] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. 2014. Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps. In *Proceedings of the Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST'14, Cleveland, Ohio, USA, March 31-April 4*. IEEE Computer Society, 183–192. https://doi.org/10.1109/ICST.2014.31

[34] Zhen Zhang, Yu Feng, Michael D. Ernst, Sebastian Porst, and Isil Dillig. 2021. Checking conformance of applications against GUI policies. In *Proceeding of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE'21, Athens, Greece, August 23-28*. ACM, 95–106. https://doi.org/10.1145/3468264.3468561

[35] Yanjie Zhao, Li Li, Kui Liu, and John C. Grundy. 2022. Towards Automatically Repairing Compatibility Issues in Published Android Apps. In *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE'22, Pittsburgh, PA, USA, May 25-27*. ACM, 2142–2153. https://doi.org/10.1145/3510003.3510128

Received 2023-02-16; accepted 2023-05-03