Detecting Potential User-data Save & Export Losses due to Android App Termination

Sydur Rahaman
New Jersey Inst. Of Technology
Newark, NJ, USA
sr939@njit.edu

Umar Farooq
U. of California, Riverside
Riverside, CA, USA
ufaro001@ucr.edu

Iulian Neamtiu
New Jersey Inst. Of Technology
Newark, NJ, USA
ineamtiu@njit.edu

Zhijia Zhao
U. of California, Riverside
Riverside, CA, USA
zhijia@cs.ucr.edu

***Abstract*—A common feature in Android apps is saving, or exporting, user's work (e.g., a drawing) as well as data (e.g., a spreadsheet) onto local storage, as a file. Due to the volatile nature of the OS and the mobile environment in general, the system can terminate apps without notice, which prevents the execution of file write operations; consequently, user data that was supposed to be saved/exported is instead lost. Testing apps for such potential losses raises several challenges: how to identify data originating from user input or resulting from user action (then check whether it is saved), and how to reproduce a potential error by terminating the app at the exact moment when unsaved changes are pending. We address these challenges via an approach that finds potential "lost writes", i.e., user data supposed to be written to a file, but the file write does not take place due to system-initiated termination. Our approach consists of two phases: a static analysis that finds potential losses and a dynamic loss verification phase where we compare lossy and lossless system-level file write traces to confirm errors. We ran our analysis on 2,182 apps from Google Play and 38 apps from F-Droid. Our approach found 163 apps where termination caused losses, including losing user's app-specific data, notes, photos, user's work and settings. In contrast, two state-of-the-art tools aimed at finding volatility errors in Android apps failed to discover the issues we found.**

***Index Terms*—Android, Mobile Apps, Static Analysis, Dynamic Analysis, Persistence**

I. Introduction

Creating and editing persistent user data is a basic features in Android apps. Examples range from creating notes, saving app-specific change histories, creating schedules, or editing pictures. This user-created data can be exported or backed up as files onto local storage; the action is typically initiated via the user, e.g., by pressing a 'Save' or 'Export' button. These file saves can be jeopardized by app interruption or abrupt app termination (as explained shortly). However testing whether such data is saved correctly requires solving two challenges: (1) identifying the extent of the data, and (2) creating the termination conditions at the exact moment when the data is in-memory but not committed onto file storage via I/O yet.

Due to resource and security constraints, mobile platforms such as Android or iOS are volatile, especially compared to their desktop/server counterparts. For example, when in foreground, apps have access to resources, but phone and user events such as orientation changes or incoming calls will perturb the app state; once sent into the background, an app is subject to resource reclamation or swift termination [1]–[3]. We name *system-initiated termination* the scenario when an Android app is terminated swiftly by the system, e.g., due to memory pressure or limits on background processes [4], [5]. Such termination has three characteristics: (1) it is less frequent than user-initiated termination, (2) the termination is swift, without giving the app a "courtesy" notice, and (3) the user is unaware of it (silent failure), making it more concerning. In particular, swift termination might lead to loss of user-created data that should have been saved into files via file write operations upon user-initiated save/export actions. There have been recent efforts to find the extent of user's data as a form of UI input that could be lost due to volatility (e.g., lifecycle events – orientation changes, app pausing, or receiving a phone call). These efforts have focused on app fields [6] or variables and UI properties ("instance state" [7]) that can be lost due to lifecycle events. However, prior work has not considered scenarios where not only the user's UI input data, but also user action-based data saved as file writes can be lost when apps are terminated swiftly.

We solve the aforementioned challenges as follows: our approach finds user input or user action-based file write data, specifically users' work that ordinarily would be saved onto the file system, and checks whether the data could be lost due to abrupt system-initiated termination.

In Section II we provide background and motivation. Section II-A discusses details on system-initiated termination and its impact on file writes, while Section II-B illustrates losses in two apps, user drawings in the Acrylic Paint app and calculator screenshots in the Wabbitemu app.

Note that even finding potential losses is challenging as it entails (a) defining and identifying the data that has to be saved, and (b) understanding whether, and how, this data flows to files. We address these challenges by introducing an automated approach that combines static and dynamic analysis. We start with finding *user-initiated file writes* which are identified via a suite of static analyses, described in Section III-A. Put simply, the data of interest is generated, modified, or saved by the user via UI actions and would normally (i.e., barring termination) reach file write methods to be saved onto persistent storage. Our automated static analysis produces a list of objects that are potentially lost due to system-initiated termination. To verify

This material is based upon work supported by the National Science Foundation under Grants No. CCF-2106710 and CCF-2106383.

these losses, we compare the system call traces (obtained dynamically) between the original, no-loss execution and the terminated, lossy execution (Section III-B).

We evaluated our approach on 2,220 apps (2,182 from Google Play and 38 from F-Droid). We found, and confirmed, losses in 163 apps. Examples of such losses, found in widely popular apps with more than 5 million installs, are described in detail in Section IV-B: user settings, artwork, edited photos, notes, history, or bookmarks. In Section IV-C we compare our approach with two state-of-the-art approaches for finding volatility-induced UI data – KREfinder and LiveDroid – and show that those approaches fail to expose the data lost due to system-initiated termination. Finally, in Section IV-G we propose potential solutions to fix file write losses.

Our work makes several contributions:

- A static analysis that identifies potential UI-to-file losses due to system-initiated termination.
- A dynamic analysis that confirms potential losses via OS-level trace differencing.
- An automated approach that enables developers to test apps for user-data loss.
- A study on 2,220 apps that has revealed such losses in 163 apps.

II. Motivation

A fundamental principle in mobile app development – on both Android and iOS – is to "not perform file writes on the synchronous UI thread, to keep the UI responsive" [8], [9]. This forces programmers to run file write operation in a separate thread, e.g., asynchronously via an AsyncTask in Android. However, asynchronous (and potentially time-consuming) file writes are on a collision course with the volatility of mobile platforms. Specifically, mobile apps can be terminated without notice (or on short notice) by the system, due to low memory or runtime changes. We detail this by first presenting a brief overview of Android app construction, file writes in app, and app termination; next, to motivate our approach, we present a suite of examples of file write losses due to termination.

A. Background: File Writes and Termination in Android

1) App Construction and File Writes: Android apps are constructed from four fundamental components: Activities managing the UI, ContentProviders managing access to data, Services that run in the background, and Broadcast Receivers which respond to system-wide events. The most common component, Activity, is a "page" in the app; apps with a UI typically consist of one or more Activities. The UI elements in an Activity are owned and managed by a special thread, named the *UI thread* or *main thread*. The UI thread plays a critical role: timely processing of UI events, to keep the UI (and app) responsive. Therefore, one of the first lessons in Android programming is "you should not perform work on the UI thread" [8]: as file write is potentially long-running and blocking, performing a file write on the UI thread could render the app unresponsive. Hence any long-running or blocking operations should run asynchronously, in a different (background) thread. Occasionally though, apps violate this requirement: among the apps we have analyzed, some perform file writes on the UI thread.

2) App Termination: There is an inherent tension between long-running operations and the constraints of the mobile platforms. Unlike desktop/server applications, mobile applications cannot expect to "run forever"; rather, mobile applications can be terminated summarily to free resources such as memory [4], conserve energy, and protect user's security (e.g., by preventing background apps from accessing users' location). Therefore, long-running operations can be interrupted or terminated without notice – in fact the entire process enclosing the Android app is terminated – for various external reasons, as listed below.

1) **Memory pressure.** Android's Low Memory Killer Daemon (LMKD) [2] handles low-memory situations: when the phone is under memory pressure, the LMKD ranks apps based on their memory usage and acts according to a configuration-described policy, e.g., the app which consumes most memory and is not in foreground will be killed to release memory.[1]
2) **Background process limit.** The Android OS provides a developer option to limit the number of background processes. When the option is set to "No background processes" an app process is killed whenever the app is not in the foreground.
3) **Kill via external signal.** Apps can also be terminated by sending them a traditional Unix signal, e.g., SIGKILL.

Besides system-initiated termination, an app process can also be terminated internally, when the app invokes API methods such as System.exit or finishAndRemoveTask; the use of these APIs in our app dataset was practically non-existent, so our approach focuses on system-initiated termination.

B. Motivational Examples

We now present several examples where file write loss can occur due to system-initiated termination.

1) Example: File write on the main thread: We show a case study on the Acrylic Paint app [11] in Figure 1. Specifically, we show two scenarios: a successful scenario where file write operation completes, and an unsuccessful scenario where the file write is eschewed due to system-initiated termination, leading to user's work being lost. Acrylic Paint is a painting app: the user can save painting progress either by clicking the 'Save' button (Figure 1(a)) or by exiting the app. The user's progress or changes to the drawing are saved in a local file inside the app directory, as shown in the middle part of the figure: Figure 1(b) shows the confirmation message, whereas Figure 1(c) shows the saved painting as a file in the app's directory. However, in the case of a system-initiated termination such as low memory, the app process is terminated before the user's changes, or progress, can be saved. In that case, illustrated in Figure 1(d), the valuable file write data is lost and the file is missing from the app's local directory.

[1] In smartphones, I/O consumes substantial energy [10], hence apps that run background I/O are at higher risk of termination due to resource pressure.

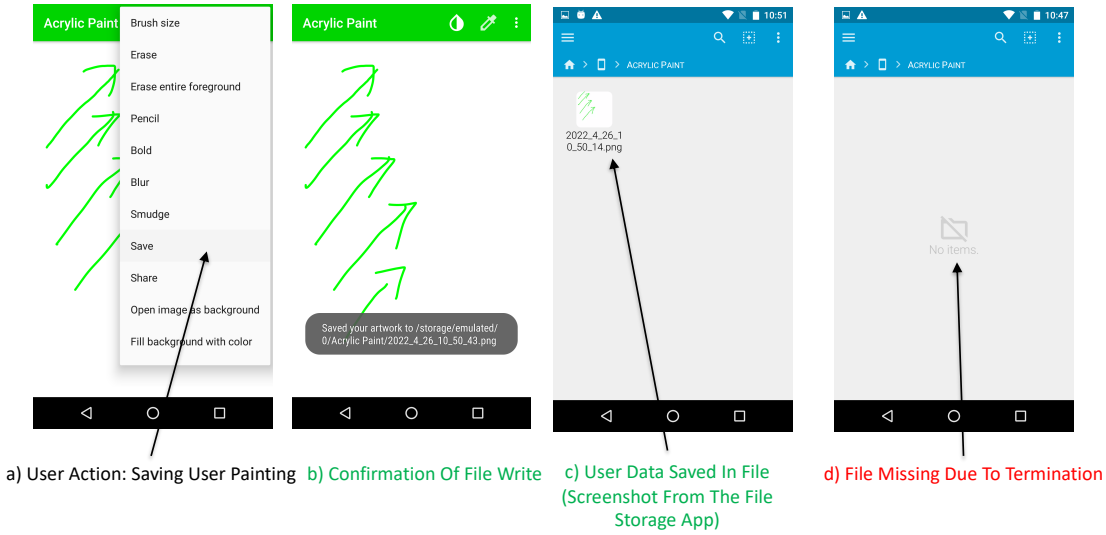

Fig. 1. Acrylic Paint app: success scenario (a-c) and user file write data loss scenario (d).

```
1  public boolean onOptionsItemSelected(MenuItem item) {
2      switch (item.getItemId()) {
3       ....
4          case R.id.save_menu:
5              takeScreenshot(true);
6              break;
7       ....}}
8  private File takeScreenshot(boolean showToast) {
9       ......
10     Bitmap copyBitmap = cachedBitmap.copy(Bitmap.Config.ARGB_8888, true);
11     File file = new File(path,"fileName.png");
12     FileOutputStream output = new FileOutputStream(file);
13     copyBitmap.compress(CompressFormat.PNG, 100, output); //Process gets
           terminated here
14      .....}
```

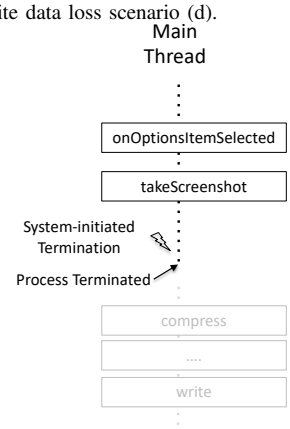

Fig. 2. Acrylic Paint app code (left) and file write operation termination events (right).

In Figure 2 we show the app source code (left) and event sequence diagram (right). When the user clicks the 'Save' menu option, the onOptionsItemSelected event is triggered (lines 1–4). The method takeScreenshot is called next (line 5); inside the method, new Bitmap, File, and FileOutputStream instances are created (lines 10–12). Finally, the file writing operation Bitmap.compress (line 13) executes on the main thread. If the app is terminated prematurely, the file write operation is terminated, resulting in data loss. The sequence of events, i.e., code that will not execute due to termination, is shown in gray.

2) Example: file write on a background thread: Next, we show an example of file write loss due to termination, where the file write operation is performed on a background thread. Wabbittemu [12] is a graphic calculator app. We show relevant app screenshots in Figure 3: the user can save calculator screenshots locally in a file by selecting the 'Screenshot' menu item (Figure 3(a)); the save confirmation is shown in Figure 3(b), whereas Figure 3(c) confirms the file's presence in the app's own directory. The file write loss scenario, due to system-initiated termination, is shown in Figure 3(d): data is lost, hence the file is missing from the directory.

Figure 4 shows the relevant source code (left) and event sequence diagram (right). When the user selects the screenshot menu item, the ScreenshotCalcTask executes. Note that ScreenshotCalcTask extends the AsyncTask class hence will execute asynchronously as follows: ScreenshotCalcTask invokes the SaveScreenshotCalc method on a background thread (line 4). Inside the SaveScreenshotCalc method, first, Bitmap and FileOutputStream instances are created (lines 8,10), then the screenshot image is written into a file (line 11). App termination in turn terminates both the main thread and its (child) background thread, hence the file write data will be lost; specifically, the file write operation and the onPostExecute method do not execute. The gray-colored part of the sequence diagram shows the parts that will not execute due to termination.

Hence our goal is to automatically identify the file-bound data and file write operations that are lost (and do not execute, respectively) due to termination.

III. Approach

In Figure 5 we provide an overview of our approach. Given an Android app we first perform a *static analysis* which produces reports of *user-initiated file writes*, i.e., file data that originates from UI events, hence is potentially lost. Then, we verify the potential losses in a *dynamic report verification* phase. We now discuss each phase in detail.

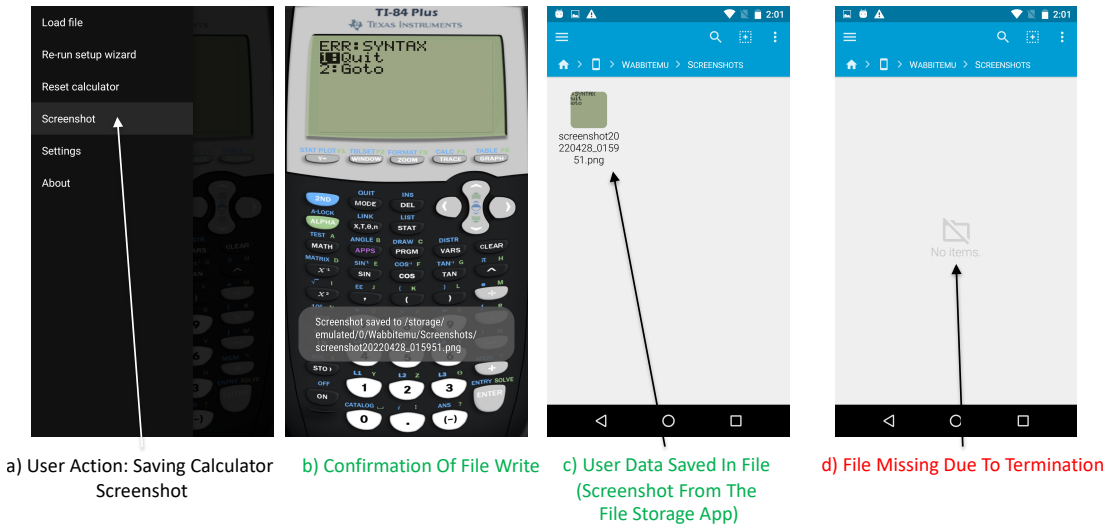

Fig. 3. Wabbitemu app: success scenario (a-c) and user file write data loss scenario (d).

```
1  private class ScreenshotCalcTask extends AsyncTask{
2  protected void onPreExecute() {...}
3  protected Boolean doInBackground() {
4      SaveScreenshotCalc();
5  }
6  void SaveScreenshotCalc() {
7       .....
8      Bitmap Screenshot = Bitmap.createScaledBitmap(screenshot,
           screenshot.getWidth() * 2, screenshot.getHeight() * 2, true);
9      try {
10         FileOutputStream out = new FileOutputStream(new
               File(outputDir, "screenshot" + new
               SimpleDateFormat("yyyyMMdd",
               Locale.getDefault()).format(new Date()) + ".png"));
11         Screenshot.compress(CompressFormat.PNG, 100, out);
               //Process gets terminated here
12          ....
13         }
14      ....
15     }
16 protected void onPostExecute(Boolean success) {...} }
```

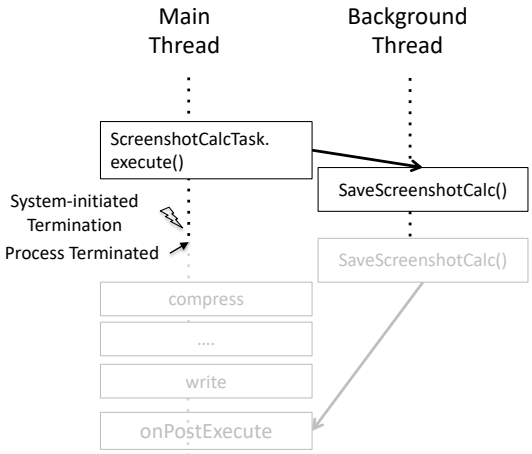

Fig. 4. Wabbitemu app code (left) and file write operation termination events (right).

A. Static Analysis

In this phase, we perform a combination of control and data flow analyses to identify data that originates from user interaction (UI) and flows into file write APIs; this data is marked as potential loss.

Our static analysis has two main objectives:

- Finding all the file write operations initiated by (generated from) user interactions with the app, and
- Tracking the data that contains user input and flows into the aforementioned file write operations.

We first define "user-initiated" more precisely, then proceed to describe how we achieved the aforementioned objectives using a combination of control-flow and data-flow analysis.

1) What is "User-initiated": A key requirement for finding potential data losses is defining exactly what data is "worth saving". Intuitively, user's work or changes are worth saving, whereas logging operations happening in analytics libraries are not. Hence we define as "worth saving" two kinds of *user-initiated* file write operations.

First, we consider file write operations where the file content is coming directly from UI input, e.g., the canvas in the Acrylic Paint app shown in Figure 1, which has type ImageView. Another example is note contents, whose type is TextView. These file writes are identified via data-flow analysis.

The second kind of worthy content is file-written but does not come from UI input; rather, the file write operation depends on user interaction with the UI such as saving screenshots or exporting log data. One such example is the 'Export to Excel' UI action in the Auto-Away app to export call history log mentioned in detail in Section IV-B. Though in that case the exported file does not contain UI data (in contrast to the painting content above), the user nevertheless initiates this file write or export operation. We find these type of file writes via control-flow analysis.

2) Defining UI Interaction Callbacks: Android apps do not have a main() method; rather, apps have multiple entry points

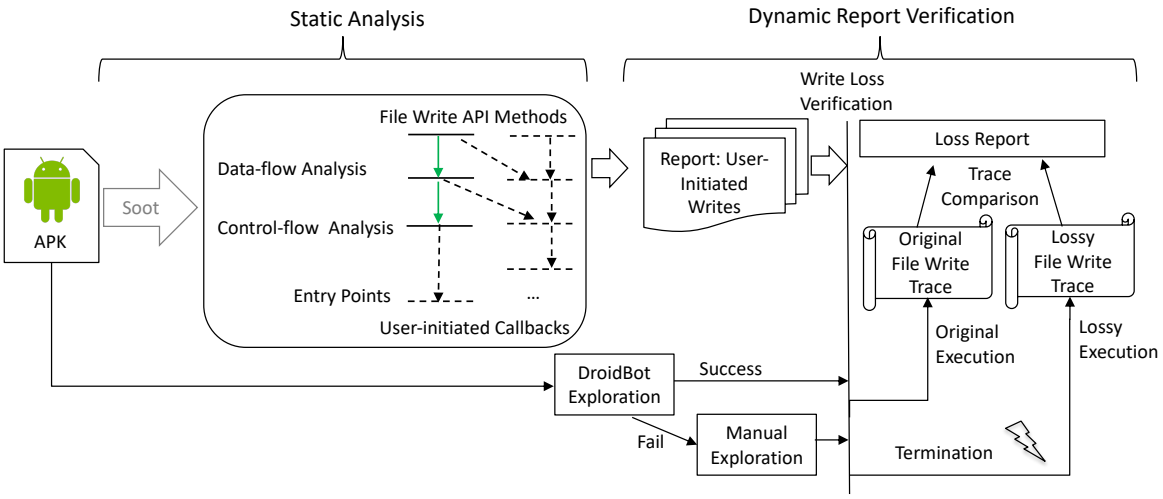

Fig. 5. Overview of our approach.

TABLE I
FILE WRITE API PREVALENCE.

File Writing API	% Apps
OutputStream	89
FileOutputStream	82
Writer	81
FilterOutputStream	81
ByteArrayOutputStream	79
StringWriter	72
BufferedOutputStream	66
BufferedWriter	54
ObjectOutputStream	47
DataOutputStream	43
FileWriter	15
PrintStream	10
PrintWriter	9
CharArrayWriter	3
FilterWriter	1

induced by top-level callback events, as follows. Apps can register callbacks for events of interest, such as GPS location updates or UI interaction (menu select, button click, etc.). We create a "dummy" main method (similar to other static analyses such as FlowDroid [13]) which contains all top-level callbacks and will serve as an end point for backward analyses. Among the top-level callbacks, we only retain UI-related ones. There are several UI interaction callback APIs in Android, such as OnClick, onMenuItemClick, etc. Generally these callbacks are part of the Android View (UI) components hence defined in the android.view class hierarchy. Non-UI callbacks are not a target of our analysis. For example onLocationChanged, defined in android.location.Location and handling GPS location updates, does not correspond to user interaction and is not considered a UI interaction endpoint.

3) Defining File Write APIs: Our analysis focuses on file write operations. We manually identified 15 java.io classes that support file writes. The first column of Table I lists these classes and the second column states the frequency of the APIs observed in our evaluated app dataset. Within these classes, we identified API methods that perform file writes, e.g., FileOutputStream.write(), Writer.append(), etc. We observed that the most common API classes were the *Stream and *Writer families, e.g., OutputStream, FileOutputStream, Writer. Note that this list is just an input configuration file in our implementation hence can be easily extended.

4) Finding User-initiated File Writes: Our static analyzer is built on top of the Soot analysis framework [14]. Using Soot, we first build an inter-procedural call graph, which will form the basis for the control- and data-flow analyses.

To find all the file write calls originating from user interaction, we proceed as follows. We perform a backward control-flow analysis from every file write callsite back to its app entry point. If the callback belongs to UI interaction callbacks (Section III-A2), then the write operation is initiated by the user, and we add this write to our list.

Example. We show an example of how our backward control-flow analysis operates in Figure 6. The example is drawn from the Privacyfriendlynotes app (a simple note-saving app). The relevant source code is shown on the left, and the corresponding control-flow diagram along with the app UI screenshots are shown on the right. We start our backward analysis from the file write API. In this case, the file write API is on line 23: the compress method call (taking a FileOutputStream as an argument) is the point where the note or sketch are saved in a file. From compress our analysis lands in saveToExternalStorage; backtracking from the saveToExternalStorage method leads to two different paths, as the method has two callers. One of them is onRequestPermissionsResult (lines 13–18) which does not belong to android.view class, hence is not a UI interaction callback. The second control-flow path traces back to onOptionsItemSelected which belongs to the android.view.MenuItem class, hence is a UI interaction callback. When the user selects the 'SAVE' option, the onOptionsItemSelected callback is triggered and saveToExternalStorage is called (lines 6–8). Hence this particular file write falls under the category of writes initiated by the user. The control-flow path that satisfies our requirement and belongs to the "user-initiated file write" path is marked with green color in the figure. Hence, we add the file write operation on line 23 as a user-initiated file write.

```
1  class SketchActivity extends AppCompatActivity{
2  public boolean onOptionsItemSelected(MenuItem item) {
3          int id = item.getItemId();
4          if (id == R.id.action_reminder) {
5              ....
6          }else if (id == R.id.action_save) {
7              ...
8              saveToExternalStorage();
9              ...
10         }
11         ...
12     }
13 public void onRequestPermissionsResult(int
       requestCode,..) {
14     switch (requestCode) {
15         ....
16         saveToExternalStorage();
17     }
18 }
19 private void saveToExternalStorage(){
20     .....
21     Bitmap bm = overlay(new
           BitmapDrawable(getResources(),
           mFilePath).getBitmap(), drawView.getBitmap());
22     ...
23     bm.compress(Bitmap.CompressFormat.JPEG, 100, new
           FileOutputStream(file));
24     ....
25 } }
```

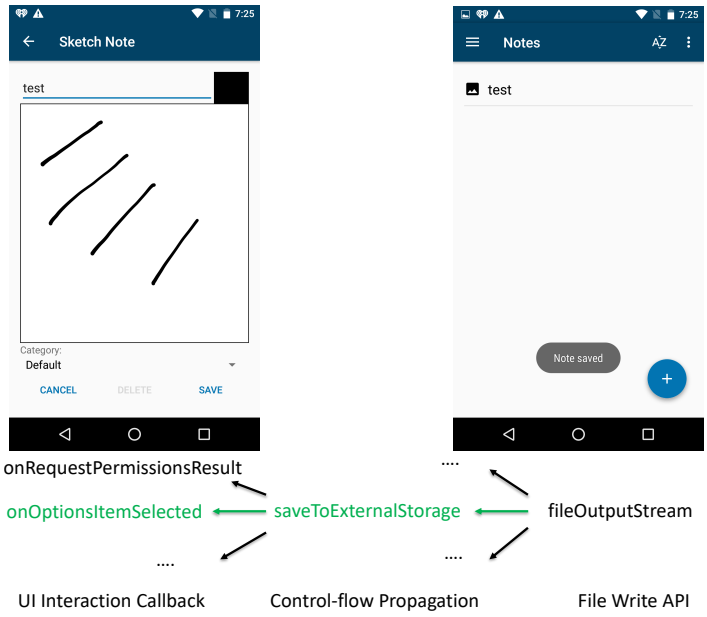

Fig. 6. Backward control-flow analysis in the Privacyfriendlynotes app.

```
1  private File takeScreenshot(boolean showToast) {
2      ......
3      View v = findViewById(R.id.CanvasId);
4      v.setDrawingCacheEnabled(true);
5      Bitmap cachedBitmap = v.getDrawingCache();
6      Bitmap copyBitmap =
           cachedBitmap.copy(Bitmap.Config.ARGB_8888, true);
7      File file = new File(path,"fileName.png");
8      FileOutputStream output = new FileOutputStream(file);
9      copyBitmap.compress(CompressFormat.PNG, 100, output);
10     .....}
```

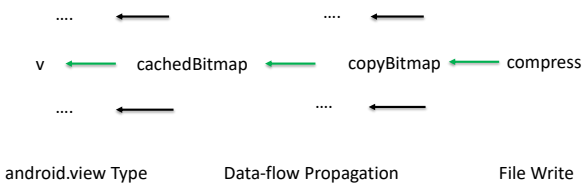

Fig. 7. Backward data-flow analysis in the Acrylic Paint app.

5) Finding User Input Flowing to File Writes: To find the extent of the data flowing to file write APIs, we perform a data-flow analysis. Similar to the control-flow analysis, we start our data-flow analysis from the write callsite (but now consider the method arguments) and trace whether the data transitively flows from a UI input class. Starting from the file write API callsite, we run a backward data-flow analysis up to the point where the data type belongs to a UI input type (android.view class) or exits via an app entry point. We now illustrate this analysis with an example.

Example. We show the data-flow analysis of the Acrylic Paint app in Figure 7. The relevant source code is shown on the left, and the corresponding data-flow edges on the right. The method in consideration here is takeScreenshot, which contains the file write call compress (line 9). Data-flow analysis of copyBitmap leads to line 6, specifically cachedBitmap. Tracing back from cachedBitmap leads to line 5, value v, which belongs to the android.view class as per line 3. Therefore, we end the data-flow analysis here, concluding that the file write content is coming from an UI input; in this example, it belongs to ImageView type UI input. Therefore, we add v as potential loss.

B. Dynamic Report Verification

Our dynamic verification phase reduces the false positives resulting from the static analysis phase. Given the list (reports) of user-initiated writes produced by the static analysis, we proceed to verify the potential losses report. Dynamic report verification has multiple components as discussed below:

1) GUI Exploration: Our goal is to explore the target app to find relevant file write initiating action (i.e., button click to initiate `Save/Export`) and then inject a termination event which should lead to a "lossy" execution and finally compare file write traces in the original and lossy executions. Those writes that are confirmed missing will help us verify whether the file writes containing user data is lost.

DroidBot Exploration. We have used DroidBot [15] to automate the app exploration or trace generation process. DroidBot's GUI-based model helped to automatically identify various view objects (such as Button or TextView) related to target user `Save/Export` actions. We wrote custom DroidBot scripts

```
1  // (a) succesful strace
2  openat(AT_FDCWD,"/data/user/0/org.secuso.privacyfriendlynotes/
3   files /sketches/sketch_1606849053910.PNG",
       O_WRONLY|O_CREAT|O_TRUNC, 0666) = 71
4
5  // (b) lossy strace
6  openat(AT_FDCWD,"/data/user/0/org.secuso.privacyfriendlynotes/
7   files /sketches/sketch_1607044227961.PNG",
       O_WRONLY|O_CREAT|O_TRUNC, 0666 <unfinished
       ...>
```

Fig. 8. `Strace` differences between a) successful file write and b) lossy execution in app PrivacyFrinedlyNotes.

to identify these target GUI elements and trigger necessary events (e.g., clicking `Save` button).

Manual Exploration. Besides DroidBot, we have explored apps manually (human-driven) for cases where DroidBot failed. DroidBot failed in two types of scenarios. First, the scenario where DroidBot could not reach targeted `Save`/`Export` options and the collected traces did not have the desired file write logs. Second, the cases where automated GUI exploration using DroidBot crashed and no trace logs were generated.

2) Triggering Termination: While termination can be triggered via various system events (Section II-A2), we used the background process limit option, based on the observation that background apps are frequently/routinely terminated due to memory pressure [4], [5], [16], [17].[2] In other words, if the user switches away from app A (which in the absence of termination would perform a file write operation, either on the main thread or on a background thread) to app B, the file write operation can be terminated, resulting in data loss. We automated switching from target app A to app B via Monkey [18] (the `adb shell monkey -p package.name` command offered by the Android Debugging Bridge). In case of manual exploration, we manually switched from the target app to another.

3) Trace Comparison: We confirm the write loss via automated trace differencing: we compare the `strace` Linux-level system call trace [19] across two runs: original execution (normal, uninterrupted) and lossy execution (operation interrupted by triggering termination via app switching). We have automated trace differencing by checking for interrupted I/O, e.g., unfinished openat(), fstat (), or write () system calls. We show an example of successful vs. lossy `strace`s for app PrivacyFriendlyNotes in Figure 8. The successful execution trace is shown on top – note the openat() call completing (lines 2–3). The lossy trace is shown on the bottom: the openat() call fails (unfinished) as shown in lines 6–7. Hence, aside from the visual confirmation of file data loss (e.g., Figure 1(d), Figure 3(d)), which is not scalable for a large set of apps, we automated the dynamic verification process via `strace` differencing; this dynamic verification is key to achieving a low false positive rate (Section IV-E).

[2]Typical background process limit (number of concurrent apps): 16 apps for mobile devices with 1GB memory and 26 apps for 2GB memory [16].

TABLE II
APP SELECTION AND FINDINGS.

	#Apps
Save/Export in UI	2,953
Soot Successful	2,220
Contains File Writes	1,476
User-initiated File Writes	298
Confirmed Losses	*163*
Automatically	*107*
Manually	*56*

TABLE III
CATEGORIES OF CONFIRMED LOSSES.

Category	**# Apps**
App-specific/misc.	61
Notes, documents, PDF files	28
Image, audio, video files	19
Database backups	17
Reports	14
Painting	8
Settings or preferences	6
History	4
Recipes	4
Schedules	2

IV. EVALUATION

We now discuss our evaluation in detail. We introduce our dataset, then quantify the effectiveness of our approach. Next, we present 27 examples of verified losses. We then compare our approach with existing tools, and quantify our analysis' efficiency. Finally, we discuss the limitations of our approach.

Dataset and test platform. To evaluate our approach, we focus on apps that support saving or exporting user data. First, we collected an app dataset of interest from the main Android app store, Google Play, and the open-source app store, F-Droid. To identify apps that offer Save or Export facilities, we used an automated filtering process on GUI data: more precisely, we extracted app resource XMLs from 20,000 popular Android apps split across across various app categories, and retained those apps whose GUI resources (e.g., buttons or menus) match keywords such as `Save`, `Export`, or similar. We found 2,953 apps whose GUI matched our keywords of interest. Next, we excluded 733 apps that could not be analyzed with Soot (on top of which we built our analysis). Soot failing on certain commercial apps is unsurprising, because popular Google Play apps tend to employ anti-analysis techniques such as packing or obfuscation. We ran our dynamic analysis on a Nexus 5 smartphone running Android 6 (API level 23).

A. Effectiveness

Table II summarizes our evaluation results, and is discussed in detail next. Among the 2,220 apps where Soot ran successfully, we identified those apps whose bytecode contained file write APIs, yielding 1,476 apps. We found that certain categories of file writes are not of interest: they are performed by third-party libraries, e.g., tracking and analytics packages that write into log files. We configured our analysis to ignore such writes – because they consist of logging and analytics data, they are out of our purview. Instead, our focus is on

user-initiated writes as mentioned in Section III-A: the static analysis has identified 298 such apps. Among the 298 candidate apps (apps with potential losses) having user-initiated file writes, we found and confirmed losses in 163 apps using our semi-automated dynamic verification approach – a combination of automated and manual analysis discussed next. We show the summary of our evaluation in Table II.

We categorize these confirmed losses in Table III. Most of the losses fall under the app-specific data category (e.g., saving newborn vitals for a baby care app). Other categories of lost data include notes, photos, database backups, artwork, or settings.

1) DroidBot Exploration Results: Using our automated GUI exploration with DroidBot, we produced trace logs for 177 cases out of 298 potential losses. DroidBot crashed and could not generate traces for 121 cases. This automated DroidBot-driven approach verified losses in 107 cases by comparing original and lossy execution traces. For the rest of the 70 cases, there were no differences between original and lossy execution traces due to the lack of file writes in the original execution (DroidBot exploration did not result in the desired file write operations).

2) Manual Exploration Results: For those 191 apps where DroidBot either crashed or could not reach the target GUI exploration, we performed a manual (human-driven) analysis. Our manual analysis confirmed losses in 56 apps, hence a total of 163 apps with automatically-confirmed or manually-confirmed losses. For 135 apps, the manual analysis could not run or did not produce traces evidencing losses. These apps fell into several categories: 58 apps could not be explored as they either required a paid membership, their operation was geo-fenced, or could only run when connected to specific hardware devices; 39 apps crashed on our test platform; finally, there were 38 apps where no save- or export-related option was found in the GUI.

B. Example Of Confirmed Write Loss Cases

Table IV summarizes data loss examples in 27 apps: 20 apps from Google Play (apps with highest number of installs, shown in the second column) and 7 apps from the open-source F-Droid store. We show a brief summary of the user file write data lost in the third column and the result of running LiveDroid on these apps in the final column (the results are discussed in Section IV-C2). We now discuss selected apps (more than 5M installations) and the semantics of lost data in detail.

SketchBook. This app allows users to sketch, paint, and draw; due to termination, new sketch data, as well as changes to an existing sketch, can be lost.

Smart TV Remote. This app is used to define and control TV channels via channel logos; due to termination, exported data (TV channels) is lost.

WPS Office. This is an all-in-one office suite app; due to termination, user-made document changes are lost.

Drum Pad Machine. This music mixer app can be used to create beats, mix loops and record new melodies; due to termination, user-created music beats are not saved.

AndrOpen Office. This office suite app allows users to view and edit PDF, Word, Excel, and PowerPoint documents; due to termination, user-made document changes are lost.

King James Bible. This Bible reader app provides options for adding bookmarks and writing notes while reading; due to termination, user settings or preferences (e.g., related to bookmarks, highlights, and notes) can be lost.

K-9 Mail. This is an open-source email client app; due to termination, exported data (user-settings backup) is lost.

Beauty Camera. This app allows editing pictures via filters or stickers; due to termination, edited pictures are not saved.

Barcode Scanner Pro. In this app the user can scan, decode, create, and share QR codes or barcodes; due to termination, the user's barcode scanning history is lost.

C. Comparison With Existing Tools

We now compare the results of our approach with the results obtained by running two state-of-the-art tools that aim to find volatility-induced UI losses in Android apps.

1) Comparison with KREfinder: KREfinder [6] is a static analyzer that looks for incorrectly-handled instance state. Specifically, the analysis looks for app fields that are written to, or modified, and for which there is no subsequent save. KREfinder explicitly looks for state flowing into OutputStream or Writer objects, and generally any Java API methods offering write or save. As the public version of KREfinder is not maintained/updated (latest release: July 2016), we asked the KREfinder's corresponding author to run it on 14 selected apps (7 top Google Play apps, 7 F-Droid apps); KREFinder reported no data losses.

2) Comparison with LiveDroid: LiveDroid [7] is a tool focused on finding UI fields that might be lost during runtime changes (e.g., phone orientation changes). LiveDroid identifies variables and GUI input which represent "necessary app state"; this state essentially captures the subset of user input data which must "survive" runtime changes. We ran LiveDroid on the 298 apps with user-initiated file writes. The LiveDroid analysis summary is:

	# Apps
Analyzed	298
Soot Error	157
Issues Found	13

LiveDroid is mainly designed for open-source apps and fails due to Soot error on the 157 Google Play apps with large and/or obfuscated bytecode. For the remaining Google Play apps and all the open-source F-Droid apps, LiveDroid ran to completion; LiveDroid found app state-saving related issues in 13 apps. For example, LiveDroid found app states saving related issues in Beauty Camera and Privacyfriendlynotes app as shown in the third column of Table IV, but these issues are unrelated to file write losses; LiveDroid reported no issues in the other 128 apps it successfully run on.

D. Efficiency

In Table V we present brief descriptive statistics for static analysis time and app dataset. Analysis time varied between

TABLE IV
LOSSES FOUND AND CONFIRMED BY OUR APPROACH; RESULTS OF RUNNING LIVEDROID.

App	#Installs (Million)	Our Approach (Data Lost)	LiveDroid
Google Play			
SketchBook	100	User artwork (sketch, painting)	Failed due to Soot error
WPS Office	100	User-made document changes	No issues found
Drum Pad Machine	100	User created music beats	No issues found
Smart TV Remote	10	Saved TV Channels export	No issues found
Barcode Scanner Pro	10	Barcode scanning history	Failed due to Soot error
Beauty Camera	10	Edited photos	Found 5 app states not saved (user input loss)
AndrOpen Office	5	User-made document changes	No issues found
King James Bible	5	User-settings / preferences	No issues found
K-9 Mail	5	User-settings / preferences	No issues found
Robin	1	App properties/preferences	Failed due to Soot error
Soccer Tactic Board	1	User-created soccer tactic	Failed due to Soot error
Baby Care	1	User-created baby growth data	Failed due to Soot error
Bills Reminder	0.5	Database backup	Failed due to Soot error
SmartTruckRoute	0.5	Truck route exported data	Failed due to Soot error
BCBSM	0.1	Patient's medicare data sharing fails	Failed due to Soot error
Wabbitemu	0.1	Calculator screenshot	Failed due to Soot error
Gallery Slideshow Music	0.1	Edited Video	Failed due to Soot error
TV Show Favs	0.1	User backup data (e.g., favorite TV, watched shows)	Failed due to Soot error
User Dictionary Manager	0.05	Dictionary words	Failed due to Soot error
Bahamas Dining Rewards	0.01	User credentials	Failed due to Soot error
F-Droid			
Sanity	n/a	Settings/preferences (e.g., audio recording, call blocking)	No issues found
Privacyfriendlynotes	n/a	New or updated note	Found 6 app states not saved (user input loss)
MedicLog	n/a	Medic log history	No issues found
Acrylic Paint	n/a	New or updated drawing	No issues found
Auto-Away	n/a	Call or message log export	No issues found
ComfortReader	n/a	New or updated note	No issues found
BeeCount	n/a	database table update	No issues found

TABLE V
EFFICIENCY RESULTS.

Analysis time (seconds)			Bytecode size (MB)		
min	max	median	min	max	median
35	25,200	115	0.04	103.4	21.2

35 seconds and 7 hours, with a typical time of 115 seconds, which we believe is efficient for a static analysis. App bytecode varied between 40KB and 103MB, with a typical size of 21MB, which shows that our analysis can handle sizable apps.

E. False Positives and False Negatives

We measured the False Positives (FP) and False Negatives (FN) by comparing the results of our automated approach with a manual analysis on 60 apps (all containing file writes) where the file write data losses were confirmed manually. The 60 apps were selected as follows: 30 true positive apps that contain user-initiated file writes and 30 true negative apps that contain file writes, but the writes are not user-initiated. Rather than exploring apps using DroidBot and performing dynamic verification of user data loss via `Strace` difference checking as mentioned in Section III-B, we performed a manual verification of data losses. We manually sent the target app into the background after inducing file write operations and then manually checked whether the expected file writes were missing, i.e., user data is lost. The confusion matrix is:

True Positives: 30	**False Positives**: 0
False Negatives: 5	**True Negatives**: 30

We found 5 False Negatives, i.e., an 85% recall for the automated approach. These are due to automated GUI exploration with DroidBot failing to reach targeted save or export-related GUI options and as a consequence, no file write operation happened, and no file write traces were found. We have no False Positives because static analysis reports are subjected to dynamic verification.

F. Limitations

Our approach has two limitations. First, the static analysis to find user-initiated file writes is implemented on top of Soot, which failed to produce call graphs for 733 apps. As a result, we were unable to analyze those apps further. Second, our automated dynamic verification used a customized version of the DroidBot input generator [15] to drive app interaction. However, DroidBot, and Android input generators in general, cannot achieve complete coverage [20]. This shortcoming led to manually exploring cases where DroidBot failed to verify user data losses. These limitations can be alleviated with more engineering efforts.

G. Potential Solutions

File write losses due to termination could be addressed by keeping the target app alive as a foreground process. The

Fig. 9. Progress bar for ongoing I/O operations in Android.

app developers could show a progress bar for file writes (e.g., Figure 9) so users do not switch to a different app while writes are in progress. Another solution is to keep the app alive as a background process by granting unrestricted battery usage (available for Android 8.0 or above), which ensures the app is running with fewer limits while in background, hence is less likely to be killed due to memory pressure. Finally, another option for updating data is to write a temporary copy and delete the old data upon successful writing of the new data, or alternatively, use storage with transactional APIs such as SQLite or Firebase.

V. Related Work

Android state volatility has been the subject of several prior efforts. However, none of the prior efforts has looked at system-initiated termination or file write operation.

LiveDroid [7] focuses on the issue of finding UI fields that might be lost during runtime changes. LiveDroid's static analysis employs a top-down approach from program variables and UI input instances to the part of saving these inputs into the Android Bundle (the default storage where Android apps can save instance state), whereas we take a bottom-up approach to trace back to UI inputs from file write APIs. However, file writes leading from user interactions are not considered as user data losses in their analysis. LiveDroid has a patching component that injects state-saving code into an app o fix state-saving issues; we do not offer an error repair component. LiveDroid handles, and was run on, F-Droid apps only; in contrast, we successfully analyzed thousands of Google Play apps in addition to F-Droid apps.

iFixDataloss [21] is similar to LiveDroid, detecting and fixing data losses due to Android lifecycle events (e.g., orientation change, back button press). Unlike LiveDroid, their approach is not limited to data losses in a singular instance of an app run, as they also detect and fix data loss issues across multiple runs. Like our approach, iFixDataloss has a reduced false positives rate as they combine static analysis with dynamic testing. However, they do not consider data losses due to system-initiated termination.

KREfinder [6] used program analysis to identify object fields that should be saved during resume-and-restart cycles to avoid user data loss. However, their technique is focused more on finding a path from a field write to an app exit without an intervening save (e.g., in the Android Bundle) rather than finding lost file writes due to system-initiated termination.

The work of Hu et al. [22], Zaeem et al. [23], and Adamsen et al. [24] focused on finding app state issues related to activity restart by generating test cases and performing systematic execution of event sequences. Our goal (lost user file writes) is different; in addition, our approach is based on static analysis whereas their approach is based on testing.

SafeExit [25] is a study on ungraceful exits in desktop applications. They propose cleanup operations on file writes that are interrupted by an ungraceful exit in order to match the program behavior to that of normal execution. However, SafeExit did not categorize file writes based on user interaction, and did not consider user data loss.

Several prior efforts have studied the issue of whether an API call was in response to a user action.

Huang et al.'s AsDroid [26] identifies user interactions that result in malicious behavior differing from the intended purpose. They perform reachability analysis of specific APIs from top-level user interaction functions via control-flow-graph and call-graph analysis and compare the result with the intended behavior described in the UI text. Although our analysis includes control-flow analysis, we do not perform UI text analysis to determine file-write-related UI interactions. In our dynamic verification, we have not found any evidence of the UI element's text description representing a file write operation. Rather, we rely on a combination of control-flow and data-flow analysis to identify file write-related user interactions.

Shan et al.'s work on detecting self-hiding behavior [27] introduced a user decision analysis to understand whether an API call performing questionable behavior in an Android app was initiated by the user or performed by the system. Their focus is mostly on discerning between different UI elements (some are considered decision-related, some not); their analysis considers control-flow only, rather than the intricate interplay between control-flow and data-flow analyses required in our case to detect user-initiated data losses.

Enforcer [28] tests application behavior in the presence of I/O failures by executing tests with and without fault injection. Our approach differs in two ways. First, Enforcer identifies all I/O types and instruments the code to simulate faulty system calls, while we focus on user-initiated I/O and instrument the code to simulate system-initiated termination. Second, Enforcer was evaluated on Java programs, including the Apache Commons Math library, whereas we target Google Play apps.

VI. Conclusions

Mobile apps' construction and operation is fundamentally different from "run forever" desktop/server programs which complicates testing whether user data is inadvertently lost due to resource pressure. In this paper, we focus on user work/data that should be saved via user-initiated file writes; while expected to be stored in local storage, such work and data can be lost due to system-initiated termination. We constructed a static analysis to find potential losses in users' work due to premature file write termination and verified losses via an automated dynamic approach. We were able to confirm such losses in 107 Google Play and F-Droid apps. Our approach can improve the overall user experience of saving user data and form the basis of further studies and explorations into (a) loss of mobile state due to volatility, (b) extending the findings from file writes to all possible I/O, and (c) the nature of losses due to unexecuted I/O in programs in general.

REFERENCES

[1] Memory allocation among processes, "Memory allocation among processes." April 2022, https://developer.android.com/topic/performance/memory-management.

[2] Android Open Source Project, "Low memory killer daemon," May 2022, https://source.android.com/devices/tech/perf/lmkd.

[3] Background Execution Limits, "Background execution limits." April 2022, https://developer.android.com/about/versions/oreo/background.

[4] N. Lebeck, A. Krishnamurthy, H. M. Levy, and I. Zhang, "End the senseless killing: Improving memory management for mobile operating systems," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 873–887.

[5] M. Xia, W. He, X. Liu, and J. Liu, "Why application errors drain battery easily? a study of memory leaks in smartphone apps," in *Proceedings of the Workshop on Power-Aware Computing and Systems*, ser. HotPower '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2525526.2525846

[6] Z. Shan, T. Azim, and I. Neamtiu, "Finding resume and restart errors in android applications," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 864–880. [Online]. Available: https://doi.org/10.1145/2983990.2984011

[7] U. Farooq, Z. Zhao, M. Sridharan, and I. Neamtiu, "Livedroid: Identifying and preserving mobile app state in volatile runtime environments," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428228

[8] Threading in Android, "Better performance through threading — android developers," April 2022, https://developer.android.com/topic/performance/threads.

[9] Apple, Inc, "Swift dispatchqueue," May 2022, https://developer.apple.com/documentation/dispatch/dispatchqueue.

[10] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 29–42. [Online]. Available: https://doi.org/10.1145/2168836.2168841

[11] Acrylic Paint, "Acrylic paint — f-droid - free and open source android app repository," April 2022, https://f-droid.org/en/packages/anupam.acrylic/.

[12] Wabbitemu, "Wabbitemu," April 2022, http://wabbitemu.org/.

[13] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 259–269. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594299

[14] Soot.2022, "Soot: a java optimization framework." April 2022, https://www.sable.mcgill.ca/soot/.

[15] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: A lightweight ui-guided test input generator for android," in *Proceedings of the 39th International Conference on Software Engineering Companion*, ser. ICSE-C '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 23–26. [Online]. Available: https://doi.org/10.1109/ICSE-C.2017.8

[16] I. A. Qazi, Z. A. Qazi, T. A. Benson, G. Murtaza, E. Latif, A. Manan, and A. Tariq, "Mobile web browsing under memory pressure," *SIGCOMM Comput. Commun. Rev.*, vol. 50, no. 4, p. 35–48, oct 2020. [Online]. Available: https://doi.org/10.1145/3431832.3431837

[17] G. Aponso, "Effective memory management for mobile operating systems," *American Journal of Engineering Research (AJER)*, vol. 246, 2017.

[18] P. Patel, G. Srinivasan, S. Rahaman, and I. Neamtiu, "On the effectiveness of random testing for android: Or how i learned to stop worrying and love the monkey," in *Proceedings of the 13th International Workshop on Automation of Software Test*, ser. AST '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 34–37. [Online]. Available: https://doi.org/10.1145/3194733.3194742

[19] Strace, "Using strace — android open source project," April 2022, https://source.android.com/devices/tech/debug/strace.

[20] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 429–440. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.89

[21] W. Guo, Z. Dong, L. Shen, W. Tian, T. Su, and X. Peng, "Detecting and fixing data loss issues in android apps," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 605–616. [Online]. Available: https://doi.org/10.1145/3533767.3534402

[22] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, effectively detecting mobile app bugs with appdoctor," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2592798.2592813

[23] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of mobile apps," in *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ser. ICST '14. USA: IEEE Computer Society, 2014, p. 183–192. [Online]. Available: https://doi.org/10.1109/ICST.2014.31

[24] C. Q. Adamsen, G. Mezzetti, and A. Møller, "Systematic execution of android test suites in adverse conditions," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 83–93. [Online]. Available: https://doi.org/10.1145/2771783.2771786

[25] Z. Jia, S. Li, T. Yu, X. Liao, and J. Wang, "Automatically detecting missing cleanup for ungraceful exits," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 751–762. [Online]. Available: https://doi.org/10.1145/3338906.3338938

[26] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 1036–1046. [Online]. Available: https://doi.org/10.1145/2568225.2568301

[27] Z. Shan, I. Neamtiu, and R. Samuel, "Self-hiding behavior in android apps: Detection and characterization," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 728–739. [Online]. Available: https://doi.org/10.1145/3180155.3180214

[28] C. Artho, A. Biere, and S. Honiden, "Exhaustive testing of exception handlers with enforcer," in *Formal Methods for Components and Objects*, 2006.