

ANCHOR: Locating Android Framework-specific Crashing Faults

Pingfan Kong
University of Luxembourg
Luxembourg
pingfan.kong@uni.lu

Li Li*
Monash University
Australia
li.li@monash.edu

Jun Gao
University of Luxembourg
Luxembourg
jun.gao@uni.lu

Timothée Riom
University of Luxembourg
Luxembourg
timothee.riom@uni.lu

Yanjie Zhao
Monash University
Australia
yanjie.zhao@monash.edu

Tegawendé F. Bissyandé
University of Luxembourg
Luxembourg
tegawende.bissyande@uni.lu

Jacques Klein
University of Luxembourg
Luxembourg
jacques.klein@uni.lu

ABSTRACT

Android framework-specific app crashes are hard to debug. Indeed, the callback-based event-driven mechanism of Android challenges crash localization techniques that are developed for traditional Java programs. The key challenge stems from the fact that the buggy code location may not even be listed within the stack trace. For example, our empirical study on 500 framework-specific crashes from an open benchmark has revealed that 37 percent of the crash types are related to bugs that are outside the stack traces. Moreover, Android programs are a mixture of code and extra-code artifacts such as the Manifest file. The fact that any artifact can lead to failures in the app execution creates the need to position the localization target beyond the code realm. In this paper, we propose ANCHOR, a two-phase suspicious bug location suggestion tool. ANCHOR specializes in finding crash-inducing bugs outside the stack trace. ANCHOR is lightweight and source code independent since it only requires the crash message and the apk file to locate the fault. Experimental results, collected via cross-validation and in-the-wild dataset evaluation, show that ANCHOR is effective in locating Android framework-specific crashing faults.

KEYWORDS

Android Crash, Crashing Fault, Fault Localization.

1 INTRODUCTION

App crashes are a recurrent phenomenon in the Android ecosystem [1]. They generally cause damages to the app reputation and beyond that to the provider's brand [2]. Apps with too many crashes can even be simply uninstalled by annoyed users. They could also receive bad reviews which limit their adoption by new users. Too many apps crashes could also be detrimental to specific app markets that do not provide mechanisms to filter out low-quality apps concerning proneness to crash. The challenges of addressing Android app crashes have attracted attention in the research community.

Fan et al. [3] have recently presented insights on their large-scale study on framework-specific exceptions raised by open source apps.

In more recent work, Kong et al. [4] have proposed an automated approach to mine fix patterns from the evolution of closed-source apps (despite the lack of change tracking systems). Tan et al. [5] further presented an approach to repair Android crashing apps. A common trait of all these crash-related studies is that the underlying approaches heavily rely on the generated stack traces to identify the fault locations. Although the state of the art is effective for many bugs, they are generally tailored to the generic cases where the stack traces provide relevant information for locating the bug. Unfortunately, there is a fair share of faults whose root causes may remain invisible outside the stack trace. Wu et al. [6] have already reported this issue in their tentative to locate crashing faults for general-purpose software. In the realm of Android, the phenomenon where the stack trace may be irrelevant for fault localization is exacerbated by two specificities of Android:

The Android system is supported by a callback-based and event-driven mechanism: Each component in Android has its lifecycle and is managed by a set of callbacks. Every callback serves as a standalone entry point and root to a separate call graph. Yet, existing crash-inducing bug localization techniques for Java such as CrashLocator [6] assume a single entry point to compute certain metrics for the suspiciousness score of different methods. Additionally, since the Android system is event-driven, the invocation sequence to functions and callbacks is affected by non-deterministic user inputs or system events, making the stack trace unreliable for quick analyses.

The Android app package includes both code and resources that together form the program: Android apps are more than just code. They are combinations of Java/Kotlin code, XML files, and resources (such as images and databases). Apps provide extensions to the Android Operating System (OS), which directly analyses XML files from the app to map callback functions, which the OS must trigger to exploit functionalities in the apps. Therefore, an error by developers within an XML document can eventually lead to a runtime crash. Similarly, it is important to note that crashes can occur due to other concerns such as the arrangements of app resources, use of deprecated APIs, omissions in permission requests, etc. Typical such errors, which occur outside of code pointed to by stack traces,

*Corresponding author.

will cause either developers or Automatic Program Repair (APR) tools (e.g., [5]) to pointlessly devote time in attempting to fix the code.

This paper. Our work aims at informing the research community on the acute challenges of debugging framework-specific crashes. To that end, we propose to perform an empirical study that investigates the share of crashes that cannot be located by current localization approaches. Following this study, we present a new approach to locate faults, aiming at covering different categories of root cause locations. Overall, we make the following contributions:

- We present the results of an empirical study performed on a set of 500 app crashes retrieved from the ReCBench dataset [4]. A key finding in this study is that we were able to identify that 37% crash root causes are associated with crash cases where the stack trace is not directly relevant for fault localization.
- We propose ANCHOR, a tool-supported approach for locating crashing faults. ANCHOR unfolds in two phases and eventually yields a ranked list of location candidates. The first phase applies a classification algorithm to categorize each new crash into a specific category. Depending on this category, a dedicated localization algorithm is developed in the second phase. ANCHOR currently implements 3 localization algorithms that eventually generate a ranked list of buggy methods (when the bug is in the code) or resource types (when it is outside of code).
- We performed 5-fold cross-validation on the 500 crash cases to assess the effectiveness of ANCHOR in placing the crashing fault location in the top of its ranked list of suggestions. ANCHOR exhibited an overall MRR (Mean Reciprocal Rank) metric value of 0.85. An analysis of the open dataset of crashed open-source Android apps further shows that our method scales to new app crashes.

The rest of this paper is organized as follows. Section 2 introduces background details on Android app crashes and callback-based event-driven mechanisms. Section 3 revisits the motivating example by the previous work [5] and demonstrates why research in crash localization has standing challenges. Section 4 discusses the findings of our empirical study and explores the insights that can be leveraged for a new approach. Section 5 presents ANCHOR. We describe experimental setup in Section 6 and approach evaluation in Section 7. We bring further discussion in Section 8. Threats to validity are acknowledged in Section 9 and related work is presented in Section 10. Finally, Section 11 concludes the paper.

2 BACKGROUND

In this section, we introduce the important concepts related to this paper.

2.1 Android App Crash Stack Trace

Like all Java¹ based software, when Android apps crash, they can dump execution traces which include the exception being thrown, a crash message, and most importantly, a stack trace of a callee-caller chain starting from the *Signaler*, i.e., the method that initially

¹Kotlin has also been widely used in recent years as an alternative for Android app development, it is designed to fully interoperate with Java.

constructed and threw the exception object. Figure 1 is an example of stack trace for the crash of the app *Sailer’s Log Book*. This app helps sailors to keep their logbook accurate and up-to-date. On the first line, the exception *IllegalArgumentException* is thrown. On the second line, the log system reports message "recursive entry to executePendingTransactions". Starting from the third line, the *Signaler* of the stack trace is listed: it is this framework method that instantiates the exception type, composes the log message and throws it to its caller to handle. On Lines 4-5 that are also marked in grey, there are other two methods that continue to pass on the exception. Line 5 holds the *API*, which is the only framework method in this stack trace that is visible to the developer. Since the crash happens directly due to invocation to it, we call it the *Crash API*. Line 6 is the developer method that invoked this API. Line 7 is the developer implementation of the callback, inherited from the superclass of the Android framework. Android framework decides, based on certain conditions and system/user events, when to invoke this method, and what parameter values to pass in. Lines 8-9 are part of the Android framework core that is, again, not accessible to developers.

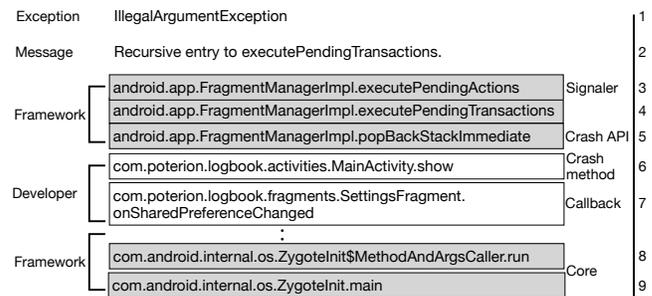


Figure 1: Crash Stack Trace of app Sailer’s Log Book.

The crash stack trace is often the first thing that developers want to examine when a crash is reported [7]. Even when it is not given, developers would reproduce and retrieve them. Intuitively, the crash arises from mistakes in the developer methods, e.g., Lines 6-7 in Figure 1. Particularly, the *Crash method* that directly invoked the *Crash API*. Our empirical study in Section 4 shows that this intuition is correct, that 63% of the total crash types are in the stack trace. However, in the rest of this section, we will introduce the specialty of Android that may lead to the rest 37%.

2.2 Callback-based and Event-driven Mechanism

Unlike traditional Java programs, Android apps have multiple entry points. Each entry point is a callback method (e.g., Line 7 in Figure 1), which is declared in one of the Android framework component classes, inherited by the developer defined subclass, and maybe overridden by the developer. The Android framework core, based on the user inputs and system environments, decides when to invoke the callbacks and what parameter values to pass in. Each callback is standalone, and in general Android does not encourage developers to invoke those callbacks from their self-defined methods, unless these methods are callbacks overriding and invoking

their super. As a result, existing static call graph based fault localization techniques [6] for Java programs can not be simply reused, since they assume single entry points and need to compute weighing scores based on the graph. There are, however, works [8, 9] that have invented methods to track the control flows or data flows and tried to build the callback connections. These proposed approaches are either computationally expensive or confined in limited number of component classes, and does not scale to all scenarios. Other approaches like [10] or [11] create a dummy main to invoke all callbacks in order to reuse Java based analysis tools, but this method discarded the relation among callbacks, which is crucial to estimate the possibility of a method containing the real bug.

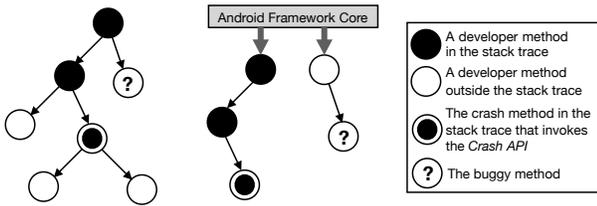


Figure 2: Call Graph Comparison between General Java Program (left) and Android App (right), inspired from [6]

Figure 2 exemplifies the difference of call graphs between general Java program (left) and Android app (right). The filled circles represent the developer methods in the stack trace, while the non-filled circles represent developer methods outside the stack trace. The partially filled circles represent the *Crash method* that invokes the *Crash API*. Generally, the buggy method is the *Crash method*. However, as shown in our empirical study, it appears that the buggy method (the circle filled with question mark in Figure 2) is not connected to the *Crash method*. A traditional Java program static call graph based approach such as CrashLocator [6] will be able to locate this buggy method only if the buggy method is "close enough" to the generated call graph (roughly speaking they generate an extended call graph leveraging the stack trace). However, on the right, in the case of Android apps, the buggy method could be in a separate call graph because of callback methods that are invoked by the Android framework. Such cases will be missed by approaches such as CrashLocator [6] that only detects buggy methods captured by its extended call graph, but does not consider callback methods.

2.3 Android APK File Format

Android apps are distributed in a package file format with extension ".apk". It is a compressed folder containing code, resources, assets, certificates, and manifest file. All of these files are crucial to the expected good functioning of the apps. Therefore, some crashes may be induced when there are problems with these files.

2.3.1 *Android Manifest File*. Every app project needs to have an AndroidManifest.xml file at the root of the project source set [12]. Along with the package name and components of the app, this file also declares the permissions that the apps needs, as well as the hardware and software features that the app requires.

2.3.2 *Android Component Layout Description File*. Android component layout description files are also crucial to the execution of the app. E.g., Listing 1 is the layout file of the main Activity of an Android app *Transistor*. In this file, a child fragment is defined and described. The attribute *android:id* defines the layout file name to be inflated for the fragment, the attribute *android:name* gives the full name of the user defined Fragment class. When the main Activity is being created, the Android framework scans this layout file, and invokes a series of relevant callbacks on this Fragment to draw it along with the main Activity.

Listing 1: Main Activity Layout File of app *Transistor*.

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/fragment_main"
  android:name="org.y20k.transistor.MainActivityFragment"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:layout="@layout/fragment_main" />
```

3 MOTIVATING EXAMPLE

We further illustrate the challenges of locating faults outside Android app stack traces by revisiting an example that was used to motivate a previous work on Android app crash automatic repairing by Tan et al. [5]. *Transistor*² is a popular online radio streaming app. We showed its partial resources in Section 2.3.2. However, it was reported that following the input sequence in Figure 3, the app will crash.

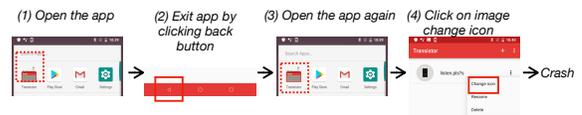


Figure 3: Crash of *Transistor*.

Listing 2: Crash Message of *Transistor*.

```
1 java.lang.IllegalStateException:
2 MainActivityFragment(e7db358) not attached to Activity
3 at ...MainActivityFragment.startActivityForResult(Fragment.java:925) (Crash API)
4 at ...agment.selectFromImagePicker(MainActivityFragment.java:482) (Crash method)
5 at ...k.transistor.MainActivityFragment.access$500(MainActivityFragment.java:58)
6 at ...transistor.MainActivityFragment$6.onReceive(MainActivityFragment.java:415)
```

The crash message filtered out from logcat is shown in Listing 2. It appears that invoking the *startActivityForResult* API on the *MainActivityFragment* (line 3) throws an unhandled *IllegalStateException* (line 1), and the Android system reports that the fragment is not attached to the hosting activity (line 2). By inspecting the source code of Android framework of the *Crashed API* (line 3), we see that the *startActivityForResult* method of the fragment instance attempts to invoke its context's (i.e., its host Activity's) API with the same name *startActivityForResult*. This invocation is guarded by an if-clause, which checks whether the fragment is still attached to the host Activity. If not, however, the *IllegalStateException* will be thrown.

²<https://github.com/y20k/transistor/issues/21>

Listing 3: Fix from Tan et al.

```
new BroadcastReceiver(){
  onReceive(...){ ...
+  if(getActivity()!=null)
    startActivityResult(pickImageIntent,REQUEST_LOAD_IMAGE);}}
```

Biased by the assumption that the fault should only be in the developer methods in the stack trace (lines 4-6), Tan et al. [5] proposed to amend the *Crash method* (line 4). Listing 3 shows their fix. Their fix applies a checker on invocation to *startActivityResult*, which will not be executed if value of *getActivity* is null (i.e., when the fragment is no longer attached to its hosting Activity). As a result, the app avoids crashing. This fix indeed prevents the exception. However, it is not explainable: applying the checker not only prevents the crash, but it should also prevent opening the *SelectImageActivity* as designed for. Due to this paradox, we have a good reason to suspect that the true bug location is still hidden.

Transistor's developer, who is also dedicated in debugging in the stack trace, proposed a fix on her/his own in Listing 4. Realizing that the Fragment lost its reference to the host Activity. The developer declared a variable *mActivity* to hold the reference. Then in the *Crash method* (line 4 in Listing 2), she/he switched the invocation of the *startActivityResult API* from Fragment to *mActivity*.

Listing 4: Fix from Developer.

```
+ mActivity = getActivity(); ...
new BroadcastReceiver(){
  onReceive(...){ ...
-  startActivityResult(pickImageIntent,REQUEST_LOAD_IMAGE);
+  mActivity.startActivityResult(pickImageIntent,
    REQUEST_LOAD_IMAGE);}}
```

This fix also bypassed the crash, but it causes regression. After the final step in Figure 3, if the user clicks on the back button two more times, the app should have first returned to the MainActivity, then back to the home screen. Instead, it opens another *SelectImageActivity*. In the issue tracking, the developer admits that she/he had no idea of how to fix it. While after several months, the bug "fixed" itself, which she/he described as "scary". Even Tan et al. failed to explain the cause of this regression.

Based on the understanding of Android' callback-based mechanism introduced in Section 2.2, we suspect that the bug may not exist in the stack trace. We confirmed our fix shown in Listing 5. This fix is reported to the developer and we received positive feedback in the issue tracking, as can be verified in *Transistor's* repository given above.

Listing 5: Fix Inspired by this Article.

```
MainActivityFragment extends Fragment{
  onDestroy(){
+  super.onDestroy();
+  LocalBroadcastManager.getInstance(mApplication).
    unregisterReceiver(imageChangeRequestReceiver,
    imageChangeRequestIntentFilter);}}
```

We broaden the search for bug outside the stack trace. Noticing the crash originated from the *onReceive* callback (cf. line 6 in

Listing 2), we examine the lifecycle of this BroadcastReceiver object. We found that it is registered in the *onCreate* callback of MainActivityFragment, but never unregistered in its counterpart callback *onDestroy*. As a result, after Step 2 (cf. Figure 3), the BroadcastReceiver and its host MainActivityFragment are leaked in the memory. In Step 4, the callbacks of the leaked objects are stealthily invoked by the Android framework and eventually caused the *IllegalStateException*. Knowing the true cause of the crash, it is not difficult to explain the paradox of Tan et al.'s fix and the regression caused by the developer's fix. However, given the page limit, we put detailed reasoning online at <https://anchor-locator.github.io>.

Hint: The fault locations in Android apps may: (1) Be outside the stack trace; (2) Be even outside the call graph extended from the stack trace; (3) Not even "exist" in the code, i.e., they are inherited methods without visible code snippets. Locating such faults may require tremendous efforts. Fixes based on incorrect localization may even cause regression.

4 EMPIRICAL STUDY ON FAULT LOCATIONS

In this section, we present the results of an empirical study that we performed on a set of 500 app crashes retrieved from the ReCBench dataset [4]. This study aims at assessing to what extent the locations of crashing faults reside outside the stack trace.

4.1 Dataset Construction

We extract our dataset from ReCBench, an open dataset proposed by Kong et al. [4] in 2019. ReCBench has been built by running hundreds of thousands of Android apps downloaded from various well-known Android markets [13, 14]. In addition to a collection of crashed Android apps focusing on framework-specific crashes³, ReCBench offers the possibility to collect crash log messages and scripts to reproduce the crashes. Today, ReCBench contains more than 1000 crashed apps (still growing). For our empirical study, we focus on crashed apps for which:

- First, the stack trace of the crash contains at least one developer method. This is a requirement to be able to start an exploration process to find the crash root cause.
- Second, since we specifically target the crashes induced by Android APIs, the *Signaler* must be Android-specific.

After applying these two rules, we randomly selected 500 crashed apps from the remaining apps. The dataset is publicly accessible at <https://github.com/anchor-locator/anchor>.

4.2 Ground Truth & Results

We manually inspect all the 500 crashed apps to understand the reason behind the crashes and to create our ground truth. We perform this manual inspection by leveraging the CodeInspect [15] tool, following same protocols discussed in [3]. Each of the crashed apps has been categorized into one of the following categories:

³Android framework methods are not visible or understandable to general developers, hence greater challenge is acknowledged for locating framework-specific crashes compared to developer-written methods. [3, 4]

- Category A groups the crashed apps for which the buggy method (i.e., the root cause) is one of the developer methods present in the stack trace;
- Category B groups the crashed apps for which the buggy method is not present in the stack trace, but still in the code.
- Category C groups the crashed apps for which the crash arises from non-code reasons.

The above partition is one out of many alternatives, e.g., one can also separate bugs based on whether they are concurrent [16–20]. We later show in Section 5.2 how this partition helps with building our localization tool. Table 1 summarizes the outcome of the empirical study. It appears that for 89 (49+40) crashed apps (representing 18% of the total cases), the crashing fault location is not in any of the developer methods present in the stack trace. The respective numbers of Categories B and C are close, with 49 cases in Category B and 40 cases in Category C. To further investigate how many types of crashes occur for each category, we use a method similar to the one described in [4]: we group crashes from a given category into *buckets*. Specifically, if two crash cases have identical framework crash sub-trace, they will be put into the same bucket. The last two columns in Table 1 present the number of buckets per category. Overall, there are 105 types of crashes (i.e., buckets) in the dataset. The percentage of types of crashes in Categories B and C are 16% and 21%, respectively. In total, there are 37% of buckets whose buggy reasons are not shown in the stack traces. Each unique framework crash sub-trace suggests a unique type of crash-inducing bug. Therefore, considering crash types encountered per the same number of cases (buckets#/case#) in each category, more debugging effort will be needed for Categories B and C than in Category A.

Table 1: Categories of Fault Locations in Android apps

Category	stack trace	code	case#	percent	bucket#	percent
A	in	yes	411	82%	66	63%
B	out	yes	49	10%	17	16%
C	out	no	40	8%	22	21%
Total	-	-	500	100%	105	100%

Hint: 18% of the crashes are due to bugs for which the location is outside the stack trace. A significant number of root causes (buckets), i.e., 37% (16%+21%), are associated with cases where the stack trace is not directly relevant for localization. In even 21% of the cases, the root causes are not located in the code.

We now detail each category in the rest of this Section.

4.3 Category A: in Stack Trace

Category A includes all crash cases whose bugs reside in one of the developer methods present in the stack trace. Most crashes in our dataset fall into this category. It is expected that by default, developers start their debugging process with the methods present in the stack trace [21–24]. The automatic crash-inducing bug repairing tool named Droix [5] implements a locator, by assuming that the *Crash method* is the bug location in all scenarios. However, we also notice that the true crashing fault may reside in other developer methods, in particular when moving downward in the stack trace. An example of such a case is when the caller methods pass

an incorrect instance to the crashed developer methods. Generally, much less effort is needed in locating faults in this category. Since the number of suspected methods is limited and their names are already known. Therefore they are not the focus of this paper.

4.4 Category B: out of Stack Trace, in the Code

It has drawn attention to researchers that Java program crashes can be caused by methods that are not listed in stack traces. Approaches like CrashLocator [6] broadens the search for such faulty methods in extended call graphs from stack traces. We demonstrate in the rest of this section why this broadened search is not enough for Android apps. There are in total 49 cases in this category, each crashed from wrongly handling a framework *API*. Based on the type of the framework *API* (call-in or callback), we further categorize them into two sub-categories: (1) Misused Call-In APIs and (2) Non-Overridden Callback APIs.

4.4.1 Type 1: Misused Call-In APIs (44 cases out of 49). This first type groups crashing faults caused by the misuse of call-in APIs. This means that the bug leading to a crash is due to a buggy explicit invocation of an API from a developer method. Moreover, this invocation is often performed from another implemented callback, other than the callback in the stack trace. Since both callback methods are triggered by the framework, it is unlikely that an extended call graph can cover such methods (cf. Figure 2).

Listing 6: Bug Explanation to app Geography Learning.

```
public class MainActivity extends Activity{
    onCreate(...){
        try{bindService(intent,serviceConnection,integer);/*Bug Location*/
        }...}...
    onDestroy(){unbindService(serviceConnection);/*Crash location*/}
```

Listing 6 gives a concrete example. This example is extracted from an app named *Geography Learning* which helps users to remember geography knowledge in a quiz game format. When the *MainActivity*⁴ of this app is launched, the callback method *onCreate* is automatically triggered by the Android framework. Then, this *onCreate* method invokes the *bindService* API to bind to *Service*. *Service* is one of the four basic components of Android, and wrongly handling of *Service* is not uncommon [25] in Android app development. When the user exits the *MainActivity*, the Android Framework invokes the *onDestroy* callback method and tries to unbind the *Service* bound in the *onCreate* method. Thereafter, the app crashes with the exception type *IllegalArgumentException*. Analysing the message which says: "Service not registered: com.yamlearning.geographylearning.e.a.e@29f6021", we understand that the *Service* has not been bound. In the method body of the overridden *onCreate* callback, we found that the invocation to API *bindService* was misused. Indeed, *bindService* is surrounded by a try-catch clause, and another statement preceding this API invocation threw an exception which redirects the execution flow to the catch block, skipping the invocation to *bindService*.

Out of a total of 49 cases in Category B, 44 falls into this sub-category.

⁴The *Main Activity* of an app is the first screen shown to the user when launched.

4.4.2 *Type 2: Non-Overridden Callback APIs (5 cases out of 49)*. This second type includes crashes caused by non-overridden callback APIs. Callbacks, or call-afters, are APIs that are invoked when the Android framework decides so, based on certain system environment change and/or user actions. Callbacks are inherited, when developers define classes that are subclassing Android base component classes. Developers are often required to override certain, although not all, callback APIs. Forgetting to handle these callbacks may cause apps to crash immediately. Moreover, these crashes may often seem flaky, since its reproduction requires re-establishing the same system environments and/or repeating user action sequences. Existing Java crash locators fail to spot such bugs with two reasons: (1) These callback APIs are not in the extended call graphs of stack traces; (2) The method snippets in developer-defined codes do not exist, so are easily missed.

Listing 7 shows an example of this crash type. The app *Fengshui Master* is a Chinese fortune teller app. The app crashes when trying to get a reference to the writable database. However, when the app crashes, the exception *SQLiteDatabaseException* is triggered with a message claiming "not able to downgrade database from version 19 to 17".

Listing 7: Bug Explanation to Android app Fengshui Master.

```
public class com.divination1518.f.s{
    a(..){sqliteOpenHelper.getWritableDatabase();/*Crash location*/}
public class com.divination1518.g.p extends SQLiteOpenHelper{ ...
+ onDowngrade(..){...}/*Bug Location*/}
```

According to the Android documentation, the app developer needs to implement the callback method *onDowngrade* in the self-defined subclass of *SQLiteOpenHelper*. This callback method will be invoked when the database in storage has a higher version than that of the system distribution. Failing to override this callback API immediately crashes the app. Note that the motivating example (cf. Section 3) also falls into this sub-category. Given the stealthiness of this fault type, it is particularly difficult, even for a human developer, to spot the bug reason without being very familiar with the Android official documentation. Out of a total of 49 cases in Category B, 5 falls into this sub-category.

Note that we use *api_n* to denote the wrongly handled API (call-in API or callback API) for cases of Category B. This denotation is later needed for Section 5.2.2.

4.5 Category C: out of Stack Trace, out of Code

As introduced in Section 2.3, except code, an Android apk also contains resources, assets, certificate, and manifest. They are critical to the functioning of the app. As a result, mistakes in those files may also cause crashes. Table 2 gives a summary of the buggy locations outside of code. As illustrated, eleven cases of crashes originate from the *Manifest.xml* file. Most cases in this type are because the permissions are not properly declared in the manifest. Resources include specifically files with ".xml" extension (excluding the *Manifest.xml* file). An Android app uses these resource files to store the layout and pieces of information like string values. If the required resource is missing or wrong, then the app will crash. Assets are the large files, like fonts, images, bitmaps. Assets should

be put in the correct directory. If the Android framework is not able to find them and it will crash the app.

Table 2: Crash Causes of Category C

Sub-Category	Manifest	Hardware	Asset	Resource	Firmware
Cases	11	5	4	2	18

Aside from the files inside the apk, some constraints put forward by the device’s hardware and firmware, i.e., the framework may also cause the app to crash. For example, the Android billing service can only be tested on real devices, if, however, tested on emulators, the app crashes [26]. Also, since Android is quickly updated with new features and designs, old apps may crash on newly distributed devices, due to reasons like deprecated APIs and new security protocols established by the framework. Developers should generally redesign the relevant functionalities, therefore no single buggy location can be decided.

5 RANKING SUSPICIOUS LOCATIONS

To help developers identify the true fault locations when debugging app crashes, including faults that reside outside the stack traces, we propose ANCHOR. ANCHOR is a fault location recommendation system based on a two-phase approach. In the first phase, ANCHOR categorizes a given crash in one of the three categories (A, B, or C) with a classification system. Then, in the second phase, according to the decided category, ANCHOR each adopts a unique algorithm to suggest a rank of locations that are suspected to contain the true faults. The rest of this section describes Phase 1 and Phase 2 in more detail.

5.1 Phase 1: Categorization

The first phase aims at assigning a new crash to one of the three categories (A, B, or C). We use the Naïve Bayes algorithm [27] for the categorization. Naïve Bayes is one of the most effective and competitive algorithms in text-based classification. It is widely used for spam detection [28, 29], weather forecasting [30], etc. It is especially suitable in the scenario when the training set does not contain a large number of records [31], e.g., our empirical dataset contains merely 500 manually constructed records.

To construct a vector for each crash record, we feature words extracted from the crash message. The value of each feature dimension is binary, indicating whether a word exists or not in the message. More specifically, we extract three parts from the crash message: (1) The exception type, which is a Java class (e.g., *IllegalArgumentException*); (2) The exception message, which briefly describes the reason of the crash, e.g., line 2 in Figure 1; (3) The top framework stack frames, each being a Java method, e.g., lines 3-5 in Figure 1. For (1) and (3), we use “.” as the word separator, for (2), we use space as the separator. To avoid overfitting and to save computing resources, we do not need the entirety of the vocabulary to build the vector. In Section 6.4, we further discuss how many words are necessary.

With this categorization system, each new crash will firstly be categorized as a type of "A", "B" or "C" before being processed in Phase 2.

5.2 Phase 2: Localization

The goal of this phase is to provide a rank of potential bug locations (in descending order of suspiciousness), in the form of developer methods when the bug is in the code (i.e., Categories A and B) and of sub-categories when the bug is not in the code (i.e., Category C). Before presenting in the following sub-sections 3 standalone algorithms, one for each category, we explain how we compute a similarity score between two crashes. This similarity score is used in both Categories B and C localization algorithms.

Similarity between two Crashes: We quantify the similarity between two crashes C_1 and C_2 by computing the similarity between their crash messages as presented in Equation 1:

$$Sim_{C_1, C_2} = Edit_Sim(seq_{C_1}, seq_{C_2}) \quad (1)$$

seq is the sequence of framework stack frames in a crash message, e.g., lines 3-5 in Figure 1. Sim_{C_1, C_2} is then computed as the *Edit Similarity* [32] between seq_{C_1} and seq_{C_2} . The intuition here is that when two crashes share similar bug reasons, their seq tends to be similar, if not identical.

5.2.1 Category A: In Stack Trace. Since the crash is assigned to Category A, it indicates that the buggy method is one of the developer methods in the stack trace. We inherit the intuition from [5], that if the developer method is closer to the *Crash API* in the stack trace, there is a higher chance that it contains the true fault. Therefore, we can obtain the rank without changing the order of the developer methods in the stack trace. For example, in Figure 1, methods on line 6 and line 7 are respectively ranked first and second.

5.2.2 Category B: Out of Stack Trace, in the Code. When the crash is classified into Category B, it indicates that the buggy developer method is not in the stack trace, but still in the code. As discussed in Section 4.4, the buggy method should either be a developer method that misused a call-in API, or a callback API that has not been overridden. In the remainder of this section, we will note api_h this API (call-in API or callback API) that has been wrongly handled (cf. Section 4.4). To infer a ranked list of potentially buggy methods, we propose Algorithm 1. The overall idea is, starting from each developer method in the stack trace, in addition to examining the developer methods (1) in the extend call graph, we also examine those that either (2) control the Android components' lifecycles, or (3) are involved in the data flow of the crash. The computation of the suspiciousness score follows the same intuition as explained in Section 5.2.1.

First of all, Algorithm 1 requires three input data: (1) $crash$, the crash under study; (2) ST , which is the list of developer methods contained in the stack trace, e.g. lines 6-7 in Figure 1; (3) api_h , the wrongly handled API, which is approximated as the associated wrongly handled API of the most similar crash present in Category B of our empirical dataset. More formally, let be $Crash_B$ the set of all the crashes in Category B. We identify the most similar crash $crash_{sim}$ by following Equation 2. Since their crash reasons are the most similar, it is with the highest possibility that both have wrongly handled the same API.

$$Sim_{crash, crash_{sim}} = \max(Sim_{crash, crash_b}), crash_b \in Crash_B \quad (2)$$

Data: $crash$: the crash to resolve

Data: ST : List of developer methods in stack trace of $crash$

Data: api_h : *Wrongly handled API*

Result: R : Rank of suspicious developer methods

```

1:  $S \leftarrow$  Developer methods that invoke  $api_h$ ;
2: for  $sf \in ST$  do
3:   if  $api_h$  type "call-in" then
4:     for  $s \in S$  do
5:       for  $am \in AM$  do
6:         if  $s$  links  $am$  then
7:            $s.score++ = \frac{1}{d}$ 
8:         end if
9:       end for
10:    end for
11:     $R \leftarrow S.sort()$ 
12:  else if  $api_h$  type "callback" then
13:    for  $nc \in NC$  do
14:      if  $nc$  inherits  $api_h$  then
15:         $R.put(nc)$ 
16:      end if
17:    end for
18:  end if
19: end for

```

Algorithm 1: Localization Algorithm for Category B

The algorithm starts with retrieving a set of developer methods S from the entire apk that has invoked the api_h (line 1). The outmost for-loop (lines 2-19) loops over each stack frame sf in the stack trace ST . Then based on the type of the api_h , there are two sub-routines: (a) when api_h type is "call-in" (lines 4-11); (b) when api_h type is "callback" (lines 13-17). Next we discuss both sub-routines in detail.

Sub-routine for type "call-in" is a for-loop (lines 4-11) that loops over each method s in S . We then loop over (lines 5-9) all *Active Methods (AM)* declared in the same class as sf , where *Active Methods* are methods having actual code snippets in the Java class files, not including the inherited ones. The function *links* (line 6) checks 3 sub-conditions: (1) if s is invoked by am , or (2) if s and am are declared in the same Java class or (3) if an instance of the declaring class of s has been passed to am as a parameter. Sub-condition (1) checks if s is in the extended call graph of am , same as locators like [6]. Sub-condition (2) implies that s is a callback method that involves controlling the component lifecycle as am does. Sub-condition (3) implies potential data flow between s and am . When the condition holds true in line 6, a score is added for s (line 7). Here d is the distance between sf and *Crashed API* in the stack trace. It reflects on the same intuition in Section 5.2.1.

Sub-routine for type "callback" is implemented with a for-loop (lines 13-17) that loops over all the inherited Non-overridden Callback (NC) of the class where sf is declared. If nc inherits from api_h (line 14), it implies that overriding it may fix the problem, therefore nc will be added to the rank R (line 15). With the same intuition in Section 5.2.1, this sub-routine is designed so that when sf is closer to *Crashed API* in the stack trace, nc is in the higher location in the rank.

Algorithm 1 addresses the concerns in the empirical study (cf. Section 4.4). It can further locate faulty methods that are not in the extended call graphs, or even methods without actual code snippets.

5.2.3 *Category C: Out of Stack Trace, out of Code.* Figure 4 describes the localization process for crashes that have been classified into Category C. To infer a ranked list of potentially buggy locations, this process computes a suspiciousness score for each location. Since the true fault locations in Category C are not in the code, the locations in this ranked list are sub-categories (e.g. manifest, asset, etc.).

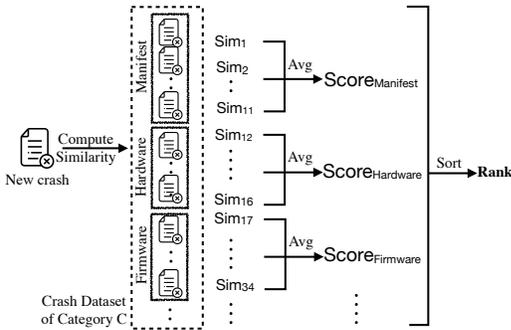


Figure 4: Localization Process for Category C.

With the new crash, we start by computing the similarity score $Sim_{crash, crash_c, crash_c} \in Crash_C$. Here $Crash_C$ is the set of all the crashes of Category C in the empirical dataset. In Figure 4, the similarity scores are denoted as Sim_{caseID} for short. We then take an average of Sim_{caseID} over the same sub-categories. Sub-categories with higher similarity scores take higher positions in the *Rank*.

6 EXPERIMENTAL SETUP

This section clarifies the research questions, the metrics used to assess ANCHOR, and the parameter values involved.

6.1 Research questions

We empirically validate the performance of ANCHOR by investigating the following research questions:

- **RQ1:** To what extent is the categorization strategy effective?
- **RQ2:** To what extent are the localization algorithms reliable?
- **RQ3:** What is the overall performance of ANCHOR?
- **RQ4:** How does ANCHOR perform on crashes in the wild?

6.2 Metrics

Crash localization is a recommendation problem. To measure the performance of ANCHOR, we rely on rank-aware metrics, which are widely used in information retrieval communities and have been previously used to evaluate crash localization techniques [6].

Recall@k: The percentage of crash cases whose buggy functions appear in top k locations. A higher score indicates better performance of ANCHOR.

MRR (Mean Reciprocal Rank): The mean of the multiplicative inverse of the rank of the first correct location. As defined in Equation 3, $Rank_i$ is the rank for the i^{th} crash case, in a set of crash cases E . A high value of MRR means developers on average need to examine fewer locations in the rank, and therefore, a better performance [33].

$$MRR = \frac{1}{|E|} \sum_{i=1}^{|E|} \frac{1}{Rank_i} \quad (3)$$

6.3 Cross-validation

We perform 5-fold cross-validation over the empirical dataset of 500 sample crashes. The dataset is randomly divided into 5 subsets of 100 sample crashes: 5 experiments are then carried where every time a specific subset of 100 is used as “test” data while the remaining subsets containing the rest 400 cases are merged to form “training” dataset. The computed performance metrics are then summed over the 5 folds.

6.4 Feature Selection

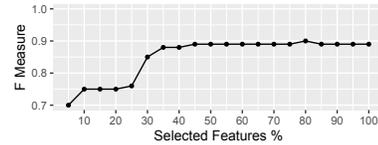


Figure 5: F Measure v.s. Selected Features.

In the empirical dataset, the vocabulary contains 1108 unique words. To avoid over-fitting, we select only a portion of them for Phase 1. We use the χ^2 test for each word [34]. A higher value of χ^2 indicates a stronger correlation between the word and the category. Figure 5 shows the relation between the F Measure of Phase 1 and the percentage of words chosen (ranked in descending order by χ^2 values). We can see that with the top 50% of the features, the overall performance already stabilizes. We then always use top 50% of the words in the vocabulary.

7 EXPERIMENTAL RESULTS

7.1 RQ1: Effectiveness of Categorization

We use our ground truth of 500 crashes to assess the performance of ANCHOR in the first phase of the approach, namely the categorization. We provide in Table 3 the confusion matrix as well as the precision and recall of our experimental results. ANCHOR yields a very high precision for predicting crashes in Category A, reaching 0.96. The precision for crashes in Categories B and C are comparably lower, at 0.65 and 0.60, respectively. In terms of recall, the approach is effective for Category A (0.91), Category B (0.82), and Category C (0.75). Overall, ANCHOR is successful in categorizing 444 out of 500 crash samples (88.8%).

Answer to RQ1: ANCHOR is overall effective in categorizing new crash samples. However, there is still room of improving the precision when predicting samples in Categories B and C.

Table 3: Effectiveness of Categorization (Phase 1)

	Actual			Total	Precision	Recall	
	A	B	C				
Predicted as Category A	374	6	8	388	Category A	0.96	0.91
Predicted as Category B	20	40	2	62	Category B	0.65	0.82
Predicted as Category C	17	3	30	50	Category C	0.60	0.75
Total	411	49	40	500			

7.2 RQ2: Effectiveness of Localization

To evaluate the localization phase of ANCHOR, we consider sample crashes for each category and assess the rank localization yielded by the specific algorithm developed for that category. Table 4 summarizes the Recall@k (with $k \in \{1, 5, 10\}$) and MRR.

To make sure the evaluation of Phase 2 is not affected by the outcome of Phase 1, we propose to assess the performance of localization with the assumption of perfect categorization.

Table 4: Localization Performance

Category	Recall@1	Recall@5	Recall@10	MRR
A	0.97(400/411)	0.99(406/411)	0.99(407/411)	0.98
B	0.39(19/49)	0.61(30/49)	0.63(31/49)	0.48
C	0.78(31/40)	1.00(40/40)	1.00(40/40)	0.85
Total	0.90(450/500)	0.95(476/500)	0.96(478/500)	0.92

For cases in Category A, the true fault location can almost always be found at the top of the rank. The high value of MRR at 0.98 confirms the intuition in Section 5.2.1 that it takes much less effort in finding fault location for Category A. For cases in Category B, the recall@1 starts at 0.39 and increased substantially for recall@5 at 0.61. One more case is successfully located with recall@10 at 0.63. The overall MRR is 0.48. Given the fact that the search space is vast (there can be tens of thousands of developer methods in the apk), Algorithm 1 demonstrates decent performance. For most cases in Category C, the true sub-category of the fault location can be found topmost, with the MRR at 0.85.

Answer to RQ2: The localization algorithms (Phase 2) of ANCHOR are reasonably effective for suggesting the correct fault location. ANCHOR shows descent performance even when challenged by the vast search space for crashes in Category B.

7.3 RQ3: Overall Performance of ANCHOR

Table 5 summarizes the overall performance of ANCHOR combining Phase 1 and 2. The MRR of all 3 categories slightly dropped, since some cases are miscategorized in Phase 1. Clearly, the overall performance is affected by Phase 1. However, since the two phases in ANCHOR are loosely coupled, we envisage improvements of overall performance in the future when better classifiers are proposed.

Answer to RQ3: ANCHOR is an effective approach for locating crashing faults when they are in/outside stack traces, even outside code. Better performance is guaranteed when categorization (Phase 1) is further improved.

Table 5: Overall Performance of ANCHOR

Category	Recall@1	Recall@5	Recall@10	MRR
A	0.90(370/411)	0.91(373/411)	0.91(373/411)	0.90
B	0.37(18/49)	0.59(29/49)	0.61(30/49)	0.46
C	0.72(29/40)	0.75(30/40)	0.75(30/40)	0.73
Total	0.83(417/500)	0.86(432/500)	0.87(433/500)	0.85

7.4 RQ4: Performance in the Wild

The heuristics based on which ANCHOR is built may be biased by the empirical dataset. To mitigate this threat, we assess the effectiveness of ANCHOR with a dataset selected in the wild. We want to verify to what extent ANCHOR can be generalized. We leverage the independent dataset prepared by Fan et al. [3] who thoroughly (by crawling the entire GitHub) and systematically (by applying strict criteria) collected 194 crashed apks from open-source Android repositories. Before evaluation, we apply the constraint rules of Section 4.1, and focus on the 69 relevant crash cases that could be identified. Note that this dataset contains true fault locations already verified by the app developers. Since the cases in the dataset are from a wide time span (2011-2017), the partition is randomly decided on normal distribution over the year of app release.

Table 6: Categorization on an independent dataset.

	Actual			Total	Precision	Recall	
	A	B	C				
Predicted as Category A	54	1	0	55	Category A	0.98	0.93
Predicted as Category B	3	6	0	9	Category B	0.67	0.86
Predicted as Category C	1	0	4	5	Category C	0.80	1.00
Total	58	7	4	69			

Table 6 shows the confusion matrix, as well as the precision and recall of Phase 1 (categorization) on this independent dataset. The precision for all categories is high, reaching 0.98 (54/55), 0.67 (6/9), and 0.80 (4/5) respectively. The recalls are also high, at 0.93 (54/58) for A, 0.86 (6/7) for B, and a perfect 1.00 (4/4) for C.

Table 7 provides measures for the overall performance. To compute the similarity scores which are required to locate the bug related to crashes from Categories B and C, we use the crash records from the empirical dataset. The recalls and MRR in Category A remain high. As for Category B, ANCHOR is able to yield recall@k values and MRR of 0.43 when suggesting fault locations. As for Category C, the total MRR is at 0.43, suggesting more stack traces in Category C might be the key for better performance.

Table 7: Recall@k and MRR on an independent dataset.

Category	Recall@1	Recall@5	Recall@10	MRR
A	0.72(42/58)	0.93(54/58)	0.93(54/58)	0.81
B	0.43(3/7)	0.43(3/7)	0.43(3/7)	0.43
C	0.25(1/4)	1.00(4/4)	1.00(4/4)	0.40
Total	0.67(46/69)	0.88(61/69)	0.88(61/69)	0.74

Answer to RQ4: The evaluation on an independent dataset shows that ANCHOR can be generalized. ANCHOR is a milestone in this respect that it considers various crashing location cases.

However, a community effort is still required to construct a representative dataset of crashes to push forward the state of the art in crashing fault localization.

8 DISCUSSION

8.1 Comparing ANCHOR with other Locators

Along with their empirical analysis of Android app crashes, Fan et al. [3] mentioned, in a single paragraph, a prototype crashing fault locator: ExLocator. Unfortunately, since the tool has not been publicly released, we could not directly compare it against ANCHOR. We note, based on its description, however, that ExLocator has a limited usage scenario since it focuses on only 5 exception types. CrashLocator [6] can also locate faults outside the stack trace. However, CrashLocator needs to abstract patterns from a great number of repeated crashes of the same project. Unfortunately, for both datasets presented in this paper, this requirement is not satisfied. Moreover, CrashLocator requires source code and change tracking of the projects, unavailable for our empirical dataset. Therefore, we are not able to apply CrashLocator.

Although direct comparison in terms of effectiveness is not possible in this scenario, we can compare the applicability. ANCHOR is considered to have a wider application range compared to ExLocator, i.e., it can be applied to all exception types, and considered to be more lightweight and source code independent compared to CrashLocator, i.e., it requires only the crash message and the apk.

8.2 Developer Effort for Locating Bugs

In the motivating example, we demonstrated why locating buggy methods outside the stack trace can be challenging. We also want to measure the effort that developers put in locating such bugs. In Fan et al.'s dataset, each crash is documented with its duration, i.e., the time between the issue creation and its official closure by the developers. For bugs in the stack trace, it takes developers 26 days on average to close the issues. For bugs outside the stack trace, it drastically increases to 41 days. The ratio is $41/26=158\%$. Although it may not always be precise to measure effort in terms of issue duration, this would confirm our observation to some extent.

9 THREATS TO VALIDITY

9.1 Internal Threats

In the empirical study presented in Section 4, we have manually built the ground truth of buggy locations that we made available to the community. Although we have tried our best to perform this manual inspection with the help of (1) the Android official documentation, (2) programmer information exchanging forums like StackOverflow or GitHub, (3) tools such as Soot or CodeInspect, there is no guarantee that all buggy locations we retrieved are the true causes for the crashes. This might affect the conclusions we draw from this dataset and the answers to RQ1-RQ3.

9.2 External Threats

We extracted our dataset from the open benchmark ReCBench built by Kong et al [4]. Although the large dataset they propose contains diverse apks collected from various popular app markets such

as Google Play (ensuring a good diversity of apps), the collected crash cases are retrieved by testing apks with only two testing tools. Therefore, the yielded crashes could not be representative of the whole spectrum of crashes present in the Android ecosystem. Similarly, the dataset proposed by Fan et al. [3] is extracted from open source Android app GitHub repositories only. Moreover, they have applied certain rules for collecting the crashed cases, e.g., they extract only crash bugs that have been closed by repository maintainers. The potential limitations with both datasets may affect the effectiveness we have shown in RQ1-RQ4.

10 RELATED WORK

A recent survey by Wong et al. [35] marks the activity of identifying the locations of faults in a program to be most tedious, time-consuming, and expensive, yet equally critical. Therefore, lots of techniques have been proposed attempting to ease the work of finding the fault locations. Although we did not find a dedicated tool for identifying locations in Android apps, there are some approaches proposed for general software programs. For example, Wu et al. proposed CrashLocator [6] to score and rank suspicious locations that have caused program crashes. CrashLocator suggests that the buggy methods can be located in the static call graphs extended from the stack traces. However, it is not suitable to work on programs with multiple entry points and separate call graphs such as Android apps. Moreover, its scoring factors, which require source code and change histories, also limit its application scope to Android apps, for which most of them are released in a closed way (i.e., no change histories). Gu et al. [36] proposed another approach called CraTer that adopts information retrieval techniques to predict whether the real fault resides inside the stack traces. However, CraTer is not able to suggest the actual buggy location. BugLocator [37] applies a revisited Vector Space Model (rSVM) to retrieve relevant files for fixing a bug on a large number of bug reports. However, its granularity falls in file level, which still requires human verification for more fine-grained location identification. Wong et al. [38] build their work on top of BugLocator [37] and leveraged stack trace to improve the approach and indeed achieved better performance. Fan et al. [3] briefly describes a fault localization prototype ExLocator for Android apps. ExLocator only supports 5 exception types and has a limited usage scenario. Furthermore, in the community of Automatic Program Repair (APR), statement-level fault localization is often among the first few steps. Researchers have improved it in various aspects [39–45].

Many research works have been proposed to address Android app crashes in recent years. For example, Fan et al. [3] performed a large scale analysis on framework-specific Android app crashes. They have invented the grouping techniques to group the Android app crash cases into buckets to study similar root causes based on each bucket. Researchers have also spent efforts attempting to automatically reproduce the reported crashes [46, 47]. Indeed, to achieve this purpose, Zhao et al. have proposed ReCDroid [48], which applies a combination of natural language processing (NLP) and dynamic GUI exploration to reproduce given crashes. Gómez et al. [49] proposed another approach for reproducing crashes by providing sensitive contexts. Moran et al. [50] further presented

a prototype tool called CrashScope, aiming at generating an augmented crash report to automatically reproduce crashes on target devices. Researchers have gone one step deeper to propose automated tools to automatically fix such identified crashes. Indeed, Tan et al. [5] have proposed an automatic repairing framework named Droix for crashed Android apps. Droix adopts 8 manually constructed fixing patterns on crashed Android apps to generate app mutants and suggest one that fixes the crash. Following this work, Kong et al. [4] present to the community an automatic fix pattern generation approach named CraftDroid for fixing apps suffering from crashes.

The special Android callback-based mechanism and its effect have drawn the attention of many researchers with the ever-booming of Android devices. Yang et al. [8] targets the even-driven and multi-entry point issue of Android apps, and proposed a program representation that captures callback sequences by using context-sensitive static analysis of callback methods. Flowdroid [11] targets at exposing privacy leakages on Android phones. It establishes a precise model of the Android lifecycle, which allows the analysis to properly handle callbacks invoked by the Android framework. Relda2 [51] is a light-weight and precise static resource leak detection tool based on Function Call Graph (FCG) analysis, which handles the features of the callbacks defined in the Android framework. Together with other existing works like [10, 52], they all dealt with Android callback-based mechanism in various manners. Although these works are different from ours, their approach in handling lifecycle and callback methods could be borrowed to enhance our approach towards better dealing with Category B crashes.

11 CONCLUSIONS

In this work, we performed an empirical study. This study shows that 37% crash types are related to bugs that are outside the stack traces, which imposes challenges to the localization problem. We then proposed ANCHOR, a two-phase categorization and localization tool that is able to generate a ranked list of bug locations for developers to examine. The effectiveness of ANCHOR is assessed with both this empirical dataset and an in-the-wild scenario on a third-party dataset. Our work brings inspiring insights into the crashing faults localization problem for Android apps and calls for more attention from both the developers and the research community.

REFERENCES

- [1] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 226–237. ACM, 2016.
- [2] Google. Crashes | android developers. <https://developer.android.com/topic/performance/vitals/crash>.
- [3] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. Large-scale analysis of framework-specific exceptions in android apps. In *Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 408–419. IEEE, 2018.
- [4] Pingfan Kong, Li Li, Jun Gao, Tegawendé F Bissyandé, and Jacques Klein. Mining android crash fixes in the absence of issue- and change-tracking systems. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 78–89. ACM, 2019.
- [5] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. Repairing crashes in android apps. In *Proceedings of the 40th International Conference on Software Engineering*, pages 187–198. ACM, 2018.
- [6] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. Crashlocator: locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 204–214. ACM, 2014.
- [7] Dongsun Kim, Xinming Wang, Sunghun Kim, Andreas Zeller, Shing-Chi Cheung, and Sooyong Park. Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. *IEEE Transactions on Software Engineering*, 37(3):430–447, 2011.
- [8] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 89–99. IEEE, 2015.
- [9] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering—Volume 1*, pages 77–88. IEEE Press, 2015.
- [10] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *The 2016 International Symposium on Software Testing and Analysis (ISSTA 2016)*, 2016.
- [11] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Acm Sigplan Notices*, volume 49, pages 259–269. ACM, 2014.
- [12] Google LLC. App manifest overview. <https://developer.android.com/guide/topics/manifest/manifest-intro>. Accessed: 2020-01-26.
- [13] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471, May 2016.
- [14] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. AndroZoo++: Collecting millions of android apps and their metadata for the research community. *arXiv preprint arXiv:1709.05281*, 2017.
- [15] Fraunhofer. Codeinspect tool of fraunhofer. <https://codeinspect.sit.fraunhofer.de/>.
- [16] Jue Wang, Yanyan Jiang, Chang Xu, Qiwei Li, Tianxiao Gu, Jun Ma, Xiaoxing Ma, and Jian Lu. Aatt+: Effectively manifesting concurrency bugs in android apps. *Science of Computer Programming*, 163:1–18, 2018.
- [17] Pavol Bielik, Veselin Raychev, and Martin Vechev. Scalable race detection for android applications. *ACM SIGPLAN Notices*, 50(10):332–348, 2015.
- [18] Qiwei Li, Yanyan Jiang, Tianxiao Gu, Chang Xu, Jun Ma, Xiaoxing Ma, and Jian Lu. Effectively manifesting concurrency bugs in android apps. In *2016 33rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 209–216. IEEE, 2016.
- [19] Hongyin Tang, Guoquan Wu, Jun Wei, and Hua Zhong. Generating test cases to expose concurrency bugs in android applications. In *Proceedings of the 31st IEEE/ACM international Conference on Automated software engineering*, pages 648–653, 2016.
- [20] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for android applications. *ACM SIGPLAN Notices*, 49(6):316–325, 2014.
- [21] Shujuan Jiang, Hongchang Zhang, Qingtan Wang, and Yanmei Zhang. A debugging approach for java runtime exceptions based on program slicing and stack traces. In *2010 10th International Conference on Quality Software*, pages 393–398. IEEE, 2010.
- [22] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. Do stack traces help developers fix bugs? In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 118–121. IEEE, 2010.
- [23] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. Fault localization and repair for java runtime exceptions. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 153–164, 2009.
- [24] Trupti S Indi, Pratibha S Yalagi, and Manisha A Nirgude. Use of java exception stack trace to improve bug fixing skills of intermediate java learners. In *2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*, pages 194–198. IEEE, 2016.
- [25] Wei Song, Jing Zhang, and Jeff Huang. Servdroid: detecting service usage inefficiencies in android applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 362–373, 2019.
- [26] Google LLC. Test google play billing. <https://developer.android.com/google/play/billing/testing.html>. Accessed: 2020-01-26.
- [27] Irina Rish et al. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46, 2001.
- [28] Vangelis Metsis, Ion Androutsopoulos, and Georgios Paliouras. Spam filtering with naive bayes-which naive bayes? In *CEAS*, volume 17, pages 28–69. Mountain View, CA, 2006.
- [29] Zhen Yang, Xiangfei Nie, Weiran Xu, and Jun Guo. An approach to spam detection by naive bayes ensemble based on decision induction. In *Sixth International Conference on Intelligent Systems Design and Applications*, volume 2, pages 861–866. IEEE, 2006.
- [30] Nephi A Walton, Mollie R Poynton, Per H Gesteland, Chris Maloney, Catherine Staes, and Julio C Facelli. Predicting the start week of respiratory syncytial

- virus outbreaks using real time weather variables. *BMC medical informatics and decision making*, 10(1):68, 2010.
- [31] Yuguang Huang and Lei Li. Naive bayes classification algorithm based on small sample set. In *2011 IEEE International Conference on Cloud Computing and Intelligent Systems*, pages 34–39. IEEE, 2011.
- [32] Jianbin Qin, Wei Wang, Yifei Lu, Chuan Xiao, and Xuemin Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1033–1044, 2011.
- [33] Yue Shi, Alexandros Karatzoglou, Linas Baltrunas, Martha Larson, Nuria Oliver, and Alan Hanjalic. Clmf: learning to maximize reciprocal rank with collaborative less-is-more filtering. In *Proceedings of the sixth ACM conference on Recommender systems*, pages 139–146, 2012.
- [34] Rupert Miller and David Siegmund. Maximally selected chi square statistics. *Biometrics*, pages 1011–1016, 1982.
- [35] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.
- [36] Yongfeng Gu, Jifeng Xuan, Hongyu Zhang, Lanxin Zhang, Qingna Fan, Xiaoyuan Xie, and Tiejun Qian. Does the fault reside in a stack trace? assisting crash localization by predicting crashing fault residence. *Journal of Systems and Software*, 148:88–104, 2019.
- [37] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 14–24. IEEE, 2012.
- [38] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 181–190. IEEE, 2014.
- [39] Rui Abreu, Peter Zoetewij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.
- [40] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. ifixr: bug report driven program repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 314–325. ACM, 2019.
- [41] Rui Abreu, Peter Zoetewij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.
- [42] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282. ACM, 2005.
- [43] Rui Abreu, Peter Zoetewij, and Arjan JC Van Gemund. Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99. IEEE, 2009.
- [44] Qianqian Wang, Chris Parmin, and Alessandro Orso. Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 1–11, 2015.
- [45] Sangeeta Lal and Ashish Sureka. A static technique for fault localization using character n-gram based information retrieval model. In *Proceedings of the 5th India Software Engineering Conference*, pages 109–118, 2012.
- [46] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 2017.
- [47] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *IEEE transactions on software engineering*, 43(9):817–847, 2016.
- [48] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ Halfond. Recroid: automatically reproducing android application crashes from bug reports. In *Proceedings of the 41st International Conference on Software Engineering*, pages 128–139. IEEE Press, 2019.
- [49] María Gómez, Romain Rouvoy, Bram Adams, and Lionel Seinturier. Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring. In *International Conference on Mobile Software Engineering and Systems*, 2016.
- [50] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Automatically discovering, reporting and reproducing android application crashes. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pages 33–44. IEEE, 2016.
- [51] Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. Light-weight, inter-procedural and callback-aware resource leak detection for android apps. *IEEE Transactions on Software Engineering*, 42(11):1054–1076, 2016.
- [52] Zheming Yang and Min Yang. Leakminer: Detect information leakage on android with static taint analysis. In *2012 Third World Congress on Software Engineering*, pages 101–104. IEEE, 2012.