How Developers Handle Exceptions and Fix Exception Bugs in Mobile Apps?

Tam Nguyen and Tung Nguyen

October 18, 2024

How Developers Handle Exceptions and Fix Exception Bugs in Mobile Apps?

Abstract—**Modern programming languages provide a mechanism for programmers to handle exceptions, i.e., unexpected errors occurring while a software system is running. However, learning to handle exceptions correctly is often challenging for mobile app developers due to the fast-changing nature of API frameworks for mobile systems and the insufficiency of API documentation and source code examples. Failing to catch and handle exceptions could leads to serious bugs and issues such as crashing, resource leaking, or causing negative end-user experiences. The goal of our paper is to present a thorough understanding about the nature of exception-related bugs and how developers handle exceptions and fix those bugs in mobile app environment. To that end, we first present a detailed empirical study on 246 exception-related bugs and fixes from 10 Android apps. Eight findings from the study suggest that handling exception has proven difficult and error-prone for many app developers. Our study suggests the need of tool support to help app developers become better in this task. Thus, we also performed another study on how professional app developers handle exceptions in 4000 high-quality Android apps. This study provides useful guidance and insights on building exception handling recommendation tools.**

I. Introduction

Exceptions are unexpected errors occurring while a software system is running. For example, when a software system wants to open a file with a given file name but, unexpectedly to the programmer, the file system contains no file having that name, a *"file not found"* exception occurs. Failing to handle exceptions properly could lead to more serious errors and issues such as system crashes or resource leaks. For example, a prior study reports that correctly releasing resources in the presence of exceptions could improve of 17% in performance of the application [1]. Thus, effective exception handling is important in software development.

Modern programming languages like Java or C++ often provide built-in supports for exception handling. For example, in Java, we can wrap a `try` block around a code fragment where one ore more exceptions potentially occur. Then, we add one or more `catch` blocks to handle such exceptions. An API library often defines many API-specific exception types and exception handling rules. For example, in Java SDK, class `FileNotFoundException` is defined for the *file not found* exception. When such an exception happens, the software system could notify users about the error and/or write the relevant information (e.g., filename) to the system's log for future debugging or investigations.

However, learning to handle exception properly is often challenging especially in mobile app development due to several reasons. First, a mobile API library often consists a large number of components. For example, the Android application framework contains over 3,400 classes, 35,000 methods, and more than 260 exception types [2]. Moreover, the documentation for handling exceptions is generally insufficient. Kechagia *et al.* found that 69% of the methods from Android packages in their stack traces had undocumented exceptions in the Android API and 19% of the crashes could have been caused by insufficient documentation. Coelho *et al.* [3] found that documentation for explicitly thrown `RuntimeExceptions` is almost never provided. Second, due to the fast development of mobile devices and the strong competition between software and hardware vendors, those frameworks are often upgraded quickly and include very large changes and therefore they could introduce new types of exceptions. For example, 17 major versions of Android framework containing nearly 100,000 method-level changes have been released within five years [2]. Thus, it is difficult to learn and memorize what method could causes what exception and what to do when a particular exception occurs. That could leads to high number of programming errors (bugs) related to exceptions in app development.

In this paper, we first present a detailed empirical study on 246 exception-related bugs and fixes from 10 Android apps. Our aim is to understand the nature of exception-related bugs in mobile app development and how app developers fix them. In particular, we focus on three main research areas:

1. Causes and Effects of Exceptions. Missing exception handling code or failing to handle exceptions properly could introduce serious errors in an app. Thus, we focus on how exception bugs affect the running apps. Secondly, we study exception bugs in several aspects including what are object types and methods often cause exception bugs, what are exception types that often happen in those bugs, and are there any correlations between object types or methods that cause exceptions and exception types.

2. Handling Exceptions. In this research area, we focus what app developers do in exception bug fixes. Swallowing exception by adding a simple `try-catch` block could avoid the program crashing when exception occurs but might introduce new bugs to an app. Thus, we study whether this type of handling occurs in the bug fixes. Next, we focus on what type of actions that app developers do in their handling code. Finally, we also identify if there are still bugs after developers handle exceptions.

3. Logging Exceptions. Logging is a method to pinpoint the existence and location of exceptions and provide runtime information that helps programmers in fixing exception bugs. In this research area, we focus on the logging code in exception bug fixes.

We manually collected and analyzed 246 bugs and fixes related to exceptions across 10 Android apps. We discovered several interesting findings from our empirical study on the dataset.

First, we found out that almost all exception bugs causes serious problems for the apps such as crashing or running in a unstable state. In 80% of the cases, exception bugs cause crashes and in 13% of the exception bugs cause the apps running in a unstable state or make some features not function properly. Further investigate, we found out that 51% (127/246) of exception bugs are caused by Android API methods. Regarding exception types, we found out that in 246 exception bugs, there are 58% (143/246) of runtime exceptions. We also found out that there are some exceptions have very strong associations with some particular method calls.

In the second research area, our results show that programmers do not perform any actions to handle occurring exceptions in about 16% (40/246) bug fixes and they add repairing code in only 42% of the exception bug fixes. More interestingly, we discover some patterns in the repairing code of developers. The patterns include closing or releasing objects that hold resources such as Cursor, Activity; invoke method that causes exception again or with different parameters; or create a new object to replace object that causes the exception. In addition, we found that mobile developers even failed to handle exceptions properly in several exception bug fixes.

Finally, we focus on the logging actions of app developers when fixing exception bugs. In totals, programmers add log statements in about 76% (187/246) of bug fixes. We further investigate on what programmers log when handling exceptions. We found that about in 71% (134/187) of bug fixes that have log statements, programmers only log messages indicates the exceptions or where the exceptions happens, i.e. stack trace. In the remaining 29% (53/187), programmers includes more context information in logs such as value or information of objects when exception. In general, including context information is a better logging scheme as it provides more runtime information to help developers in debugging and fixing exception bugs.

The study suggests that automated exception handling recommendations are needed to assist app developers prevent and fix exception bugs, and help them to handle exception correctly. Thus, we also performed study on how professional developers handle exceptions on 4000 high-quality mobile apps. We found several interesting findings. First, we found that developers spend most of their time handle runtime exceptions. Thus, a recommendation model could focus on those types of exceptions. Second, we found that there are correlations between methods and exception types in terms of co-occurrence, which suggests that co-occurrence can be used to predict potential exceptions that might occur when using a specific method. Finally, we found that most of the time, app developers only use one main method per an object to handle exceptions. The result from the study suggests several guidelines for building exception handling support models.

The key contributions of our paper include:

- A study of 246 exception-related bugs and fixes from 10 mobile apps, and eight findings providing insights about exception bugs and fixes in mobile app development.
- A study on how professional app developers handle exceptions on 4000 high-quality mobile apps. Which provides several suggestions on how to detect and handle exception correctly.

II. Study on Exception Bug Fixes in Mobile Apps

In this section, we describe how we collected and built the dataset used in our empirical study and our research questions. The dataset includes real bug fixes that related to exceptions in mobile app development. In particular, we focus on bugs that are caused by not catching exceptions or adding proper exception handlers. For convenience, we defined those bugs as **exception bugs**. The fix for those type of bugs is called **exception bug fixes**. Understanding the nature of exception bugs and fixes helps us to answer our research questions.

A. Dataset

To perform our empirical study on bug fixes, we collected a dataset consists of several open-source Android projects. Table I lists 10 subject projects used in our study. Each project is an application published in Google Play Store [1]. AntennaPod is an open-source podcast manager for Android. ConnectBot is a SSH, telnet and terminal emulator. Conversations is a XMPP-based instant messaging client. FBReader is an e-book reader. K-9 is email client focused on managing large volumes of email. MozStumbler is a wireless network scanner developed by Mozilla. PressureNet is a crowd-sourced barometer network. Signal and Surespot are secure instant messaging apps. WordPress is the official Android client of WordPress. All the projects are written in Java and have source code repositories available on GitHub. For each project, we checked out its source code repository to retrieve all the code and commits. The number of commits of each project (at the time we checked out the repositories) are listed in column Commits. To ensure the reliability of our study, we selected medium to large projects compared to other Android projects. Each project has at least over 1,000 commits and the total number of commits are over 66,000.

To study exception bug fixes, the first step of our analysis on the dataset is identifying bug fixes. We define a bug fix is a commit in which the fixer fixed a bug found in the project. Identifying all bug fixes in a software project is often a hard task as issue tracking systems are incomplete to capture all bugs, and the fixers might not describe a fix explicitly. Generally, bug fixes are identified based on two types of bugs: 1) bugs reported through issue trackers, which are called *reported bugs*, and (2) those not reported to issue trackers, which are called *on-demand bugs* [4].

1. Reported bugs. All the projects in our empirical study have an issue tracking system to track various issues including bugs,

[1]play.google.com

TABLE I: The Empirical Dataset

Mobile App	**Website**	**Commits**	**Bug Fixes**	**Candidate Bug Fixes**	**Exception Bug Fixes**
AntennaPod	antennapod.org	3,404	767	89	28
ConnectBot	connectbot.org	1,450	327	39	8
Conversations	conversations.im	3,318	922	123	23
FBReaderJ	fbreader.org	27,944	1,403	118	30
K-9	github.com/k9mail	7,254	1,797	223	31
MozStumbler	location.services.mozilla.com	2,667	692	55	7
PressureNet	cumulonimbus.ca	1,017	203	31	11
Signal	whispersystems.org	2,754	1,030	130	28
Surespot	surespot.me	1,590	274	54	8
WordPress	apps.wordpress.org	15,546	3,691	425	72
Total		**66,944**	**11,106**	**1,287**	**246**

improvements, new features, tasks, etc. Using the system, users can report issues or requests they encountered when they are using the software. Each reported issue has an associated issue number and can be labeled based on the type of it (*e.g.* bugs, improvements, new features, tasks, and sub-tasks). As an example, in Wordpress, an user reported a bug with the title "Crash report 3.7: IllegalStateException in WPDrawerActivity #2372". The issue number for this bug is #2372 and it is labeled as [Type] Bug. Additional information is also included in the thread of the issue such as its status, its description, the discussions among programmers, and the relations to other issues. The programmers or fixers of the project then look at the reported issue and try to resolve it. If a change of an issue is committed, programmers often write its issue number to the message of the commit. For example, in the fix commit for the above bug, the commit's message says "fix #2372 by catching IllegalStateException" to indicate that, in the commit, the fixer fixed the bug in the issue #2372 by catching IllegalStateException. As the fixer already fixed the bug, he marked the status of the issue as closed or fixed.

To identify bug fixes from *reported bugs*, [4], [5] used a pattern to extract all commits that mention a issue number. Then they used the issue number to determine whether it indicates a bug or not (using the label of the issue). If the issue is a bug, the associated commits are considered bug fixes. For instance, the commit in the previous example points to the issue #2372, and as the issue is a bug, the commit is considered as a bug fix.

2. On-demand bugs. Issue trackers often do not contains all information about the bugs and fixes. In many cases, developers might bypass the issue trackers, especially when they discover bugs from other sources by themselves or other sources (e.g. user reviews, discussion with other developers). In such cases, they often fix the bugs and commit the changes without creating issues in the issue tracking system. When they commit a change, programmers may write a message to describe the fix. For example, in the project Wordpress, the message of a commit indicates "Fixes crash where postID could be larger than max int value". In the commit, the developer added a try-catch block to handle a NumberFormatException which caused the app scraching if postID is larger than maximum integer value. The bug that the developer referred in the commit message is an *on-demand bug*.

To identify bug fixes from *on-demand bugs*, a number of previous studies (e.g. [4], [6], [7] used a simple keyword-based technique. They identified a commit is a bug fix if its message contains words such as "fix", "fixes","bug", or "patch". The method is based on assumption that when fixing a bug, developers often write commit message to describe the fix. In the previous, as the commit message contains the word "fixes", it is considered as a bug fix.

In our study, we identified bug fixes using both methods described above and combine the result to form a set of bug fixes for each project. The column Bug Fixes lists the number of bug fixes of each project. In total, there are over 11,000 bug fix commits over 10 projects.

The next step in our analysis is identifying exception bug fixes from all bug fixes. Not all bug fixes are related to exception and manually identifying exception bug fixes from all bug fixes is very time consuming. Thus, we used a semi-automated method for identifying exception bug fixes. First, we developed a simple filter technique to only consider bug fixes that can potentially be exception bug fixes. As most of the exception bug fixes involving adding try-catch blocks, we consider a commit to be *candidate* exception bug fixes if the changes in the commit contain at least one adding of catch statement. For example, consider a change of the an example bug fix shown in Figure 1.

COMMIT MESSAGE: "fix #2695: re-introduce a workaround we were using in previous versions"

```
- postContent = new SpannableStringBuilder(
-                               mEditorFragment.getSpannedContent());
+ try {
+     postContent = new SpannableStringBuilder(
+                               mEditorFragment.getSpannedContent());
+ } catch (RuntimeException e) {
+     // A core android bug might cause an out of
+     // bounds exception, if so we'll just use the current editable
+     // See https://code.google.com/p/android/issues/detail?id=5164
+     postContent = new SpannableStringBuilder(
+         StringUtils.notNullStr((String) mEditorFragment.getContent()));
+ }
```

Fig. 1: Example 1

We can see that in the fix version, the developer added an catch statement. Thus, the commit is considered as a *candidate* exception bug fixes. After using this filtering method, the number of *candidate* exception bug fixes are 1,287 as showed in column Candidate Bug Fixes of Table I.

Next, we manually inspected the remaining 1,287 bug fixes to find exception bug fixes. We formed an assessment group consists three Ph.D. students in the field of software engineering to do this task. A bug fix is considered an exception bug fix if an exception is thrown on the bug and the fixers fix the bug by adding proper exception handling code. Our criteria to determine whether a bug fix is an exception bug fix is based on commit message, changes in commit, and comments adding in source files. In some bug fixes, we may not have all these three information to consider whether a bug fix is an exception bug fix, e.g. comments are missing. In these situations, we used all available information and tried our best to understand what developers do in these bug fixes before making classifications. At the end, a bug fix is considered as an exception bug fix if at least two of three students vote for it. For example, in the preceding example bug fix, the developer noted that he fixed the issue #2695. By looking at the issue in the issue tracking system of the project, we found that it is a reported bug as its description showed a stack trace in which an IndexOutOfBoundsException had been thrown when the code was executed. In the change of the commit, we can see that he added a try-catch block to catch the method new SpannableStringBuilder. He also commented that he found the cause of the bug is related to an core issue of Android and he tried to work around with it by using the current editable. Based on these information, we identified that the example commit is an exception bug fix.

COMMIT MESSAGE: “Catching IllegalArgumentException when unregistering receiver.”

```
@Override
protected void onPause() {
  super.onPause();

+ try {
+   unregisterReceiver(mGalleryReceiver);
+ } catch (IllegalArgumentException e) {
+   AppLog.d(T.EDITOR, "Illegal state! Can't unregister receiver that was no
+   registered");
+ }
  stopMediaUploadService();
  mAutoSaveTimer.cancel();
}
. . .
@Override
protected void onDestroy() {
  super.onDestroy();
- unregisterReceiver(mGalleryReceiver);
  AnalyticsTracker.track(AnalyticsTracker.Stat.EDITOR_CLOSED_POST);
}
```

Fig. 2: Example 2

Figure 2 shows a more complicated example that we agreed to be an exception bug fix. Because this is a *on-demand bug*, there was not associated issue for us to know more about the bug. Thus, our judgment mostly based on the commit message and the code changes. In the commit message, the fixer noted that he caught an IllegalArgumentException when unregistering receiver. We inferred that there was an IllegalArgumentException being thrown when unregistering a receiver. By investigating the code changes, we found a call of the method unregisterReceiver in the definition of the method onDestroy, so that might be where the exception was thrown. We also found that in the fix, the fixer moved the method unregisterReceiver from onDestroy to onPause and wrapped the method with a try-catch block. In the exception log message, the fixer also noted that the code might lead to an illegal state. With all the information, at the end, we considered the bug fix is an exception bug fix.

COMMIT MESSAGE: "fixed crashes when activity got destroyed when selecting pgp key."

```
. . .
super.onCreate(savedInstanceState);
setContentView(R.layout.manage_accounts);

+ if (savedInstanceState != null) {
+     String jid = savedInstanceState.getString(
+                                         STATE_SELECTED_ACCOUNT);
+     if (jid != null) {
+         try {
+             this.selectedAccountJid = Jid.fromString(jid);
+         } catch (InvalidJidException e) {
+             this.selectedAccountJid = null;
+         }
+     }
+ }

accountListView = (ListView) findViewById(R.id.account_list);
this.mAccountAdapter = new AccountAdapter(this, accountList);
accountListView.setAdapter(this.mAccountAdapter);
. . .
```

Fig. 3: Example 3

Figure 3 shows an example in which the bug fix was not considered as an exception bug fix. In the commit message, the fixer only noted that he fixed crashes when activity got destroyed when selecting pgp key. With this information, we were unable to locate exactly the what type of the crashes and where the crashes happened in the code. Although, in the code change, the fixer add a portion of code with a try-catch block, he also modified the code in other functions and files so we agreed that the bug fix should be too general to be recorded as an exception bug fix.

If a bug fix is considered as an exception bug fix, we recorded several information related to it for the purpose of our study, including the effects of the exception bugs, what types and methods causing exception, the exception types, the handling action of the fixer, the logging messages, etc. Finally, we identified 246 exception bug fixes across 10 projects. Column Fixes of Table I shows the number of exception bug fixes for each projects. Our dataset is available at: rebrand.ly/ExDataset.

B. Empirical Results

1) RQ1: Causes and Effects of Exceptions:

Finding 1.1. First, we focused on how exception bugs affect the running apps. Thus, we classified exception bugs into different types of effects including: CRASH, UNSTABLE, UNKNOWN. CRASH implies that the app crashed when an exception bug occurred. Figure 1 shows an example of an exception bug is classified as CRASH. The app crashed when calling the constructor of SpannableStringBuilder. UNSTABLE means that the app continued to run but in an unstable state or some features might not function properly. Figure 4 shows an example of an exception bug classified as UNSTABLE. The exception occurred in another thread instead of the main thread which caused the notifications not rendering generated avatars. If we did not have enough information to figure out the effect of an exception bug we labeled it as UNKNOWN.

We studied the number of exception bugs classified by the defined types. Exception bugs caused apps crashing in over 80% of the cases (199/246). Over 13% (33/246) of the

```
COMMIT MESSAGE: "Fix for lollipop notifications not rendering
generated avatars."
- bitmap = Bitmap.createBitmap(canvasWidth, canvasHeight,
                                          Bitmap.Config.ARGB_8888);
- Canvas canvas = new Canvas(bitmap);
- drawable.draw(canvas);
+ try {
+     bitmap = Bitmap.createBitmap(canvasWidth, canvasHeight,
+                                          Bitmap.Config.ARGB_8888);
+     Canvas canvas = new Canvas(bitmap);
+     drawable.setBounds(0, 0, canvas.getWidth(), canvas.getHeight());
+     drawable.draw(canvas);
+ } catch (Exception e) {
+     Log.w(TAG, e);
+     bitmap = null;
+ }
```

Fig. 4: Example 4

TABLE II: Top-10 Types

Types	#
android.app.Activity	14
android.content.Context	8
java.lang.Integer	5
android.database.sqlite.SQLiteDatabase	5
android.graphics.BitmapFactory	5
android.graphics.Bitmap	4
java.text.SimpleDateFormat	4
android.content.ContentResolver	3
android.database.sqlite.SQLiteDatabase	2
android.media.MediaPlayer	2
android.database.Cursor	2

TABLE III: Top-10 Exceptions

Exceptions	#
java.lang.Exception	54
java.lang.NullPointerException	25
java.lang.IllegalArgumentException	22
java.lang.OutOfMemoryError	19
android.content.ActivityNotFoundException	14
java.lang.NumberFormatException	9
java.lang.IllegalStateException	9
java.lang.Throwable	8
android.database.sqlite.SQLiteException	8
java.lang.ClassCastException	8

TABLE IV: Frequency of methods that causes exceptions

OutOfMemoryError	#
Bitmap.createBitmap	2
BitmapFactory.decodeFile	2
BitmapFactory.decodeByteArray	2
Byte.new	2
BitmapFactory.decodeResource	1
NumberFormatException	
Integer.parseInt	4
Long.parseLong	2
Integer.valueOf	1
ActivityNotFoundException	
Activity.startActivityForResult	5
Context.startActivity	4
Activity.startActivity	3
LinkMovementMethod.onTouchEvent	1
Fragment.startActivity	1

exception bugs caused the app running in a unstable state or some features might not function properly. There are 14 exception bugs are labeled as UNKNOWN as we did not have enough information to classify those bugs in one the two types above. From the statistics, we can see that exception bugs often cause serious problems for the apps such as crashing or running in a unstable state.

Finding 1.2. Most of exception bugs are caused by Android API methods. Table II shows top-10 types by the number of times the methods of those types cause exceptions. Our first observation is that all 10 classes are Android APIs (to save space, we do not show the full qualify name of each classes). Further investigated, we found out that 51% (127/246) of exception bugs is caused by Android API methods. Note that, in our study, we only identified which methods cause exceptions inside the try block of a bug fix. A third-party method that causes exception may used several Android API objects and methods inside its implementation, and the actual cause of exception might be from those API calls. Thus, in reality, we believe the percentage of exception bugs caused by Android APIs could be even higher. The finding suggests that using Android APIs could lead to considerable number of exception-related bugs.

Finding 1.3. Most of exceptions in bugs occurs by runtime exceptions. Recall that, in Java, exceptions are divided into three categories: checked exceptions, runtime (unchecked) exceptions, and runtime errors. Table III also top-10 exception types appear in the exception bug fixes. From the table, we have several observations. First, the java.lang.Exception class appears most often. Exception is the base class for all checked and unchecked exception in Java. An Exception can be checked or unchecked exception. This result could be explained as the fact that when catching exception, developers may not know or care about what type of exception is thrown and they just catch the most general exception. Another observation is that most of remaining exception types are runtime exceptions. There is only one type of runtime errors in the ranked list, which is OutOfMemoryError. Further investigation on the type of exceptions, we find out that in 246 exception bugs, there is 58.13% (143/246) of runtime exception, 7.31% (18/246) of checked exceptions, and 9.34% (23/246) of runtime errors. For the 62 remaining exception bug fixes, developers catch exceptions using general exception types such as Exception or Throwable, thus, we cannot identify types of exceptions in these fixes. The result indicates that the majority of exceptions in exception bugs are runtime exceptions. The finding suggests that an exception support tool should focus on predicting the occurrences of runtime exceptions.

Finding 1.4. There are associations between methods and exception types. Studying the dataset, we found that some exceptions have a very strong association with specific methods. Table IV shows frequency of methods that causes OutOfMemoryError, NumberFormatException and ActivityNotFoundException. From the table we can see that parsing numbers often throws NumberFormatException, starting an activity could introduce ActivityNotFoundException and using Bitmap often causes OutOfMemoryError. The reverse relationship is also true as all three exceptions are only occur by methods in the table.

2) RQ2: Exception Handling:

Finding 1.5. In this finding, we study what developers do in exception bug fixes. Based on the fixes of developers in the dataset, we classify 3 types of fixes that the fixers performed: catch the exception and do not perform any actions (SWALLOW), re-throw another exception to transfer handling jobs to other functions (RETHROW), invoke method calls or operations to handle exceptions (HANDLING). After analyzing each type of actions, we found that, in total, programmers did not perform any actions to handle exception in about 16% (40/246) bug fixes. They re-threw another exception in about 12.6% (31/246) of the cases. Finally, fixers handled exceptions in 41.86%(103/246) of the exception bug fixes. We label a bug fix as SWALLOW if programmers only add try-catch blocks to the portion of code that causes exception but the catch block in the fix does not contain any statements or only contains log statements. This action is also called swallowing exception. An example for this type of handling is shown in Figure 5.

```
- unregisterReceiver(mediaUpdate);
+ try {
+     unregisterReceiver(mediaUpdate);
+ } catch (IllegalArgumentException e) { }
```

Fig. 5: An example of swallowing an exception

In this example, the method unregisterReceiver throws an exception when it is executed and makes program crash. The programmer fixed the bug by adding a try-catch block to cover the method. This handling method will avoid the program from crashing when an exception occurs. In general, swallowing exception is a bad practice, the data and logic in program might change because of the exception, thus, continue running without modification could introduce new bugs to the program. Our results suggests that there are considerable exception bugs are fixed by swallowing exceptions.

Finding 1.6. In totals, programmers invoke method calls or operations to handle exceptions in 41.86%(103/246) of bug fixes. We label a bug fix as HANDLING if the catch block in that fix contains at least one statement other than a log statement. To further understanding how programmers handle exceptions, we categorized handling actions into several categories. We found that about 52% (54/103) of these bug fixes, programmers used default values to handle exceptions. One example of this handling type is if an exception occurs inside a method that returns an object, in the catch block of the fix, programmers handle exception by returning null. Or if exceptions occur while getting or creating object, he or she will assign the result variable to a default values such as null, true, 0, etc. Figure 8 shows an example of handling an exception by return null value. We also found that in 2 cases of the bug fixes, programmers created alerts to notify users about the bug that happened.

Programmers invoke method calls in catch block to handle exceptions in the remaining 47 bug fixes. After analyzing these bug fixes, we found that in almost all of these bug fixes (40/47), programmers invoke method calls of the same class with the method that causes exception. Let consider an example bug fix in Figure 6.

```
- in = new BufferedInputStream(conn.getInputStream());
+ try {
+     in = new BufferedInputStream(conn.getInputStream());
+ } catch (Exception ex) {
+     in = httpURLConnection.getErrorStream();
+ }
```

Fig. 6: An example of exception bug fix that invoke method calls

In this example, the call of method getInputStream causes exception, the programmer handle this exception by invoke function getErrorStream on the same object with getInputStream.

We further find patterns in the remaining 47 bug fixes. The patterns includes closing or releasing objects that hold resources such as Cursor or Activity (10), invoke method that causes exception again or with different parameters (12), create a new object to replace object that cause exception (4), and other actions (21). Examples for each pattern are shown in the Figure 7. This finding suggests that there are patterns in exception-handling actions of app developers.

Handling Type #1: Close or release objects that hold resources

```
- db.delete(COMMENTS_TABLE, "blogID=" + blogID, null);
+ try {
+   db.delete(COMMENTS_TABLE, "blogID=" + blogID, null);
+ } catch (Exception e) {
+   Log.i("WordPress", e.getMessage());
+   db.close();
+   return false;
+ }
```

Handling Type #2: Invoke methods again with different parameters

```
- postContent = new SpannableStringBuilder(mEditorFragment.getSpannedContent());
+ try {
+  postContent = new SpannableStringBuilder(mEditorFragment.getSpannedContent());
+ } catch (IndexOutOfBoundsException e) {
+  // A core android bug might cause an out of bounds exception, if so we'll just
use the current editable
+  // See https://code.google.com/p/android/issues/detail?id=5164
+  postContent = new SpannableStringBuilder(StringUtils.notNullStr((String)
mEditorFragment.getContent()));
+ }
```

Handling Type #3: Create new objects

```
Date d;
- d = (Date) thisHash.get("dateCreated");
- values.put("dateCreated", d.getTime());
+ try {
+  d = (Date) thisHash.get("dateCreated");
+  values.put("dateCreated", d.getTime());
+ } catch (Exception e) {
+  Date now = new Date();
+  values.put("dateCreated", now.getTime());
+ }
```

Handling Type #4: Combine multiple handling actions

```
+ try {
   LocationManager lm =
(LocationManager)this.getSystemService(Context.LOCATION_SERVICE);
   Location loc = lm.getLastKnownLocation(LocationManager.NETWORK_PROVIDER);
   double latitude = loc.getLatitude();
   double longitude = loc.getLongitude();
   Intent intent = new Intent(getApplicationContext(),PNDVActivity.class);
   intent.putExtra("latitude", latitude);
   intent.putExtra("longitude", longitude);
   startActivity(intent);
+ } catch (NullPointerException npe) {
+  // Android 4.2 NPEs here. Try again but still be careful
+  try {
+    Intent intent = new Intent(getApplicationContext(),PNDVActivity.class);
+    intent.putExtra("latitude", mLatitude);
+    intent.putExtra("longitude", mLongitude);
+    startActivity(intent);
+  } catch (Exception e) {
+    Intent intent = new Intent(getApplicationContext(),PNDVActivity.class);
+    startActivity(intent);
+  }
+ }
```

Fig. 7: Different types of handling actions

Finding 1.7. In several exception bug fixes, we found that programmers did not handle exceptions properly. Figure 8 shows an exception bug fix of Cursor object. In this example, an exception occurs when using Cursor object. To handle this bug, the programmer added a try-catch block to cover the code portion that uses the Cursor object, and returned null in the catch block. This action of handling may lead to memory leak bugs. After exception occurs and the function returns, the Cursor object still lives and holds system resources until it is collected by garbage collector. This may prevent other parts of the program access the resources. The proper way to handle this exception is calling close method on the cursor object before the return statement. The close method will release resources that hold by the cursor object before the function lost reference to it.

COMMIT MESSAGE: "catch exception when reading message id from database."

```
. . .
+ try {
      if (cursor.getCount() == 0) {
          return null;
      } else {
          cursor.moveToFirst();
          return new Pair<>(cursor.getLong(), cursor.getString(1));
      }
+ } catch (Exception e) {
+     return null;
+ }
```

Fig. 8: Example 6

Figure 9 shows an exception bug fix of the object MediaPlayer. As the MediaPlayer object is in an exception state, it is recommended to call release function immediately so that resources used by the internal player engine associated with the MediaPlayer object can be released immediately. Resource may include singleton resources such as hardware acceleration components and failure to call release may cause subsequent instances of MediaPlayer objects to fallback to software implementations or fail altogether.

COMMIT MESSAGE: “Prevented IllegalStateException when calling getDurationSafe()”

```
- return player.getDuration();
+ try {
+     return player.getDuration();
+ } catch (IllegalStateException e) {
+     e.printStackTrace();
+     return INVALID_TIME;
+ }
```

Fig. 9: Example 7

Finally, Figure 10 shows an exception bug fix of creating an bitmap image. When OutOfMemoryError occurs, it is recommended to not perform any handling actions. In this case, the fixer continued to call createBitmap which might cause OutOfMemoryError one more time. This finding suggests that programmers can still make mistakes when handling exceptions.

3) RQ3: Logging Exceptions:

Finding 8. In this finding, we focus on the logging actions of fixers in exception bug fixes. In totals, programmers add log statements in about 50% (121/246) of bug fixes. We labeled a bug fix has logging actions if programmers add at least one log statement in the catch block. Table V shows top-5 log APIs used in the bug fixes. From the table we can see

COMMIT MESSAGE: “sometimes fixes possible OOME”

```
. . .
- myBitmaps[iIndex] = Bitmap.createBitmap(myWidth, myHeight, Bitmap.Config.RGB_565);
+ try {
+   myBitmaps[iIndex] = Bitmap.createBitmap(myWidth, myHeight, Bitmap.Config.RGB_565);
+ } catch (Exception e) {
+   System.gc();
+   System.gc();
+   myBitmaps[iIndex] = Bitmap.createBitmap(myWidth, myHeight, Bitmap.Config.RGB_565);
+ }
```

Fig. 10: Example 8

that most of the time, programmers log using Android Log APIs (50%) and standard output stream (23%). Some projects use their custom log APIs such as WordPress uses AppLog and Crashlytics, Surespot uses SurespotLog. We further investigate on what programmers log when handling exceptions. We found that about in 76% (91/121) of bug fixes that have log statements, programmers only log messages indicates the exceptions or where the exceptions happens, i.e. stack trace. In the remaining 24% (30/121), programmers include more context information in logs such as value or information of objects when an exception happens. In general, including context information is a better logging scheme as it provides more runtime information and help developers in fixing exception bugs. This finding suggests that developers perform logging in half of the time fixing exception bugs while most of log statements are used to pinpoint the existence and location of exceptions

TABLE V: Top-5 Log APIs

Log APIs	Usage
android.util.Log	61
java.io.PrintStream	28
org.wordpress.android.util.AppLog	16
surespot.common.SurespotLog	4
android.widget.Toast	4

III. Study on Exception Handling Code

In this section, we describe our study on exception handling code of mobile apps. In particular, we focus on how professional app developers handle exceptions on high-quality Android apps. Results from the study could be used as guidelines on building exception handling support tools.

A. Dataset

TABLE VI: Data Statistics

Data Collection	
Number of apps	4,000
Number of classes	13,969,235
Number of methods	16,489,415
Number of bytecode instructions	341,912,624
Space for storing .dex files	19.9 GB

The dataset used in our study is summarized in Table VI. In total, we downloaded and analyzed 4000 top free apps from 36 categories in Google Play Store. To ensure the quality of the apps, the app extractor only downloaded apps has

TABLE VII: Top-10 exception types handled by developers

Exceptions	Frequency	%
java.lang.Exception	181095	30.9%
android.os.RemoteException	47051	8.05%
org.json.JSONException	45421	7.77%
java.lang.InterruptedException	20459	3.50%
java.lang.NoSuchFieldError	18305	3.31%
java.lang.NumberFormatException	17633	3.30%
PackageManager.NameNotFoundException	14022	3.01%
java.lang.IllegalArgumentException	13576	2.32%
android.database.sqlite.SQLiteException	12518	2.14%
java.lang.IllegalAccessException	11638	1.99%

TABLE VIII: Frequency of exception types against a method

Activity.startActivityForResult	#	%
android.content.ActivityNotFoundException	189	40.6%
java.lang.Exception	164	35.2%
java.lang.ClassNotFoundException	33	7.1%
pm.PackageManager.NameNotFoundException	31	6.7%
java.lang.SecurityException	10	2.1%
Cursor.moveToFirst		
android.database.sqlite.SQLiteException	4874	61.9%
java.lang.Exception	2020	25.6%
android.database.SQLException	264	3.3%
java.lang.Throwable	180	2.2%
android.database.sqlite.SQLiteFullException	130	1.6%
Integer.parseInt		
java.lang.NumberFormatException	7025	57.6%
java.lang.Exception	2860	23.4%
java.lang.IllegalArgumentException	397	3.26%
java.lang.Throwable	346	2.8%
java.lang.NullPointerException	206	1.8%

overall rating of at least 3 (out of 5). This filtering is based on assumption that the high-rating apps would have high quality of code, and thus, would have better exception handling mechanism. Next, we adopted techniques in [8] and [9] to extract exception handling instances from bytecode of the apps for our study.

B. Results

Finding 2.1: In this finding, we first study the most popular exception types handled by developers. We collected exception types in each catch block appears in the dataset. Table VII shows top-10 handled exceptions by frequency. From the table, we could see that the most frequent exception type is no surprises java.lang.Exception as developers can use it to catch any of its subclasses. More interestingly, 7/10 exception types in the list are runtime exceptions. There is only one checked exception (JSONException) and one error (NoSuchFieldError) in the list. The result suggests that developers spend most of their time to handle runtime exceptions. The result is consistent with the Finding 1.3 in our previous study as runtime exceptions are also most often cause exception bugs.

Finding 2.2: We then focus on the co-occurrence between methods/object types and exceptions. For this purpose, we selected three methods that we found often lead to exception bugs in the previous study, Activity.startActivityForResult, Cursor.moveToFirst, and Integer.parseInt. For each method, we collected top-5 exception types that co-occur with the methods by frequency, i.e. the method is in a try block while the exception is in the corresponding catch block. The result is shown in Table VIII. From the table, we have several interesting observations. First, in all three cases, the subjected method mostly co-occurs with the first two exception types in the ranked list. While the general java.lang.Exception is in the top-2 results, we could infer that each method mostly occurs with the top-1 exception in a rank list, such as ActivityNotFoundException for the method startActivityForResult. Second, we see that the methods mostly co-occurs with exceptions of the same topic or context. For example, The startActivityForResult method mostly co-occurs with exception types related to locating activities and classes such as Activity/Class/Name-NotFoundException. Similarly, moveToFirst mostly co-occurs with exception types related to database such as SQLiteException or SQLException. Third, we found that the result is consistent with the Finding 1.4 in our previous study. For example, in all exception bugs caused by not handling startActivityForResult in the previous study, the thrown exception type is ActivityNotFoundException. The result is similar for the moveToFirst, and parseInt. These observations suggest that there is high correlation between methods and exception types in terms of co-occurrence. Thus, we could use the co-occurrence between methods and exception types as a measure to predict which exceptions are likely to occur when using a method.

Finding 2.3: In this finding, we focus on how developers use method calls to handle exceptions. We first studied object types that often occur in handling code. We define an object type occurs in a handling code if at least an object of that type appears in the try block. Table IX shows top-10 object types occur in handling code by frequency. The second column shows the frequency. Note that we exclude several popular and too general object types such as android.util.Log or java.lang.String. From the table, we can see that the object types in the table relate to accessing files (FileOutputStream, File), connecting to database (SQLiteDatabase, Cursor), networking (HttpURLConnection, Uri), and app management (Context, Intent, Parcel. This implies that developers often write handling code for such topics.

In each handling code that an object type occurs, we also collected all the method calls on the corresponding object in the catch block. We stored those the method calls as a set. The second column shows the number of distinct sets for each object type. We could see most object types have more than 10 distinct method call sets that might appear in handling code. The more general object types such as Context, File tends to have more distinct method call sets. While more specific object types such as Parcel, Cursor use less combinations of method calls. We further calculate the average number of distinct method call sets for all object types and the value is 14.68.

The third column shows the percentage of a method call set that contains only one method in handling code. Overall, the

TABLE IX: Top-10 object types occur in handling code

Object Types	#Occur	#Sets	%1Method	Top-1	%	Top-2	%
android.content.Context	3078	37	96.1%	startActivity()	23.1%	deleteFile()	19.4%
java.io.File	2813	37	64.3%	delete()	30.1%	exist()	28.6%
org.json.JSONObject	2508	37	76.5%	new	49.7%	toString()	10.6%
sqlite.SQLiteDatabase	1871	21	93.8%	endTransaction()	78.1%	close()	7.3%
java.io.FileOutputStream	1371	10	98.1%	close()	86.8%	flush()	10.6%
android.content.Intent	1508	74	66.6%	new	49.4%	addFlags()	7.16%
android.os.Parcel	1270	2	99.4%	recycle()	99.4%	readString()	0.5%
java.net.HttpURLConnection	1220	17	97.5%	disconnect()	50.5%	getErrorStream()	36.9%
java.net.Uri	1175	13	97.9%	parse()	77.6%	toString()	16.3%
android.database.Cursor	672	14	95.3%	close()	85.8%	getPosition()	2.5%

value is over 60% across all object types. We could see that although the number of distinct method call sets are high, most of the sets contain only one method. Which implies that developers often only use 1 method of an object type in handling code.

Furthermore, the fifth and seventh columns show the top-2 method calls used in handling code along with their percentage of frequency. We have several observations from the statistics. First, the total frequency of top-2 method calls is often greater than 50%. It suggests that most of the time, developers only use one of the top-2 methods of an object type in handing code. The more general objects such as Context, File often have more spread of using method calls where there is not much different in frequency between the top-2 method calls. While other more specific object such as Cursor, Parcel, the top-1 method is mostly used.

IV. Discussions and Future Work

In our first study, we found that there are considerable number of exception-related bugs occurs in app developments. Those bugs often cause serious problems for the apps such as crashing or apps running in a unstable state. We also found that to fix those bugs, app developers still use bad practices such as swallowing exceptions. In several exception bug fixes, we also found that app developers did not handle exceptions properly. Thus, the results suggest the need of exception support tools to help app developers prevent exceptions from happening and assist developers to handle exceptions correctly.

In Finding 1.3, we found that most of exception bugs occurs by runtime exceptions. Thus, we should build models to predict potential runtime exceptions that might occur given a piece of code. Such models are useful to detect potentitial exception bugs. Finding 1.4 suggests that there are associations between methods and exception types in exception bugs. Some exceptions have a strong association with specific methods. In Finding 2.2, we studied co-occurrences between methods in try block and the exceptions. The result suggests that we could build a model to predict potential runtime exceptions using co-occurrences between methods and exceptions collected from large amount of code.

In Finding 1.6, we discovered that developers often use four main types of actions to fix exception bugs including close or release objects that hold resources, invoke methods with different parameters, create new objects, and combine multiple actions. In Finding 2.3, we also discovered that most of the time, professional app developers use just one main method call to when handling exceptions for an object. These two findings could provide several guidelines for building models to handle exceptions.

In the future, we plan to expand our work in several directions. First, similar bugs might have similar corresponding fixes. A bug could be identified and fixed if it matches other bugs that occurred in the history of the current project or other projects. Nguyen *et al.* [10] showed that there is repetitiveness in small size bug fixes. As exception bugs are a subset of general bugs, they are likely to share the same property. In the future, we plan to study the repetitiveness of exception bug fixes. From the findings we could desire approaches to automatically group similar exception bug fixes together.

Second, exception bugs could be exposed to developers differently. Expert developers have deep knowledge about the programming language, APIs, and the current working project. Thus, they are unlikely to introduce exception bugs to the system compared to inexperience or novice developers. On the other hand, developers might also have different preferences when fixing exception bugs. Some developers might prefer fixing exception bugs by swallowing the exceptions as shown in our study. Other fixers might prefer fixing bugs by adding defensive code. In our future work, we plan to study exception bugs and fixes from developer's standpoint. Insights from the study are useful for developing models for bug assignment and code recommendation.

V. Threats to Validity

The threat to internal validity includes errors when we identified exception bug fixes from the selected projects. Firstly, we might incorrectly identify reported bugs and on-demand bugs. For reported bugs, a commit that refers to an issue might not be a bug fix, thus, identifying bug fixes from issue numbers might introduce false-positive instances. For on-demand bugs, not all the bug fixes contains the keywords such as "fix", "fixes","bug", or "patch", thus, we might miss true-positive instances. To reduce this threat, we could apply more patterns (e.g. adding more keywords) when identifying bug fixes from commits. Secondly, we might incorrectly identify exception bug fixes from the set of candidate bug fixes. As the filtering process is manually carried out by three researchers, there are might be some errors when classifying bug fixes due to human errors. The threat could be reduced by adding more people to our labelling process.

The threat of external validity includes our selected projects. Although we analyzed 10 medium to large Android projects, the selected projects may still be limited to Android. It is likely that most our findings still hold for mobile platforms. To reduce this threat, our study should be replicated in future work by using projects other mobile platforms such as iOS. Also, the number of exception of bug fixes should be bigger for the result of the study more convincing. We plan to extend the exception bug fixes in our dataset in our future work.

VI. Related Work

There are various empirical studies on bugs and fixes, which focus on different aspects including bug fixing rate [11], bugs in build process [12], bugs in machine learnings systems [13], buffer overflow bugs [14], effects of expert knowledge [15], supplementary bug fixes [16], dormant bugs [17], javascript [18], [19], repreated bug fixes [20], bug linking [21], testing [22], [23], [24], and bugs in distributed systems [25], refactoring [26]. Zhong *et al.* [4] performed an empirical study on real bug fixes. Yin *et al.* [27] performed a study show that bug fixes could introduce new bugs. Nguyen *et al.* [10] showed that repetitiveness is commons in small size bug fixes. Eyolfson *et al.* [28] showed that the bugginess of a commit is correlated with the commit time. Bird *et al.* [29] showed that many projects did not carefully maintained the links between bug reports and bug fixes. In [30], they studied thether bug fixes recorded in these historical datasets a fair representation of the full population of bug fixes. Thung *et al.* [31] manually examined bug fixes; their results show that faults are not localized. Ray *et al.* [32] studied the naturalness of buggy code. Canfora et at. [33] studied the survival time by extracting, from versioning repositories, changes introducing and fixing bugs. Pan *et al.* [6] developed a technique to extract 27 bug fix patterns using the syntax components and context of the source code involved in bug fix changes. Other researches on bugs and fixes include [34], [35], [36], [37], [38], [39], [5]. Due to space limitation, we could not describe each research in details.

There are several studies empirical studies on exception handling. To the best of our knowledge, the closest work to our study are [40] and [3]. Ebert *et al.* [40] presented an exploratory study on exception handling bugs by surveying of 154 developers and an analysis of 220 exception handling bugs from two Java programs, Eclipse and Tomcat. While our study focuses on exception bugs specifically for Android app development. Coelho *et al.* [3] performed a detailed empirical study on exception-related issues of over 6,000 Java exception stack traces extracted from over 600 open source Android projects. Our empirical study is based on real exception bugs and fixes and handling code of app developers, while the study in [3] mostly focus on exception stack traces.

Pádua *et al.* [41] investigated the relationship between software quality measured by the probability of having post-release defects with exception flow characteristics and exception handling anti-patterns. In [42], they studied exception handling practices with exception flow analysis. Kechagia *et al.* [43] investigated the exception handling mechanisms of the Android platforms API to understand when and how developers use exceptions. In [44], they examined Java exceptions and propose a new exception class hierarchy and compile-time mechanisms that take into account the context in which exceptions can arise. In [45], they showed that a significant number of crashes could have been caused by insufficient documentation concerning exceptional cases of Android API. Bruntink *et al.* [46] provided empirical data about the use of an exception handling mechanism based on the return code idiom in an industrial setting. Coelho *et al.* [47] studied exception handling bug hazards in Android based on GitHub and Google code issues. In [48], they studied exception handling guidelines adopted by Java developers.

Exception handling recommendation has been studied in several researches [49], [50], [51], [52], [53], [54]. Barbosa *et al.* [50] proposed a set of three heuristic strategies used to recommend exception handling code. In [51], they proposed RAVEN, a heuristic strategy aware of the global context of exceptions that produces recommendations of how violations in exception handling may be repaired. In [52], they presented a DSL to specify and verify exception handling policies. Rahman *et al.* [49] proposed a context-aware approach that recommends exception handling code examples from a number of GitHub projects. Filho *et al.* [54] proposed ArCatch, an architectural conformance checking solution to deal with the exception handling design erosion. Lie et al. [55] proposed an approach, named EXPSOL, which recommends online threads as solutions for a newly reported exception-related bug.

There exist several methods for mining exception-handling rules. WN-miner [56] and CAR-miner [57] are approaches that use association mining techniques to mine association rules between method calls of try and catch blocks in exception handling code. Both models are used to detect bugs related to exceptions. Zhong *et al.* [58] proposed an approach named MiMo, that mines repair models for exception-related bugs.

VII. Conclusions

Exceptions are unexpected errors occurring while an app is running. Learning to handle exceptions correctly is often challenging for mobile app developers due to the fast-changing nature of API frameworks for mobile systems and the insufficiency of API documentation. Failing to handle exceptions could lead to serious bugs and issues such as crashing, or causing negative end-user experiences. To understand the nature of exceptions in app development and how app developers handle them, in this paper, we performed a detailed empirical study of exception handling bugs and fixes 246 exception-related bugs and fixes from 10 mobile apps. We discovered eight findings provide insights about exception bugs and fixes of mobile apps based on three research questions. We also performed another study on how professional developers handle exceptions on 4000 high-quality mobile apps. The study provides useful guidance on building exception handling support tools.

References

[1] W. Weimer and G. C. Necula, "Finding and preventing run-time error handling mistakes," in *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '04. New York, NY, USA: ACM, 2004, pp. 419–431. [Online]. Available: http://doi.acm.org/10.1145/1028976.1029011

[2] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "Api change and fault proneness: A threat to the success of android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 477–487. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491428

[3] R. Coelho, L. Almeida, G. Gousios, and A. van Deursen, "Unveiling exception handling bug hazards in android based on github and google code issues," in *MSR*, 2015.

[4] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 913–923. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818864

[5] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting faults from cached history," in *ICSE*, 2007.

[6] K. Pan, S. Kim, and E. J. Whitehead, Jr., "Toward an understanding of bug fix patterns," *Empirical Softw. Engg.*, vol. 14, no. 3, pp. 286–315, Jun. 2009. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9077-5

[7] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, ser. ICSM '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 120–. [Online]. Available: http://dl.acm.org/citation.cfm?id=850948.853410

[8] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Recommending api usages for mobile apps with hidden markov model," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, 2015.

[9] T. Nguyen, P. Vu, H. Pham, and T. Nguyen, "Poster: Recommending exception handling patterns with exassist," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, May 2018, pp. 282–283.

[10] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 180–190.

[11] W. Zou, X. Xia, W. Zhang, Z. Chen, and D. Lo, "An empirical study of bug fixing rate," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2, July 2015, pp. 254–263.

[12] X. Zhao, X. Xia, P. S. Kochhar, D. Lo, and S. Li, "An empirical study of bugs in build process," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ser. SAC '14. New York, NY, USA: ACM, 2014, pp. 1187–1189. [Online]. Available: http://doi.acm.org/10.1145/2554850.2555142

[13] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, Nov 2012, pp. 271–280.

[14] T. Ye, L. Zhang, L. Wang, and X. Li, "An empirical study on detecting and fixing buffer overflow bugs," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, April 2016, pp. 91–101.

[15] D. Huo, T. Ding, C. McMillan, and M. Gethers, "An empirical study of the effects of expert knowledge on bug reports," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sep. 2014, pp. 1–10.

[16] J. Park, M. Kim, B. Ray, and D. Bae, "An empirical study of supplementary bug fixes," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, June 2012, pp. 40–49.

[17] T.-H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan, "An empirical study of dormant bugs," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 82–91. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597108

[18] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "An empirical study of client-side javascript bugs," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Oct 2013, pp. 55–64.

[19] J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei, "A comprehensive study on real world concurrency bugs in node.js," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 520–531. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155628

[20] R. Yue, N. Meng, and Q. Wang, "A characterization study of repeated bug fixes," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2017, pp. 422–432.

[21] T. F. Bissyande, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Reveillere, "Empirical evaluation of bug linking," in *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, ser. CSMR '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 89–98. [Online]. Available: http://dx.doi.org/10.1109/CSMR.2013.19

[22] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: An empirical investigation," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 547–558.

[23] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ser. ICSME '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 101–110. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2015.7332456

[24] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, pp. 560–564.

[25] Y. Gao, W. Dou, F. Qin, C. Gao, D. Wang, J. Wei, R. Huang, L. Zhou, and Y. Wu, "An empirical study on crash recovery bugs in large-scale distributed systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 539–550. [Online]. Available: http://doi.acm.org/10.1145/3236024.3236030

[26] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 104–113. [Online]. Available: https://doi.org/10.1109/SCAM.2012.20

[27] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 26–36. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025121

[28] J. Eyolfson, L. Tan, and P. Lam, "Correlations between bugginess and time-based commit characteristics," *Empirical Softw. Engg.*, vol. 19, no. 4, pp. 1009–1039, Aug. 2014. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9245-0

[29] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: Bias in bug-fix datasets," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 121–130. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595716

[30] ——, "Fair and balanced? bias in bug-fix datasets," in *Proceedings of the the Seventh joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. Association for Computing Machinery, Inc., August 2009.

[31] Lucia, F. Thung, D. Lo, and L. Jiang, "Are faults localizable?" in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, June 2012, pp. 74–77.

[32] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 428–439. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884848

[33] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta, "How long does

a bug survive? an empirical study," in *2011 18th Working Conference on Reverse Engineering*, Oct 2011, pp. 191–200.

[34] B. Zhou, I. Neamtiu, and R. Gupta, "A cross-platform analysis of bugs and bug-fixing in open source projects: Desktop vs. android vs. ios," in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '15. New York, NY, USA: ACM, 2015, pp. 7:1–7:10. [Online]. Available: http://doi.acm.org/10.1145/2745802.2745808

[35] G. Rodríguez-Pérez, A. Zaidman, A. Serebrenik, G. Robles, and J. M. González-Barahona, "What if a bug has a different origin?: Making sense of bugs without an explicit bug introducing change," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '18. New York, NY, USA: ACM, 2018, pp. 52:1–52:4. [Online]. Available: http://doi.acm.org/10.1145/3239235.3267436

[36] A. Takahashi, N. Sae-Lim, S. Hayashi, and M. Saeki, "A preliminary study on using code smells to improve bug localization," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: ACM, 2018, pp. 324–327. [Online]. Available: http://doi.acm.org/10.1145/3196321.3196361

[37] H. Wang and H. Kagdi, "A conceptual replication study on bugs that get fixed in open source software," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2018, pp. 299–310.

[38] Y. Wang, N. Meng, and H. Zhong, "An empirical study of multi-entity changes in real bug fixes," *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 287–298, 2018.

[39] P. Bhattacharya, L. Ulanova, I. Neamtiu, and S. C. Koduru, "An empirical analysis of bug reports and bug fixing in open source android apps," in *CSMR*, 2013.

[40] F. Ebert, F. Castor, and A. Serebrenik, "An exploratory study on exception handling bugs in java programs," *J. Syst. Softw.*, vol. 106, no. C, pp. 82–101, Aug. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2015.04.066

[41] G. B. de Pádua and W. Shang, "Studying the relationship between exception handling practices and post-release defects," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: ACM, 2018, pp. 564–575. [Online]. Available: http://doi.acm.org/10.1145/3196398.3196435

[42] G. B. d. Pdua and W. Shang, "Revisiting exception handling practices with exception flow analysis," in *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sep. 2017, pp. 11–20.

[43] M. Kechagia, M. Fragkoulis, P. Louridas, and D. Spinellis, "The exception handling riddle: An empirical study on the android api," *Journal of Systems and Software*, vol. 142, 04 2018.

[44] M. Kechagia, T. Sharma, and D. Spinellis, "Towards a context dependent java exceptions hierarchy," in *Proceedings of the 39th International Conference on Software Engineering Companion*, ser. ICSE-C '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 347–349. [Online]. Available: https://doi.org/10.1109/ICSE-C.2017.134

[49] M. M. Rahman and C. K. Roy, "On the use of context in recommending exception handling code examples," in *SCAM*, 2014.

[45] M. Kechagia and D. Spinellis, "Undocumented and unchecked: Exceptions that spell trouble," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 312–315. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597089

[46] M. Bruntink, A. van Deursen, and T. Tourwé, "Discovering faults in idiom-based exception handling," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 242–251. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134320

[47] R. Coelho, L. Almeida, G. Gousios, and A. van Deursen, "Unveiling exception handling bug hazards in android based on github and google code issues," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 134–145. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820536

[48] H. Melo, R. Coelho, and C. Treude, "Unveiling exception handling guidelines adopted by java developers," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2019, pp. 128–139.

[50] E. A. Barbosa, A. Garcia, and M. Mezini, "Heuristic strategies for recommendation of exception handling code," in *Brazilian Symposium on Software Engineering*, 2012.

[51] E. A. Barbosa and A. Garcia, "Global-aware recommendations for repairing violations in exception handling," *TSE*, 2017.

[52] E. A. Barbosa, A. Garcia, M. P. Robillard, and B. Jakobus, "Enforcing exception handling policies with a domain-specific language," *TSE*, 2016.

[53] T. Montenegro, H. Melo, R. Coelho, and E. Barbosa, "Improving developers awareness of the exception handling policy," in *SANER*, 2018.

[54] J. L. M. Filho, L. Rocha, R. Andrade, and R. Britto, "Preventing erosion in exception handling design using static-architecture conformance checking," in *Software Architecture*, 2017.

[55] X. Liu, B. Shen, H. Zhong, and J. Zhu, "Expsol: Recommending online threads for exception-related bug reports," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, Dec 2016, pp. 25–32.

[56] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection," in *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'05, Berlin, Heidelberg, 2005.

[57] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09, Washington, DC, USA, 2009.

[58] H. Zhong and H. Mei, "Mining repair model for exception-related bug," *Journal of Systems and Software*, vol. 141, pp. 16 – 31, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121218300505