Repairing Crashes in Android Apps

Shin Hwei Tan
Southern University of Science and Technology
shinhwei@hotmail.com

Xiang Gao
National University of Singapore
gaoxiang@comp.nus.edu.sg

Zhen Dong
National University of Singapore
zhengdong@comp.nus.edu.sg

Abhik Roychoudhury
National University of Singapore
abhik@comp.nus.edu.sg

ABSTRACT

Android apps are omnipresent, and frequently suffer from crashes — leading to poor user experience and economic loss. Past work focused on automated test generation to detect crashes in Android apps. However, automated repair of crashes has not been studied. In this paper, we propose the first approach to automatically repair Android apps, specifically we propose a technique for fixing crashes in Android apps. Unlike most test-based repair approaches, we do not need a test-suite; instead a single failing test is meticulously analyzed for crash locations and reasons behind these crashes. Our approach hinges on a careful empirical study which seeks to establish common root-causes for crashes in Android apps, and then distills the remedy of these root-causes in the form of eight generic transformation operators. These operators are applied using a search-based repair framework embodied in our repair tool Droix. We also prepare a benchmark DroixBench capturing reproducible crashes in Android apps. Our evaluation of Droix on DroixBench reveals that the automatically produced patches are often syntactically identical to the human patch, and on some rare occasion even better than the human patch (in terms of avoiding regressions). These results confirm our intuition that our proposed transformations form a sufficient set of operators to patch crashes in Android.

CCS CONCEPTS

• Software and its engineering → Automatic programming; Software testing and debugging; Dynamic analysis;

KEYWORDS

Automated repair, Android apps, Crash, SBSE

1 INTRODUCTION

Smartphones have become pervasive, with 492 millions sold worldwide in the year of 2011 alone [21]. Users tend to rely more on their smartphones to conduct their daily computing tasks as smartphones are bundled with various mobile applications. Hence, it is important to ensure the reliability of apps running in their smartphones.

Testing and analysis of mobile apps, with the goal of enhancing reliability, have been studied in prior work. Some of these works focus on static and dynamic analysis of mobile apps [2, 7, 18, 56], while other works focus on testing of mobile apps [3, 4, 30, 31, 43]. To further improve the reliability of mobile applications, several approaches go beyond automated testing of apps by issuing security-related patches [6, 39]. While fixing security-related vulnerabilities is important, a survey revealed that most of the respondents have experienced a problem when using a mobile application, with 62 percent of them reported a crash, freeze or error [1]. Indeed, frequent crashes of an app will lead to negative user experience and may eventually cause users to uninstall the app. In this paper, we study automated approaches which alleviate the concern due to app crashes via the use of automated repair.

Recently, several automated program repair techniques have been introduced to reduce the time and effort in fixing software errors [24, 28, 35, 40, 42, 52]. These approaches take in a buggy program $P$ and some correctness criterion in the form of a test-suite $T$, producing a modified program $P'$ which passes all tests in $T$. Despite recent advances in automated program repair techniques, existing approaches cannot be directly applied for fixing crashes found in mobile applications due to various challenges.

The key challenge in adopting automated repair approaches to mobile applications is the quality of the generated patches is heavily dependent on the quality of the given test suite. Indeed, any repair technique tries to patch errors to achieve the intended behavior. Yet, in reality, the intended behavior is incompletely specified, often through a set of test cases. Thus, repair methods attempt to patch a given buggy program, so that the patched program passes all tests in a given test-suite $T$ (We call repair techniques that use test cases to drive the patch generation process test-driven repair). Unsurprisingly, test-driven repair may not only produce incomplete fixes but the patched program may also end up introducing new errors, because the patched program may fail tests outside $T$, which were previously passing [45, 49]. Meanwhile, several unique properties of test cases for mobile applications pose unique challenges for test-driven repair. First, regression test cases may not be available for a given mobile app $A$. While prior researches on automated test generation for mobile apps could be used for generating crashing...
inputs, regression test inputs that ensure the correct behaviors of A are often absent. Secondly, instead of simple inputs, test inputs for mobile apps are often given as a sequence of UI commands (e.g., clicks and touches) leading to crashes in the app. Meanwhile, GUI tests are often flaky [29, 36]: their outcome is non-deterministic for the same program version. As current repair approaches rely solely on the test outcomes for their correctness criteria, they may not be able to correctly reproduce tests behavior and subsequently generate incorrect patches due to flaky tests.

Another key challenge in applying recent repair techniques to mobile applications lies on their reliance on the availability of source code. However, mobile applications are often distributed as standard Android .apk files since the source code for a given version of a mobile app may not be directly accessible nor actively maintained. Moreover, while previous automated repair techniques are applied for fixing programs used by developers and programmers, mobile applications may be utilized by general non-technical users who may not have any prior knowledge regarding source code and test compilations.

We present a novel framework, called Droix for automated repair of crashes in Android applications. In particular, our contributions can be summarized as follows:

**Android repair:** We propose a novel Android repair framework that automatically generates a fixed APK given a buggy APK and a UI test. Android applications were not studied in prior work in automated program repair; but various researches on analysis [2, 7, 18, 56] and automated testing [3, 4, 30, 31, 43] illustrate the importance of ensuring the reliability of Android apps.

**Repairing UI-based test cases:** Different from existing repair approaches based on a set of simple inputs, our approach fixes a crash with a single UI event sequence. Specifically, we employ techniques allowing end users to reproduce the crashing event sequence by recording user actions on Android devices instead of writing test codes. The crashing input could be either recorded manually by users or automatically generated by GUI testing approaches [30, 47].

**Lifecycle-aware transformations** Our approach is different from existing test-driven repair approaches since it does not seek to modify a program to pass a given test-suite. Instead, it seeks to repair the crashes witnessed by a single crashing input, by employing program transformations which are likely to repair the root-causes behind crashes. We introduce a novel set of lifecycle-aware transformations that could automatically patch crashing Android apps by using management rules from the activity lifecycle and fragment lifecycle.

**Evaluation:** We propose DroixBench, a set of 24 reproducible crashes in 15 open source Android apps. Our evaluation on 24 defects shows that Droix could repair 15 bugs, and seven of these repairs are syntactically equivalent to the human patches.

## 2 BACKGROUND: LIFECYCLE IN ANDROID

Different from Java programs, Android applications do not have a single main method. Instead, Android apps provide multiple entry points such as `onCreate` and `onStart` methods. Via these methods, Android framework is able to control the execution of apps and maintain their lifecycle.

![Figure 1: Activity Lifecycle, Fragment Lifecycle and the Activity-Fragment Coordination](image-url)

Figure 1 shows the lifecycles of activity and fragment in Android. Each method in Figure 1 represents a lifecycle `callback`, a method that gets called given a change of state. Lifecycle transition obeys certain principles. For instance, an activity with the paused state could move to the resumed state or the stopped state, or may be killed by the Android system to free up RAM.

A `fragment` is a portion of user interface or a behavior that can be put in an Activity. Each fragment can be modified independently of the `host activity`(activity containing the fragment) by performing a set of changes. For a fragment, it goes through more states than an Activity from being launched to the active state, e.g., `onAttach` and `onCreateView` states.

The communication between an activity and a fragment needs to obey certain principles. A fragment is embedded in an activity and could communicate with its host activity after being attached. The allowed states of a fragment are determined by the state of its host activity. For instance, a fragment is not allowed to reach the `onStart` state before its host activity enters the `onStart` state. A violation of these principles may cause crashes in Android apps.

## 3 A MOTIVATING EXAMPLE

We illustrate the workflow of our automated repair technique by showing an example app, and its crash. The crash occurred in Transistor, a radio app for Android with 63 stars in GitHub. According to the bug report1, Transistor crashes when performing the event sequence shown in Figure 2: (a) starting Transistor; (b) shutting it down by pressing the system back button; (c) starting Transistor again and changing the icon of any radio station. Then, it crashes with a notification “Transistor keeps stopping”.(d) Listing 1 shows the log relevant to this crash. The stack trace information in Listing 1 suggests that the crash is caused by `IllegalStateException`.

---

1[https://github.com/y20k/transistor/issues/21](https://github.com/y20k/transistor/issues/21)
Our automated repair framework, Droix, performs analysis of the Activity-Fragment coordination (dashed lines in Figure 1) and reports potential violations in the communication between a fragment and its host activity. Our manual analysis of the source code for this app further reveals that the crash occurs because the fragment attempts to call an inherited method `startActivityForResult` at line 482, which indirectly invokes a method of its host activity.

However, the fragment is detached from the previous activity during the termination of the app and needs to be attached to a new activity in the restarting app. The method invocation occurs before the new activity has been completely created and leads to the crash.

```
FATAL EXCEPTION: main Process: org.y20k.transistor, PID: 2416
java.lang.IllegalStateException
at y20k...selectFromImagePicker(MainActivityFragment.java:482)
```

Listing 1: Stack trace for the crash in Transistor

```
482: startActivityForResult(pickImageIntent, REQUEST_LOAD_IMAGE);
```

Listing 2: Droix’s patch for the crash in Transistor

```
startActivityForResult(pickImageIntent, REQUEST_LOAD_IMAGE);
```

Listing 3: Developer’s patch for the crash in Transistor

```
if (getActivity() != null) {
    startActivityForResult(pickImageIntent, REQUEST_LOAD_IMAGE);
```

Droix defines specific repair operators based on our study of crashes in Android apps and the Android API documentation (see Section 4). One of the transformation operators identified through our study, `GetActivity-check`, is designed to check if the activity containing the fragment has been created. The condition `getActivity() != null` prevents the scenario where a fragment communicates with its host activity before the activity is created.

Listings 2 shows the patch automatically generated by Droix. With the patch, method `startActivityForResult` will not be invoked if the host activity has not been created. The related function (i.e., changing station icon) works well after our repair. In contrast, although the developer’s patch does not crash on the given input, it introduces regressions. Listing 3 shows the developer’s patch where `mActivity` is a field of the fragment referencing its host activity. When restarting the app, this field still points to the previously attached activity. The developer’s patch explicitly invokes `startActivityForResult` method of the previously attached activity instead of the newly created activity. After applying the developer’s patch, a user reports that the system back button no longer functions correctly when changing the station icon (i.e., pressing the back button does not close the app but mistakenly opens a window for selecting images). Specifically, the user reports the following event sequence when the app fails to function properly: open Transistor → tap to change icon → press back twice → open Transistor → tap to change icon → press back twice. We test the APK generated by Droix with this event sequence, and we observe that our fixed APK does not exhibit the faulty behavior reported by the user. Hence, we believe that the patch generated by Droix works better than the developer’s patch.

## 4 IDENTIFYING CAUSES OF CRASHES IN ANDROID APPLICATIONS

To study the root causes of crashes in Android apps, we manually inspect Android apps on GitHub and API documentation (as prior work has showed success in finding bugs via API documentation [48]). Our goal is to identify a set of common causes for Android crashes. We first obtain a set of popular Android apps by crawling GitHub and searching for the word “android app” written in Java using the GitHub API. For each app repository, we search for closed issues (resolved bug report) with the word “crash”. We focus on closed issues because those issues have been confirmed by the developers and are more likely to contain fixes for the crashes. From the list of closed issues on app crashes, we further extract issues that contain at least one corresponding commit associated with the crash. The final output of our crawler is a list of crash-related closed issues that have been fixed by the developers. Overall, our crawler searches through 7691 GitHub closed issues where 1155 (15%) of these issues are related to crashes. The relatively high percentage of crash-related issues indicates the prevalence of crashes in Android apps. Among these 1155 issues, 107 of these issues from 15 different apps have corresponding bug-fixing commits. We manually analyzed all issues and attempted to answer two questions:

**Q1:** What are the possible root causes and exceptions that lead to crashes in Android apps?

**Q2:** How does the complexity of activity/fragment lifecycle affect crashes in Android apps?

We study Q2 because a survey of Android developers suggests that the topmost reasons (47%) for `NullPointerException` in Android apps occur due to the complexity of activity/fragment lifecycle [18]. Our goal is to identify a set of generic transformations that are often used by Android developers in fixing Android apps. To gain deeper understanding of the root causes of each crash (Q1) and to identify the affect of activity/fragment lifecycle on the likelihood of introducing crashes (Q2), we manually examine lifecycle management rules in the official Android API documents [3].

Our study shows that the most common exceptions are:

- `NullPointerException (40.19%)`
- `IllegalStateException (7.48%)`

---

https://developer.github.com/v3/

The high percentage of NullPointerExceptions confirms with the findings of prior study of Android apps [18].

Table 1 shows the common root causes of crashes in Android apps we investigated. Column "Category" in Table 1 describes the high-level causes of the crashes, while the "Specific reasons" column gives the specific causes that lead to the crash. The last column (Category Total (%)) presents the total percentage of issues that fits into a particular category. Overall, 14.02% of crashes in our study occur due to the violation of management rules for Android Activity/Fragment lifecycles. The reader can refer to Section 5 on the explanation of these lifecycle-related crashes. Meanwhile, 16.82% of the investigated crashes are due to improper handling of resources, including resources either not available (Resource-related) or limited resources like memory (Resource limit). Furthermore, improper handling of callbacks contributes to 17.76% of crashes. Note that this “Callback” category denotes implementation-specific problems of different components in Android library (e.g., Activity, View and Intent). Among 40.19% of NullPointerException Exceptions thrown in these crashing apps, only 12.15% is related to missing the check for null objects (Missing Null-check). Interestingly, 4.67% of the GitHub issues include comments by Android developers acknowledging the fact that the patch issued are merely temporary fixes (Workaround) for these crashes that may require future patches to completely resolve the crash.

Overall, Table 1 shows that the complexity of activity/fragment lifecycle and incorrect resource handling are two serviceable causes of crashes in Android apps. Moreover, “Missing Null-check” in the “Other” category also often leads to crashes in Android apps.

5 STRATEGIES TO RESOLVE CRASHES

Our manual analysis of crashes in Android apps identifies eight program transformation operators which are useful for repairing these crashes. Table 2 gives an overview of each operator derived through our analysis. As “Missing Null-check” is one of the common causes of crashes in Table 1, we include this operator (S7: Null-check) in our set of operators. Another frequently used operator (5%) that fixes crashes that occur across different categories in Table 1 is inserting exception handler (S8: Try-catch) which we also include into our set of operators. We now proceed to discuss other program transformation operators in Table 2 and the specific reasons of crashes associated with each operator in this section.

Retain stateful object Configuration changes (e.g., phone rotation and language) cause activity to be destroyed and recreated which allows apps to adapt to new configuration (transition from onDestroy() to onCreate()). According to Android documentation 4, developer could resolve this kind of crashes by either (1) retaining a stateful object when the activity is recreated or (2) avoiding the activity recreation. We choose the first strategy because it is more flexible as it allows activity recreations instead of preventing the activity recreation. Listing 4 presents an example that explains how we retain the Option object by using the saved instance after the configuration changes to prevent null reference of the object (S2: Retain object).

```java
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    + setRetainInstance(true); // retain this fragment
    + new field for saving the object
    + private static Option saveOption;

    public View onCreateView(LayoutInflater inflator, ViewGroup container, Bundle savedInstanceState) {
        // saving and loading the object
        + if (option!=null) { saveOption = option; }
        + else { option = saveOption; }
        switch (option.getButtonStyle()) {
            //crashing point
        }
```
Commit transactions Each fragment can be modified independently of the host activity by performing a set of changes. Each set of changes that we commit (perform requested modifications atomically) to the activity is called a transaction. Android documentation specifies rules to prohibit committing transactions at certain stages of the lifecycle. Transactions that are committed in disallowed stages will cause the app to throw an exception. For example, invoking commit() after onSaveInstanceState() will lead to IllegalStateException since the transaction could not be recorded during this stage. We employ two strategies for resolving the incorrect commits: (S6: Move stmt) moving commit() to a legal callback (e.g., onPostResume()), (S4: Replace method) replacing commit() with commitAllowingStateLoss().

Communication between activity and fragment The lifecycle of a fragment is affected by the lifecycle of its host activity. For example, in Figure 1, when an activity is created (onCreate()), the fragment cannot proceed beyond the onActivityCreated() stage. Invoking getActivity() in the illegal stage of the lifecycle will return null, since the host activity has not been created or the fragment is detached from its host activity. A NullPointerException may be thrown in the following execution. We employ two strategies for resolving this problem: (S3: GetActivity-check) inserting condition if(getActivity()==null), and (S6: Move stmt) moving getActivity() to another stage (when the host activity is created and the fragment is not detached from the host activity) of the fragment lifecycle.

Retrieve wrong resource id Android resources are the additional files and static content used in Android source code (e.g., bitmaps, and layout). A resource id is of the form R.x.y where x refers to the type of resource and y represents the name of the resource. The resource id is defined in XML files and it is the parameter of several Android APIs (e.g., findViewById(id) and setText(id)). Android developers may mistakenly use a non-existing resource id which leads to Resources$NotFound exception. Listing 5 shows a scenario where the developers change the string resource id (S3: Replace resource id).

Listing 5: Example of handling crashes due to wrong resource id

```
int msgStrId = R.string.confirmation_remove_alert;
int msgStrId = R.string.confirmation_remove_file_alert;
```

Incorrect type-cast of resource To implement UI interfaces, an Android API (findViewById(id)) could be invoked to retrieve widgets (view) in the UI. As each widget is identified by attributes defined in the corresponding XML files, an Android developer may misinterpret the correct type of a widget, resulting in crashes due to ClassCastException. We repair the crash by replacing the type cast expression with correct type (S5: Replace cast). Listing 6 shows an example where the ImageButton object is incorrectly type caster.

Listing 6: Example fix for incorrect resource type-cast

```
mDefinition = (TextView) findViewById(R.id.definition);
mDefinition = (ImageButton) findViewById(R.id.definition);
```

6.1 Test with UI Sequences

Existing techniques in automated program repair typically rely on unit tests [32] or test scripts [28, 35, 53] to guide repair process. As additional UI tests for checking correctness are often unavailable, Droix uses user event sequences (e.g., clicks and touches) as input to repair buggy apps, which introduces new challenges: (1) these event sequences are often not included as part of the source code repository and reproducing these event sequences is often time-consuming; (2) ensuring that a recorded sequence has been reliably replayed multiple times is difficult as UI tests tend to be flaky (the test execution results may vary for the same configuration).

To reduce manual effort in obtaining UI sequences, Droix supports several kinds of event sequences, including: (1) a set of actions (e.g., clicks, and touches) leading to the crash which can be recorded using monkeyrunner; (2) a set of Android Debug Bridge (adb) commands; and (3) scripts with a mixture of recorded actions and adb commands. Non-technical users could record their actions with monkeyrunner while Android developers could write adb commands to have better control of the devices (e.g., rotate screen).

Droix employs several strategies to ensure that the UI test outcome is consistent across different executions [36]. Specifically, for each UI test, Droix automatically launches the app from the home screen, inserts pauses in between each event sequence, terminates
the apps after test execution, and brings the android device back to home screen (ensure that the last state of the device is the same as the initial state of the device). Moreover, Droix executes each UI test for at least three times in which each test execution has pauses of different duration (5, 10, 15 seconds) inserted in between events.

6.2 Fault Localization

Our fault localization step pinpoints faulty program locations leading to the crash. Since our approach does not require source code nor heavy test suite, we leverage stack trace information for fault localization. The stack trace contains (1) the type of exceptions being thrown, (2) the specific lines of code where the exception is thrown, and (3) the list of classes and method calls in the runtime stack when the exception occurs. We use stack trace information for fault localization because (1) this information is often included in the bug report of crashes (which allows us to compare the actual exception thrown with the expected exception) and (2) prior study has shown the effectiveness of using stack trace to locate Java runtime exceptions [44]. The stack trace information is given to our search algorithm for fix localization. When searching for complex fixes, once a fix using initial stack trace is generated, it may enable other crashes, leading to new stack traces and new fixes.

6.3 Code Checker and Test Checker

Instead of relying solely on the UI test outcome, Droix enforces two kinds of properties: code-level properties (properties that are checked prior to test execution) and test-level properties (properties that are verified during/after test execution). These properties are important because (1) they serve as additional test oracles for validating candidate apps; and (2) they could compensate for the lack of passing UI tests.

Table 3 shows different properties enforced in Droix. Bug hazard is a circumstance that increases likelihood of a bug being present in a program [13]. A recent study of Android apps reveals several exception handling bug hazards and Java exception handling best practices [18]. Given an exception \( E \) that leads to a crash, our code checker categorizes \( E \) as either a checked exception, an unchecked exception, or an error to determine if we could insert a handler (try-catch block) for \( E \). According the Java exception handling best practice “Error represents an unrecoverable condition which should not be handled”, hence, our code checker considers inserting handler for runtime errors a hard constraint and eliminates such patches. In contrast, inserting handlers for unchecked and checked exceptions are encoded as soft constraints that could affect the score of a mutant. Meanwhile, we encode the well-formedness property and the exception type property as hard constraints that should be satisfied.

6.4 Mutant Generation and Evaluation

Droix supports eight operators derived from our study of crashes in Android apps (Section 4). Table 2 shows the details of each operator.

---

**Table 3: Code-level and Test-level Properties Enforced in Droix**

<table>
<thead>
<tr>
<th>Level</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code-level</td>
<td>Well-formedness</td>
<td>Verify that a mutated APK is compilable and the structural type of the program matches the requires context of the selected operator.</td>
</tr>
<tr>
<td></td>
<td>Bug hazard</td>
<td>Checks whether a transformation violates Java exception-handling best practices.</td>
</tr>
<tr>
<td></td>
<td>Exception Type</td>
<td>Checks whether a transformation matches a given exception type. (e.g., Insert Null-check should be used for fixing NullPointerException exclusively)</td>
</tr>
<tr>
<td></td>
<td>Lifecycle</td>
<td>Checks that the event transition matches with the activity and fragment lifecycle model (Figure 1).</td>
</tr>
<tr>
<td>Test-level</td>
<td>Activity-Fragment</td>
<td>Checks that the interaction between a fragment and its parent activity matches the activity-fragment coordination model (dashed lines in Figure 1)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
<td>Checks that a commit of a fragment’s transactions is performed in the allowed states (i.e., after an activity’s state is saved).</td>
</tr>
</tbody>
</table>

**Algorithm 1: Patch generation algorithm**

```
Input: Buggy APK, Operators Op, Population size PopSize, UI test U, Program Locations Locs
Result: APK that passes U and contains least property violations Pop ← initialPopulation(APK, PopSize);
while \( 3C \in Pop \) passes U do
    Mutants ← Mutate(Pop, Op, Locs); // apply Op at \( l \in Locs \\
    /\ select mutant with least \( Rc \) and \( Rr \) violations \( *\)
    Pop ← Select(Mutants, PopSize, Fitness).
end
```
Algorithm 1 presents our patch generation algorithm. Droix leverages \((\mu + \lambda)\) evolutionary algorithm with \(\mu = 40\) and \(\lambda = 20\). Given as input population size \(\text{PopSize}\), fitness function \(\text{Fit}\), and a list of faulty locations \(\text{Locs}\), our approach iteratively generates new mutants by applying one of the operators listed in Table 2 at each location in \(\text{Locs}\), evaluates each mutant by executing the input UI event sequences \(U\), and computes the number of code-level property \(R_c\) and test-level property \(R_t\) violations. The generate-and-validate process terminates when either there exists at least one mutant in the population that passes \(U\) or the time limit is exceeded. Our patch generation algorithm differs from existing approaches that use evolutionary algorithm [24, 53] in which we use a different patch representation and fitness function. Specifically, each mutant is an APK in our representation. Instead of using the number of passing tests as the fitness function, our fitness function \(\text{Fit}\) computes the number of code-level and test-level property violations.

7 IMPLEMENTATION

Our Android repair framework leverages various open source tools to support different components. Specifically, our log analyzer uses Logcat 11, a command-line tool that generates logs when events occur on an Android device. We implement the eight operators in Table 2 on top of the Soot framework (v2.5.0) [25]. Soot is a Java optimization framework that supports analysis and transformation of Java bytecode. Dexpler, a module included in Soot leverages an Dalvik bytecode disassembler to produce Jimple (a Soot representation) which enables reading and writing Dalvik bytecode directly [11]. We use the Dexpler module in Soot for our decompiler component in Figure 3. To support the “S4: Replace method” operator, we use the Levenshtein distance to select a method with a similar method name and compatible parameter types. Our implementation for the “S3: Replace resource id” operator uses Android resource parser in FlowDroid [7] to obtain a resource id of the same type. As each compiled APK needs to be signed before installation, we use jarsigner 12 for signing the compiled APK. We re-install the signed APK onto the device using adb commands. 13 Instead of uninstalling and re-installing each signed app, app re-installation allows us to keep the app data (e.g. account information and settings) to save time in re-entering the required information during subsequent execution of \(U\).

8 SUBJECTS

While there are various benchmarks used in evaluating the effectiveness of automated testing of Android applications [4, 5, 15, 30] and the effectiveness of repair approaches for C programs [27, 50, 57], a recent study [16] showed that the crashes in these benchmarks cannot be adequately reproduced by existing Android testing tools. Meanwhile, Android-specific benchmark like DROIDBENCH [7] does not contain real Android apps and it is designed for evaluating taint-analysis tools. Although empirical studies on Android apps [12, 18] investigated the bug reports of real Android apps, none of these studies try to replicate the reported crashes. Therefore, all existing benchmarks cannot be used for evaluating the effectiveness of analyzing crashes in Android apps.

We introduce a new benchmark, called DroixBench that contains 24 reproducible crashes in 15 real-world Android apps. Apart from evaluating Droix, this benchmark could be used to assess the effectiveness of detecting and analyzing crashes in Android apps. To facilitate future research on analysis of crashes, we made DroixBench publicly available at: https://droix2017.github.io/.

DroixBench is a new set of Android apps for evaluating Droix. Apps used for deriving transformation operators in Section 4 are excluded from DroixBench to avoid the overfitting problem in the evaluation. Specifically, we modified our crawler to find the most recent issues (bug reports) on Android apps crashes on GitHub. Our goal is to identify a set of reproducible crashes in Android apps. To reduce the time in manual inspection of these bug reports, our crawler excludes (1) issues without any bug-fixing commits (which is essential for comparing patch quality); (2) unresolved issues (to avoid invalid failures); and (3) non-Android related issues (e.g., iOS crashes). This step yields more than 300 GitHub issues. We further exclude defects that do not fulfill the criteria below:

- **Device-specific defects.** We eliminate defects that require specific versions/brands of Android devices.
- **Resource-dependent defects.** We eliminate defects that require specific resources (e.g., making phone calls) as we may not be able to replicate these issues easily on an Android emulator.
- **Irreproducible crashes.** We eliminate crashes that are deemed irreproducible by the developers.

9 EVALUATION

We perform evaluation on the effectiveness of Droix in repairing crashes on real Android apps and we compare the quality of Droix’s patch with the quality of the human patch. Our evaluation aims to address the following research questions:

- **RQ1** How many crashes in Android apps can Droix fix?
- **RQ2** How is the quality of the patches generated by Droix compared with the patches generated by developers?

9.1 Experimental Setup

We evaluate Droix on 24 defects from 15 real Android apps in DroixBench. Table 4 lists information about the evaluated apps. The “Type” column contains information about the specific type of exception that causes the crash, whereas the “TestEx” column represents the time taken in seconds to execute the UI test. Overall, DroixBench contains a wide variety of apps of various sizes (4-115K lines of code) and different types of exceptions that lead to crashes.

As Droix relies on randomized algorithm, we use the same parameters (10 runs for each defect with \(\text{PopSize}=40\) and a maximum of 10 generations) as in GenProg [26] for our experiments. In each run, we report the first found among the lowest score (minimum property violations) patches. Each run of Droix is terminated after one hour or when a patch with minimal violations is generated. All experiments were performed on a machine with a quad-core Intel Core i7-5600U 2.60GHz processor and 12GB of memory. All apps are executed on a Google Nexus 5x emulator (Android API25).
Table 4: Subject Apps and Their Basic Statistics

<table>
<thead>
<tr>
<th>App Name</th>
<th>Description</th>
<th>Version</th>
<th>LOC</th>
<th>Type</th>
<th>TestEx(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistor</td>
<td>radio players</td>
<td>4K</td>
<td></td>
<td>NullPointerException</td>
<td>42.1</td>
</tr>
<tr>
<td>Pix-art</td>
<td>photo editor</td>
<td>11.5</td>
<td>4K</td>
<td>IllegalState</td>
<td>40.1</td>
</tr>
<tr>
<td>PoetAssistant</td>
<td>poet writing helper</td>
<td>11.04</td>
<td>6K</td>
<td>NullPointer</td>
<td>42.0</td>
</tr>
<tr>
<td>Anynemo</td>
<td>flashcard learning</td>
<td>1.10.1</td>
<td>29K</td>
<td>NullPointer</td>
<td>37.2</td>
</tr>
<tr>
<td>AnkiDroid</td>
<td>flashcard learning</td>
<td>10.9.922</td>
<td>33K</td>
<td>NullPointer</td>
<td>37.2</td>
</tr>
<tr>
<td>Fdroid</td>
<td>opensource app</td>
<td>0.103.2</td>
<td>50K</td>
<td>IllegalState</td>
<td>38.7</td>
</tr>
<tr>
<td>Yalp</td>
<td>app repository</td>
<td>0.17</td>
<td>11K</td>
<td>NullPointer</td>
<td>57.4</td>
</tr>
<tr>
<td>LabCoat</td>
<td>GitLab client</td>
<td>2.2.4</td>
<td>45K</td>
<td>NullPointer</td>
<td>49.2</td>
</tr>
<tr>
<td>GnuCash</td>
<td>finance expense tracker</td>
<td>2.1.4</td>
<td>42K</td>
<td>IllegalArgument</td>
<td>32.0</td>
</tr>
<tr>
<td>NoiseCapture</td>
<td>noise evaluator</td>
<td>0.42b</td>
<td>10K</td>
<td>NullPointer</td>
<td>42.5</td>
</tr>
<tr>
<td>ConnectBot</td>
<td>secure shell client</td>
<td>1.9.2</td>
<td>26K</td>
<td>OutOfBounds</td>
<td>57.4</td>
</tr>
<tr>
<td>K9</td>
<td>email client</td>
<td>5.111</td>
<td>115K</td>
<td>NullPointer</td>
<td>42.2</td>
</tr>
<tr>
<td>OpenMF</td>
<td>Mifosx client</td>
<td>1.0.1</td>
<td>75K</td>
<td>IllegalState</td>
<td>134.0</td>
</tr>
<tr>
<td>Transdroid</td>
<td>torrents client</td>
<td>2.5.0b3</td>
<td>37K</td>
<td>NullPointer</td>
<td>45.9</td>
</tr>
<tr>
<td>Beem</td>
<td>communication tool</td>
<td>0.1.7rc1</td>
<td>21K</td>
<td>NullPointer</td>
<td>61.3</td>
</tr>
</tbody>
</table>

For each defect, we manually inspect the source code of human patched program and the source code decompiled from Droix’s patched program. If the source code of automatically patched program differs from the human patched program, we further investigate the UI behavior of patched programs by installing both the human generated APK and the automatically generated APK onto the Android device. For each APK, we manually perform visual comparison of the screens triggered by a set of available UI actions (clicks, swipes) after the crashing point.

Definition 1. Given the source code of human patched program $S_{\text{human}}$, the code of an automatically generated patch $S_{\text{machine}}$, the compiled APK of human patched program $APK_{\text{human}}$, the compiled APK of automatically generated patch $APK_{\text{machine}}$, we measure patch quality using the criteria defined below:

(C1) Syntactically Equivalent. $S_{\text{machine}}$ is “Syntactically Equivalent” if $S_{\text{machine}}$ and $S_{\text{human}}$ are syntactically the same.

(C2) Semantically Equivalent. $S_{\text{machine}}$ is “Semantically Equivalent” if $S_{\text{machine}}$ and $S_{\text{human}}$ are not syntactically the same but produce the same semantic behavior.

(C3) UI-behavior Equivalent. $APK_{\text{machine}}$ is “UI-behavior Equivalent” to $APK_{\text{human}}$, if the UI-state at the crashing point after applying the automated fix is same as the UI-state at the crashing point after applying the human patch. Two UI-state are considered to be same if their UI layouts are same, the set of events enabled are same, and these events again (recursively) lead to UI-equivalent states. UI-behavior equivalence of $APK_{\text{human}}$ against $APK_{\text{machine}}$ is checked manually in our experiments.

(C4) Incorrect. We label an $APK_{\text{machine}}$ as “Incorrect” if $APK_{\text{machine}}$ leads to undesirable behavior (e.g., causes another crash) but this behavior is not observed in $APK_{\text{human}}$.

(C5) Better. We label an $APK_{\text{machine}}$ as “Better” when $APK_{\text{human}}$ leads to regression witnessed by another UI test $U_R$ whereas $APK_{\text{machine}}$ passes $U_R$.

Formally, $C_1 \Rightarrow C_2 \land C_2 \Rightarrow C_3$, hence, a generated patch that is syntactically equivalent to the human patch is superior to both semantically equivalent patch and UI-behavior equivalent patch. We note that, in general, checking whether a patch is semantically equivalent to the human patch (C2) is an undecidable problem. However, in our manual analysis, the correct behavior for all evaluated patches is well-defined. While C1 and C2 investigate the behavior of patches at the source-code level, we introduce C3 to compare the behavior of patches at the GUI-level. We consider C3 because our approach uses GUI tests for guiding the repair process. Furthermore, since our approach does not require source code, direct manual checking of source code may be sometimes tedious.

9.2 Evaluation Results
Table 5 shows the patch quality results for Droix. The “Time” column in Table 5 indicates the time taken in seconds across 10 runs for generating the patch before the one-hour time limit is reached. On average, Droix takes 30 minutes to generate a patch. Meanwhile, the “Repair” column denotes the number of plausible patches (APKs that pass the UI test) generated by Droix. Overall, Droix generates 15 plausible patches (rows marked with √) out of 24 evaluated defects. Our analysis of the 9 defects that are not repaired by Droix reveals that all of these defects are difficult to fix because all the corresponding human patches require at least 10 lines of edits.

The “Fix type” column in Table 5 shows the operator used in each patch (Refer to Table 2 for the description of each operator). The “Null-check” operator is the most frequently used operators (used in six patches). The “Merge” operator tends to produce high quality patches because this operator aims to enforce the “Activity-Fragment” property that checks for the coordination between the host activity and its embedded fragment.

The “Syntactic Equiv.” column in Table 5 shows the patches that fulfill C1, while the “Semantic Equiv.” column denotes patches that fulfill C2. Similarly, the “UI-behavior Equiv.” column demonstrates the number of fixed APKs that fulfill the C3 criteria. Particularly, we consider the patch generated by Droix for Anynemo v10.9.922 as “Semantically Equivalent” because both patches use an object of the same type retained before configuration changes to fix a NullPointerException exception but the object is retained in different program locations (i.e., not syntactically equivalent). Meanwhile, Droix generates three APKs that are UI-behavior equivalent to the human generated APKs. Interestingly, we observed that although the human patches for these defects require multi-lines fixes, the bug reports for these UI-behavior equivalent patches indicate that specific conditions are required to trigger the crashes (e.g., mSpinner.getSelectedItemId()!=INVALID_ROW_ID for the GnuCash v2.0.5 defect). As these conditions are difficult to trigger
Table 5: Patch Quality Results

<table>
<thead>
<tr>
<th>App</th>
<th>Version</th>
<th>Time (s)</th>
<th>Fix type</th>
<th>Repair</th>
<th>Syntactic Equiv.</th>
<th>Semantic Equiv.</th>
<th>UI-behavior Equiv.</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistor</td>
<td>1.2.3</td>
<td>616</td>
<td>-</td>
<td>GetActivity-check</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.1.5</td>
<td>987</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FixArt</td>
<td>1.17.1</td>
<td>1164</td>
<td>-</td>
<td>Null-check</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.17.0</td>
<td>1525</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PortAssistant</td>
<td>1.18.2</td>
<td>955</td>
<td>Null-check</td>
<td>√</td>
<td>△</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.10.4</td>
<td>3600</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Anymemo</td>
<td>10.10.1</td>
<td>2084</td>
<td></td>
<td>Retain Object</td>
<td>√</td>
<td></td>
<td></td>
<td>□</td>
</tr>
<tr>
<td></td>
<td>10.9.22</td>
<td>1336</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AnkiDroid</td>
<td>2.8.1</td>
<td>3609</td>
<td></td>
<td>Try-catch</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.7b1</td>
<td>3609</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fdroid</td>
<td>0.103.2</td>
<td>2993</td>
<td>Replace method</td>
<td>√</td>
<td>□</td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>0.98</td>
<td>518</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yalp</td>
<td>0.17</td>
<td>2970</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LabCoat</td>
<td>2.2.4</td>
<td>2074</td>
<td>Null-check</td>
<td>√</td>
<td>△</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GnuCash</td>
<td>2.1.3</td>
<td>360</td>
<td></td>
<td>Try-catch</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.0.5</td>
<td>1492</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.1.4</td>
<td>3609</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConnectBot</td>
<td>1.9.2</td>
<td>572</td>
<td>Try-catch</td>
<td>√</td>
<td>△</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NoiseCapture</td>
<td>0.4.2b</td>
<td>340</td>
<td>Null-check</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>0.4.2b</td>
<td>520</td>
<td>Replace cast</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>K9</td>
<td>5.111</td>
<td>1718</td>
<td>Try-catch</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>crash(*)</td>
</tr>
<tr>
<td>OpenMF</td>
<td>1.0.1</td>
<td>3609</td>
<td>GetActivity-check</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Bem</td>
<td>0.1.7e1</td>
<td>2578</td>
<td>Null-check</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Transdroid</td>
<td>2.5.0b1</td>
<td>1315</td>
<td>Null-check</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

from the UI level, synthesizing precise conditions is not required for ensuring UI-behavior equivalent.

The "Others" column in Table 5 includes one patch that is better than the human patch (marked as ⊕) and three patches that are incorrect (marked as ×). We consider the patch for Transistor v1.1.5 to be better than human patch as it passes regression test stated in the bug report whereas the human patch introduces a new regression (See Section 3 for detailed explanations). For two of the incorrect patches, we notice that some texts that appear on the screen of human APKs are missing in the screen of fixed APKs (text missing). Meanwhile, the crash in k9 v5.111 occurs due to an invalid email address for a particular contact. In this case, the human APK treats the contact as a non-existing contact while the patched APK displays the contact as unknown recipient and crashes when the unknown recipient is selected. We think that both the human APK and the patched APK could be improved (e.g., prompt the user to enter a valid email address instead of ignoring the contact). Although the patch generated by Droix for k9 violates the bug hazard property (catching a runtime exception), we select this patch as no other patches are found within the time limit.

Droix fixes 15 out of 24 evaluated crashes, seven of these fixes are the same as the patched patches, one repair is semantically equivalent, three are UI-behavior equivalent. In one rare case, we generate better repair.

10 THREATS TO VALIDITY

We identify the following threats to the validity of our experiments:

Operators used. While we derive our operators from frequently used operators in fixing open source apps and from Android API documentation, our set of operators is not exhaustive.

Reproducing crashes. We manually reproduce each crash in our proposed benchmark. As we rely on Android emulator for reproducing crashes, the crashes in our benchmark are limited to crashes that could be reliably reproduced on Android emulators. Crashes that require specific setup (e.g., making phone calls) may be more challenging or impractical to replay.

Crashes investigated. As we only investigate open source Android apps in our empirical study and in our proposed benchmark, our results may not generalize to closed-source apps. We focus on open source apps because our patch analysis requires the availability of source codes. Nevertheless, as Droix takes as input Android APK, it could be used for fixing closed source apps. We leave the empirical evaluation of closed source apps as our future work.

Patch Quality. During our manual patch analysis, at least two of the authors analyze the quality of human patches versus the quality of automatically generated patches separately and meet to resolve any disagreement. As most bug reports include detailed explanations of human patches and the expected behavior of the crashing UI test, the patch analysis is relatively straightforward.

11 RELATED WORK

Testing and Analysis of Android Apps. Many automated techniques (AndroidRipper [4], ACTEVE [5], A³E [9], Collider [23], Dynodroid [30], FSmfordroid [47], Fuzzdroid [43], Orbit [56], Sapienz [31], Swifthand [15], and work by Mirzaei et al. [37]) are proposed to generate test inputs for Android apps. Our approach is orthogonal to these approaches and the tests generated by these approaches could
Automated Program Repair. Several techniques (Angelix [35], ASTOR [33], ClearView [41], Directfix [34], GenProg [26], PAR [24], Prophet [28], NOPOL [54], relifix [49]) have been introduced to automatically generate patches. There are several key differences of our Android repair framework compared to other existing repair approaches. Firstly, instead of relying on the quality of the test suite for guiding the repair process, our approach augments a given UI test with code-level and test-level properties for rank-generated patches. Secondly, existing approaches could not handle flaky UI tests as they may misinterpret the test outcome. Our approach allows UI sequences in forms of scripts recorded in the user interface, the record-and-replay mechanism in RERAN could allow Droix to handle more complex UI events. Although our code checker incorporates some Java exception handling best practices listed in recent study of Android apps [18], our empirical study of crashes that occur in Android apps goes beyond prior study by performing a thorough investigation of the common root causes of Android crashes.

Automated Program Repair. Several techniques (Angelix [35], ASTOR [33], ClearView [41], Directfix [34], GenProg [26], PAR [24], Prophet [28], NOPOL [54], relifix [49]) have been introduced to automatically generate patches. There are several key differences of our Android repair framework compared to other existing repair approaches. Firstly, instead of relying on the quality of the test suite for guiding the repair process, our approach augments a given UI test with code-level and test-level properties for rank-generated patches. Secondly, existing approaches could not handle flaky UI tests as they may misinterpret the test outcome. Our approach allows UI sequences in forms of scripts recorded in the user interface, the record-and-replay mechanism in RERAN could allow Droix to handle more complex UI events. Although our code checker incorporates some Java exception handling best practices listed in recent study of Android apps [18], our empirical study of crashes that occur in Android apps goes beyond prior study by performing a thorough investigation of the common root causes of Android crashes.

Automated Program Repair. Several techniques (Angelix [35], ASTOR [33], ClearView [41], Directfix [34], GenProg [26], PAR [24], Prophet [28], NOPOL [54], relifix [49]) have been introduced to automatically generate patches. There are several key differences of our Android repair framework compared to other existing repair approaches. Firstly, instead of relying on the quality of the test suite for guiding the repair process, our approach augments a given UI test with code-level and test-level properties for rank-generated patches. Secondly, existing approaches could not handle flaky UI tests as they may misinterpret the test outcome. Our approach allows UI sequences in forms of scripts recorded in the user interface, the record-and-replay mechanism in RERAN could allow Droix to handle more complex UI events. Although our code checker incorporates some Java exception handling best practices listed in recent study of Android apps [18], our empirical study of crashes that occur in Android apps goes beyond prior study by performing a thorough investigation of the common root causes of Android crashes.

Automated Program Repair. Several techniques (Angelix [35], ASTOR [33], ClearView [41], Directfix [34], GenProg [26], PAR [24], Prophet [28], NOPOL [54], relifix [49]) have been introduced to automatically generate patches. There are several key differences of our Android repair framework compared to other existing repair approaches. Firstly, instead of relying on the quality of the test suite for guiding the repair process, our approach augments a given UI test with code-level and test-level properties for rank-generated patches. Secondly, existing approaches could not handle flaky UI tests as they may misinterpret the test outcome. Our approach allows UI sequences in forms of scripts recorded in the user interface, the record-and-replay mechanism in RERAN could allow Droix to handle more complex UI events. Although our code checker incorporates some Java exception handling best practices listed in recent study of Android apps [18], our empirical study of crashes that occur in Android apps goes beyond prior study by performing a thorough investigation of the common root causes of Android crashes.

Automated Program Repair. Several techniques (Angelix [35], ASTOR [33], ClearView [41], Directfix [34], GenProg [26], PAR [24], Prophet [28], NOPOL [54], relifix [49]) have been introduced to automatically generate patches. There are several key differences of our Android repair framework compared to other existing repair approaches. Firstly, instead of relying on the quality of the test suite for guiding the repair process, our approach augments a given UI test with code-level and test-level properties for rank-generated patches. Secondly, existing approaches could not handle flaky UI tests as they may misinterpret the test outcome. Our approach allows UI sequences in forms of scripts recorded in the user interface, the record-and-replay mechanism in RERAN could allow Droix to handle more complex UI events. Although our code checker incorporates some Java exception handling best practices listed in recent study of Android apps [18], our empirical study of crashes that occur in Android apps goes beyond prior study by performing a thorough investigation of the common root causes of Android crashes.

Automated Program Repair. Several techniques (Angelix [35], ASTOR [33], ClearView [41], Directfix [34], GenProg [26], PAR [24], Prophet [28], NOPOL [54], relifix [49]) have been introduced to automatically generate patches. There are several key differences of our Android repair framework compared to other existing repair approaches. Firstly, instead of relying on the quality of the test suite for guiding the repair process, our approach augments a given UI test with code-level and test-level properties for rank-generated patches. Secondly, existing approaches could not handle flaky UI tests as they may misinterpret the test outcome. Our approach allows UI sequences in forms of scripts recorded in the user interface, the record-and-replay mechanism in RERAN could allow Droix to handle more complex UI events. Although our code checker incorporates some Java exception handling best practices listed in recent study of Android apps [18], our empirical study of crashes that occur in Android apps goes beyond prior study by performing a thorough investigation of the common root causes of Android crashes.

Automated Program Repair. Several techniques (Angelix [35], ASTOR [33], ClearView [41], Directfix [34], GenProg [26], PAR [24], Prophet [28], NOPOL [54], relifix [49]) have been introduced to automatically generate patches. There are several key differences of our Android repair framework compared to other existing repair approaches. Firstly, instead of relying on the quality of the test suite for guiding the repair process, our approach augments a given UI test with code-level and test-level properties for rank-generated patches. Secondly, existing approaches could not handle flaky UI tests as they may misinterpret the test outcome. Our approach allows UI sequences in forms of scripts recorded in the user interface, the record-and-replay mechanism in RERAN could allow Droix to handle more complex UI events. Although our code checker incorporates some Java exception handling best practices listed in recent study of Android apps [18], our empirical study of crashes that occur in Android apps goes beyond prior study by performing a thorough investigation of the common root causes of Android crashes.

Automated Program Repair. Several techniques (Angelix [35], ASTOR [33], ClearView [41], Directfix [34], GenProg [26], PAR [24], Prophet [28], NOPOL [54], relifix [49]) have been introduced to automatically generate patches. There are several key differences of our Android repair framework compared to other existing repair approaches. Firstly, instead of relying on the quality of the test suite for guiding the repair process, our approach augments a given UI test with code-level and test-level properties for rank-generated patches. Secondly, existing approaches could not handle flaky UI tests as they may misinterpret the test outcome. Our approach allows UI sequences in forms of scripts recorded in the user interface, the record-and-replay mechanism in RERAN could allow Droix to handle more complex UI events. Although our code checker incorporates some Java exception handling best practices listed in recent study of Android apps [18], our empirical study of crashes that occur in Android apps goes beyond prior study by performing a thorough investigation of the common root causes of Android crashes.

Automated Program Repair. Several techniques (Angelix [35], ASTOR [33], ClearView [41], Directfix [34], GenProg [26], PAR [24], Prophet [28], NOPOL [54], relifix [49]) have been introduced to automatically generate patches. There are several key differences of our Android repair framework compared to other existing repair approaches. Firstly, instead of relying on the quality of the test suite for guiding the repair process, our approach augments a given UI test with code-level and test-level properties for rank-generated patches. Secondly, existing approaches could not handle flaky UI tests as they may misinterpret the test outcome. Our approach allows UI sequences in forms of scripts recorded in the user interface, the record-and-replay mechanism in RERAN could allow Droix to handle more complex UI events. Although our code checker incorporates some Java exception handling best practices listed in recent study of Android apps [18], our empirical study of crashes that occur in Android apps goes beyond prior study by performing a thorough investigation of the common root causes of Android crashes.
REFERENCES


