Repairing Crashes in Android Apps

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

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

Xiang Gao
National University of Singapore
gaoxiang@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

ACM Reference Format:
Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing Crashes in Android Apps. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180243

[*]This work was done during the author's PhD study at National University of Singapore

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180243

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 that 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.

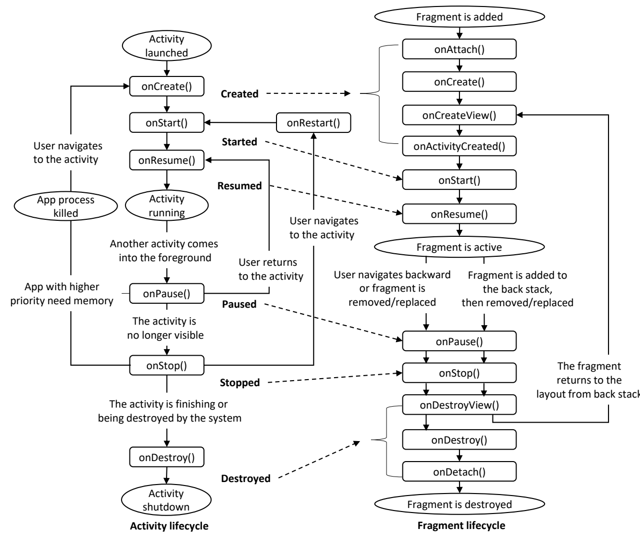


Figure 1: Activity Lifecycle, Fragment Lifecycle and the Activity-Fragment Coordination

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.

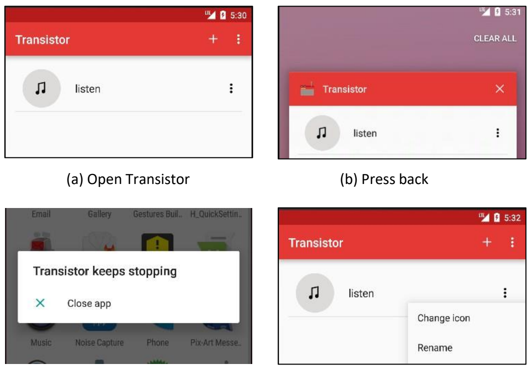
A *fragment* is a portion of user interface or a behavior that can be put in an Activity. Each fragment can be modified independently of the *host activity* (activity containing the fragment) by performing a set of changes. For a fragment, it goes through more states than an Activity from being launched 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 report[1], 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



(a) Open Transistor (b) Press back

(d) Crashed with a notification (C) Open again and change an icon

Figure 2: Continuous snapshots of a crash in Transistor.

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:
    Fragment MainActivityFragment{82e1bec} not attached to Activity
at android...startActivityForResult(Fragment.java:925)
at y20k...selectFromImagePicker(MainActivityFragment.java:482)
```

Listing 1: Stack trace for the crash in Transistor

```
      +if(getActivity()!=null)
482:  startActivityForResult(pickImageIntent, REQUEST_LOAD_IMAGE);
```

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

```
      -startActivityForResult(pickImageIntent, REQUEST_LOAD_IMAGE);
482:  +mActivity.startActivityForResult(pickImageIntent,
       REQUEST_LOAD_IMAGE);
```

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

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.

Listing 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 manual 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 [2]. 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 crashes-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 *Q*2 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 documentations [3].

Our study shows that the most common exceptions are:

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

[2]https://developer.github.com/v3/

[3]https://developer.android.com/guide/components/activities/activity-lifecycle.html

Table 1: Root cause of crashes in Android apps

Category	Specific reason	Description	GitHub Issues (%)	Frequent Exception Type	Category Total (%)
Lifecycle	Configuration changes	activity recreation during configuration changes	5.61	NullPointer	14.02
	Stateloss	transaction loss during commit	2.80	IllegalState	
	GetActivity	activity-fragment coordination	2.80	IllegalState	
	Activity backstack	inappropriate handling of activity stack	1.87	IllegalArgument	
	Save instance	uninitialized object instances in onSaveInstance() callback	0.93	IllegalState	
Resource	Resource-related	resource type mismatches	10.28	NullPointer	16.82
	Resource limit	limited resources	4.67	OutOfMemory	
	Incorrect resource	retrieve a wrong resource id	1.87	SQLite	
Callback	Activity-related	missing activities	7.48	NullPointer	17.76
	View-related	missing views	6.54	NullPointer	
	Intent-related	missing intents	3.74	NullPointer	
	Unhandled callbacks	missing callbacks	2.80	NullPointer	
Others	Missing Null-check	missing check for null object reference	12.15	NullPointer	52.34
	External Service/Library	defects in external service/library	8.41	NullPointer	
	Workaround	temporary fixes for defect	4.67	IndexOutOfBound	
	API changes	API version changes	2.80	SQLite	
	Others	project-specific defects	24.30	-	

The high percentage of `NullPointerException` 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 NullPointerExceptions 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 general 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

Table 2: Supported Operators in Droix

Operator	Description
S1: GetActivity-check	Insert a condition to check whether the activity containing the fragment has been created.
S2: Retain object	Store objects and load them when configuration changes
S3: Replace resource id	Replace resource id with another resource id of same type.
S4: Replace method	Replace the current method call with another method call with similar name and compatible parameter types.
S5: Replace cast	Replace the current type cast with another compatible type.
S6: Move stmt	Removes a statement and add it to another location.
S7: Null-check	Insert condition to check if a given object is null.
S8: Try-catch	Insert try-catch blocks for the given exception.

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()`→ `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 configuration changes altogether. 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).

```
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 inflater,
      ViewGroup container, Bundle savedInstanceState) {
// saving and loading the object
+  if(option!=null){  saveOption = option; }
+ else{ option = saveOption; }
  switch (option.getButtonStyle()) { //crashing point
```

Listing 4: Example of handling crashes during configuration changes

[4]https://developer.android.com/guide/topics/resources/runtime-changes.html

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 [5] specifies rules to prohibit committing transactions at certain stages of the lifecyle. 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 [6]. 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: (S1: 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) [7]. 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 API (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).

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

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

Incorrect type-cast of resource To implement UI interfaces, an Android API [8] (`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.

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

Listing 6: Example fix for incorrect resource type-cast

[5]https://developer.android.com/reference/android/app/FragmentTransaction.html
[6]https://developer.android.com/guide/components/fragments.html
[7]https://developer.android.com/guide/topics/resources/accessing-resources.html
[8]https://developer.android.com/reference/android/app/Activity.html

6 METHODOLOGY

Figure 3 presents the overall workflow of Droix's repair framework. Droix consists of several components: a test replayer, a log analyzer, a mutant generator, a test checker, a code checker, and a selector. Given a buggy APK_P and UI event sequences U extracted from its bug report, Droix produces a patched $APK_{P'}$ that passes U and has the minimum number of properties violations.

Droix fixes a crash using a two-phase approach. In the first phase, Droix generates an instrumented APK_I to log all executed callbacks. With the instrumented APK, Droix replays the UI event sequences U on a device. The log analyzer parses the logs dumped from the execution, extracts program locations *Locs* from the stack trace, and identifies test-level property Rt_{orig} using the recorded callbacks.

In the second phase, Droix decompiles APK_P to the intermediate representation. Then, our mutant generator produces a set of candidate apps (stored in the mutant pool) by applying a set of operators at each location l in *Locs*. For each operator *op*, our code checker records code-level property Rc_{cand} based on the program structure of l and the information in thrown exception. For each candidate APK_C, Droix reinstalls APK_C onto the device and replays U on APK_C. Then, our log analyzer parses the dumped logs that include the execution information of callback methods to extract new buggy locations and information of test-level property Rt_{cand}. Given as input Rt_{cand} for APK_C, the test checker compares Rt_{orig} with Rt_{cand} to check if APK_C introduces any new property violations. Finally, our evaluator analyzes Rt_{cand} and Rc_{cand} to compute the number of property violations and passes the results to the selector, which chooses the best app as the final fixed APK.

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` [9] GUI, (2) a set of Android Debug Bridge (adb) commands [10], 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

[9]Monkeyrunner contains API that allows controlling Android devices: https://developer.android.com/studio/test/monkeyrunner/index.html
[10]ADB is a command-line tool that are used to control Android devices: https://developer.android.com/studio/command-line/adb.html

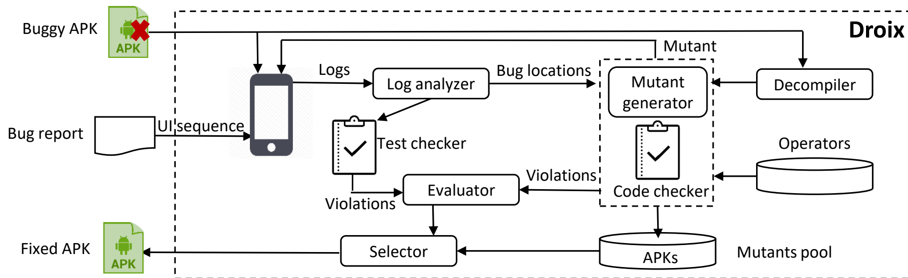


Figure 3: Droix's Android Repair Framework

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

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

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.

Given a previous lifecycle callback *prev* and a current lifecycle callback *curr*, our test checker verifies if *prev* $\rightarrow$ *curr* obeys the activity/fragment lifecycle management rules (Figure 1). Droix considers all test-level properties as soft constraints because these properties may not be directly related to the crash (e.g., resource-related crashes).

Algorithm 1: Patch generation algorithm

Input: Buggy APK_P, Operators Op, Population size $PopSize$, UI test U, Program Locations $Locs$
Input: Fitness Fit: $< Patch, Rc, Rt > \rightarrow \mathbb{Z}$
Result: APK that passes U and contains least property violations
$Pop \leftarrow initialPopulation(APK_P, PopSize)$;
while $\nexists C \in Pop.C\ passes\ U$ **do**
  $Mutants \leftarrow Mutate(Pop, Op, Locs)$; // apply Op at $l \in Locs$
  /* select mutant with least Rc and Rt violations */
  $Pop \leftarrow Select(Mutants, PopSize, Fit)$;
end

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.

Algorithm 1 presents our patch generation algorithm. Droix leverages (μ+λ) evolutionary algorithm with $\mu = 40$ and $\lambda = 20$. Given as input population size *PopSize*, fitness function *Fit*, and a list of faulty locations *Locs*, our approach iteratively generates new mutants by applying one of the operators listed in Table 2 at each location in *Locs*, evaluates each mutant by executing the input UI event sequences U, and computes the number of code-level property *Rc* and test-level property *Rt* 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 *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 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 *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).

[11] https://developer.android.com/studio/command-line/logcat.html
[12] http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html
[13] https://developer.android.com/studio/command-line/adb.html

Table 4: Subject Apps and Their Basic Statistics

App Name	Description	Version	LOC	Type	TestEx(s)
Transistor	radio players	1.2.3	4K	NullPointer	42.1
		1.1.5	4K	IllegalState	40.1
Pix-art	photo editor	1.17.1	54K	NullPointer	37.2
		1.17.0	60K	NullPointer	42.0
PoetAssistant	poet writing helper	1.18.2	12K	NullPointer	42.3
		1.10.4	6K	SQLite	60.9
Anymemo	flashcard learning	10.10.1	29K	NullPointer	50.5
		10.9.922	33K	NullPointer	83.9
AnkiDroid	flashcard learning	2.8.1	73K	IllegalState	50.6
		2.7b1	73K	ClassCast	37.2
Fdroid	opensoure app repository	0.103.2	50K	IllegalState	38.7
		0.98	38K	SQLite	37.3
Yalp	app repository	0.17	11K	NullPointer	57.4
LabCoat	GitLab client	2.2.4	45K	NullPointer	49.2
GnuCash	finance expense tracker	2.1.4	42K	IllegalArgument	32.0
		2.1.3	40K	NullPointer	37.2
		2.0.5	37K	IllegalArgument	42.2
NoiseCapture	noise evaluator	0.4.2b	10K	NullPointer	42.5
		0.4.2b	10K	ClassCast	41.2
ConnectBot	secure shell client	1.9.2	26K	OutOfBounds	57.4
K9	email client	5.111	115K	NullPointer	42.2
OpenMF	Mifosx client	1.0.1	75K	IllegalState	134.0
Transdroid	torrents client	2.5.0b1	37K	NullPointer	45.9
Beem	communication tool	0.1.7rc1	21K	NullPointer	61.3

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 Src_{human}, the code of an automatically generated patch $Src_{machine}$, the compiled APK of human patched program APK_{human}, the compiled APK of automatically generated patch $APK_{machine}$, we measure patch quality using the criteria defined below:

(C1) Syntactically Equivalent. $Src_{machine}$ is "Syntactically Equivalent" if $Src_{machine}$ and Src_{human} are syntactically the same.

(C2) Semantically Equivalent. $Src_{machine}$ is "Semantically Equivalent" if $Src_{machine}$ and Src_{human} are not syntactically the same but produce the same semantic behavior.

(C3) UI-behavior Equivalent. $APK_{machine}$ is "UI-behavior Equivalent" to APK_{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_{human} against $APK_{machine}$ is checked manually in our experiments.

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

(C5) Better. We label a $APK_{machine}$ as "Better" when APK_{human} leads to regression witnessed by another UI test U_R whereas $APK_{machine}$ passes U_R.

Formally, $C1 \implies C2 \land C2 \implies C3$, 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 are 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 and 4/6=67% of these patches are syntactically equivalent to the human patches). These results match with the high frequency of "Null-check" operator in our empirical study (Table 1). Interestingly, we also observe that the "GetActivity-check" 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 Anymemo v10.9.922 as "Semantically Equivalent" because both patches use an object of the same type retained before configuration changes to fix a `NullPointerException` 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

App	Version	Time (s)	Fix type	Repair	Syntactic Equiv.	Semantic Equiv.	UI-behavior Equiv.	Others
Transistor	1.2.3	616	-					
	1.1.5	987	GetActivity-check	√				better(⊕)
PixArt	1.17.1	1164	-					
	1.17.0	1525	Null-check	√			△	
PoetAssistant	1.18.2	955	Null-check	√			△	
	1.10.4	3600	-					
Anymemo	10.10.1	2104	-					
	10.9.922	1336	Retain Object	√		⊙		
AnkiDroid	2.8.1	3600	-					
	2.7b1	3600	Try-catch	√				text missing(×)
Fdroid	0.103.2	2293	Replace method	√	★			
	0.98	518	-					
Yalp	0.17	2970	-					
LabCoat	2.2.4	2074	Null-check	√	★			
GnuCash	2.1.3	360	-					
	2.0.5	1492	Try-catch	√			△	
	2.1.4	3600	-					
ConnectBot	1.9.2	572	Try-catch	√				text missing(×)
NoiseCapture	0.4.2b	340	Null-check	√	★			
	0.4.2b	520	Replace cast	√	★			
K9	5.111	1718	Try-catch	√				crash(×)
OpenMF	1.0.1	3600	GetActivity-check	√	★			
Beem	0.1.7rc1	2378	Null-check	√	★			
Transdroid	2.5.0b1	1315	Null-check	√	★			
	24			15	7	1	3	4

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 human 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], A^3E [9], Collider [23], Dynodroid [30], FSMdroid [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 serve as inputs to our Android repair system. Several approaches focus on reproducing crashes in Java projects [14, 46, 55]. Meanwhile, CRASHSCOPE [38] automatically detects and reproduces crashes in Android apps. Our benchmark with 24 reproducible crashes could be used for evaluating the effectiveness of these approaches. Similar to Flowdroid [7], we implement our fix operators on top of the Soot framework, and we use activity lifecycle information for our analysis of Android apps. Instead of considering only the activity lifecycle as in Flowdroid, we also encode fragment lifecycle and activity-fragment coordination as test-level properties. RERAN [22] could precisely record and replay UI events on Android devices, including gestures (e.g., multitouch). While 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 ranking generated patches. Secondly, existing approaches could not handle flaky UI tests as they may misinterpret the test outcome of UI tests and may mistakenly produce invalid patches. Finally, our repair framework modifies compiled APK and each test execution is performed remotely on Android emulators, whereas other approaches modify source code directly where each test is being executed on the same platform as other components of the repair system. Other studies for automated repair use benchmark for C programs [27, 50, 57, 58], whereas Droixbench contains a set of reproducible crashes for Android apps. QACrashFix [19] and work by Azim et al. [8] use Android apps as dataset for experiments, without any Android-specific study of cause for crashes. Their repair operators are Android-agnostic. Specifically, QACrashFix merely add/delete/replace single node in the Abstract Syntax Tree, wheareas work by Azim et al only inserts fault-avoiding code that is similar to workaround identified in our study in Section 4. To eliminate invalid patches, anti-patterns are proposed as a set of forbidden rules that can be enforced on top of search-based repair approaches [51]. Although our code-level and test-level properties could be considered as different forms of anti-patterns that are examined prior to and after test executions, we use these properties for selecting mutants that violate fewer properties instead of eliminating these mutants. Similar to Droix that uses stack trace information for fault localization, the work of Sinha et al. uses stack trace information for locating Java exceptions [44]. However, their approach only supports analysis of `NullPointerException`, whereas our approach could automatically repair different types of exceptions.

Other Repairs of Android Apps EnergyPatch fixes energy bugs in Android apps using a repair expression that captures the resource expression and releases system calls [10]. The battery-aware transformations proposed in [17] aims to reduce power consumption of mobile devices. Several approaches generate security patches for Android apps [39, 59]. While energy bugs and security-related vulnerabilities may cause crashes in Android apps, we present a generic framework for automated repair of Android crashes, focusing on crashes that occur due to the misunderstanding of Android activity and fragment lifecycles.

UI Repair. FlowFixer is an approach that repairs broken workflow in GUI applications that evolve due to GUI refactoring. SITAR uses annotated event-flow graph for fixing unusable GUI test scripts [20]. Although Droix takes as input UI test, it automatically fixes buggy Android apps rather than the inputs that crash the GUI applications.

12 CONCLUSIONS AND FUTURE WORK

We study the common causes of 107 crashes in Android apps. Our investigation reveals that app crashes occur due to missing callback handler (17.76%), improper handling of resources (16%), and violations of management rules for the Android activity and fragment lifecycles (14%). Based on our analysis of patches issued by Android developers to fix these crashes and the Android API documentations that specify the correct usage of Android API, we derive a set of lifecycle-aware transformations. To reduce time and effort in fixing crashes in Android apps, we also introduce Droix, a novel Android repair framework that automatically generates a fixed APK when given as input a buggy APK and UI event sequences. To encourage future research of Android crashes, we propose DroixBench, a benchmark that contains 24 reproducible crashes occurring in 15 open source Android apps. Our evaluation on DroixBench demonstrates that Droix could generate repair for 63% of the evaluated crashes and seven of the automatically generated patches are syntactically equivalent to the human patches.

Although our repair framework currently performs analysis and mutation of Android apps on desktop machine while executing UI tests on an Android emulator, in the future, it is feasible to have a standalone repair system that could be installed as an app that automatically fixes crashes occurring in other apps on Android devices. Since our GUI interface does not assume any programming knowledge, our repair framework could potentially benefit general non-technical users who would like to have their own versions of fixed apps instead of waiting for the official releases. Moreover, as we observe that many crashes occur due to the misunderstanding of activity/fragment lifecycle that are specified in the Android API documentations, we think that Droix could be used as a plugin that automatically provides management rule violations together with patch suggestions to assist developers in understanding Android API specifications.

ACKNOWLEDGEMENT

This research is supported by the National Research Foundation, Prime Minister's Office, Singapore under its Corporate Laboratory at University Scheme, National University of Singapore, and Singapore Telecommunications Ltd. The first author thanks Southern University of Science and Technology for the travel support.

REFERENCES

[1] 2017. What Consumers Really Need and Want. https://goo.gl/puYdkG. (2017). Accessed 2017-03-27.

[2] Sharad Agarwal, Ratul Mahajan, Alice Zheng, and Victor Bahl. 2010. Diagnosing mobile applications in the wild. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 22.

[3] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. 2011. A gui crawling-based technique for android mobile application testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. IEEE, 252–261.

[4] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. 2012. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 258–261.

[5] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 59.

[6] Alessandro Armando, Alessio Merlo, Mauro Migliardi, and Luca Verderame. 2013. Breaking and fixing the android launching flow. *Computers & Security* 39 (2013), 104–115.

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[8] Md Tanzirul Azim, Iulian Neamtiu, and Lisa M Marvel. 2014. Towards self-healing smartphone software via automated patching. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 623–628.

[9] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Acm Sigplan Notices*, Vol. 48. ACM, 641–660.

[10] A. Banerjee, L. K. Chong, C. Ballabriga, and A. Roychoudhury. 2017. EnergyPatch: Repairing Resource Leaks to Improve Energy-efficiency of Android Apps. *IEEE Transactions on Software Engineering* PP, 99 (2017), 1–1. https://doi.org/10.1109/TSE.2017.2689012

[11] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. 2012. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis (SOAP '12)*. ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/2259051.2259056

[12] Pamela Bhattacharya, Liudmila Ulanova, Iulian Neamtiu, and Sai Charan Koduru. 2013. An empirical analysis of bug reports and bug fixing in open source android apps. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, 133–143.

[13] Robert V Binder. 2000. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional.

[14] N. Chen and S. Kim. 2015. STAR: Stack Trace Based Automatic Crash Reproduction via Symbolic Execution. *IEEE Transactions on Software Engineering* 41, 2 (Feb 2015), 198–220. https://doi.org/10.1109/TSE.2014.2363469

[15] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided gui testing of android apps with minimal restart and approximate learning. In *Acm Sigplan Notices*, Vol. 48. ACM, 623–640.

[16] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?(e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 429–440.

[17] Jürgen Cito, Julia Rubin, Phillip Stanley-Marbell, and Martin Rinard. 2016. Battery-aware transformations in mobile applications. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 702–707.

[18] Roberta Coelho, Lucas Almeida, Georgios Gousios, Arie Van Deursen, and Christoph Treude. 2016. Exception handling bug hazards in Android. *Empirical Software Engineering* (2016), 1–41.

[19] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. 2015. Fixing recurring crash bugs via analyzing q&a sites (T). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 307–318.

[20] Zebao Gao, Zhenyu Chen, Yunxiao Zou, and Atif M Memon. 2016. Sitar: GUI test script repair. *Ieee transactions on software engineering* 42, 2 (2016), 170–186.

[21] Laurence Goasduff and Christy Pettey. 2012. Gartner says worldwide smartphone sales soared in fourth quarter of 2011 with 47 percent growth. *Visited April* (2012).

[22] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. RERAN: Timing- and Touch-sensitive Record and Replay for Android. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 72–81. http://dl.acm.org/citation.cfm?id=2486788.2486799

[23] Casper S. Jensen, Mukul R. Prasad, and Anders Møller. 2013. Automated Testing with Targeted Event Sequence Generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, New York, NY, USA, 67–77. https://doi.org/10.1145/2483760.2483777

[24] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *ICSE' 2013*. IEEE Press, 802–811.

[25] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Vol. 15. 35.

[26] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 3–13.

[27] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256.

[28] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 298–312.

[29] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 643–653. https://doi.org/10.1145/2635868.2635920

[30] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 224–234. https://doi.org/10.1145/2491411.2491450

[31] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 94–105. https://doi.org/10.1145/2931037.2931054

[32] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2016. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *Empirical Software Engineering* (2016), 1–29.

[33] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 441–444. https://doi.org/10.1145/2931037.2948705

[34] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 448–458.

[35] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 691–701.

[36] Atif M. Memon and Myra B. Cohen. 2013. Automated Testing of GUI Applications: Models, Tools, and Controlling Flakiness. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 1479–1480. http://dl.acm.org/citation.cfm?id=2486788.2487046

[37] Nariman Mirzaei, Sam Malek, Corina S. Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. 2012. Testing Android Apps Through Symbolic Execution. *SIGSOFT Softw. Eng. Notes* 37, 6 (Nov. 2012), 1–5. https://doi.org/10.1145/2382756.2382798

[38] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically discovering, reporting and reproducing android application crashes. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*. IEEE, 33–44.

[39] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. 2013. Patchdroid: Scalable third-party security patches for android devices. In *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 259–268.

[40] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 772–781.

[41] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. 2009. Automatically Patching Errors in Deployed Software. In *SOSP*. 87–102.

[42] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, New York, NY, USA, 254–265.

[43] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. 2017. Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 300–311. https://doi.org/10.1109/ICSE.2017.35

[44] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. 2009. Fault Localization and Repair for Java Runtime

Exceptions. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA '09)*. ACM, New York, NY, USA, 153–164. https://doi.org/10.1145/1572272.1572291

[45] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 532–543.

[46] Mozhan Soltani, Annibale Panichella, and Arie van Deursen. 2017. A guided genetic algorithm for automated crash reproduction. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 209–220.

[47] Ting Su. 2016. FSMdroid: Guided GUI Testing of Android Apps. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. ACM, New York, NY, USA, 689–691. https://doi.org/10.1145/2889160.2891043

[48] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST '12)*. IEEE Computer Society, Washington, DC, USA, 260–269. https://doi.org/10.1109/ICST.2012.106

[49] Shin Hwei Tan and Abhik Roychoudhury. 2015. Relifix: Automated Repair of Software Regressions. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 471–482. http://dl.acm.org/citation.cfm?id=2818754.2818813

[50] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. 2017. Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. IEEE Press, Piscataway, NJ, USA, 180–182. https://doi.org/10.1109/ICSE-C.2017.76

[51] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 727–738.

[52] W. Weimer, Z.P. Fry, and S. Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE)*.

[53] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *ICSE*. 364–374.

[54] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. Lamelas Marcote, T. Durieux, D. Le Berre, and M. Monperrus. 2016. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* PP, 99 (2016), 1–1.

[55] Jifeng Xuan, Xiaoyuan Xie, and Martin Monperrus. 2015. Crash reproduction via test case mutation: Let existing test cases help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 910–913.

[56] Wei Yang, Mukul R Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 250–265.

[57] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 740–751. https://doi.org/10.1145/3106237.3106262

[58] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. 2017. A correlation study between automated program repair and test-suite metrics. *Empirical Software Engineering* (2017), 1–32.

[59] Mu Zhang and Heng Yin. 2014. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *NDSS*.