Mining Android Crash Fixes in the Absence of Issue- and Change-Tracking Systems

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

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

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

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

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

ABSTRACT

Android apps are prone to crash. This often arises from the misuse of Android framework APIs, making it harder to debug since official Android documentation does not discuss thoroughly potential exceptions.Recently, the program repair community has also started to investigate the possibility to fix crashes automatically. Current results, however, apply to limited example cases. In both scenarios of repair, the main issue is the need for more example data to drive the fix processes due to the high cost in time and effort needed to collect and identify fix examples. We propose in this work a scalable approach, `CraftDroid`, to mine crash fixes by leveraging a set of 28 thousand carefully reconstructed app lineages from app markets, without the need for the app source code or issue reports. We developed a replicative testing approach that locates fixes among app versions which output different runtime logs with the exact same test inputs. Overall, we have mined 104 relevant crash fixes, further abstracted 17 fine-grained fix templates that are demonstrated to be effective for patching crashed apks. Finally, we release ReCBench, a benchmark consisting of 200 crashed apks and the crash replication scripts, which the community can explore for evaluating generated crash-inducing bug patches.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Software libraries and repositories.*

KEYWORDS

Android, crash, testing, debugging, mining software repository

*Corresponding author.

ACM Reference Format:
Pingfan Kong, Li Li, Jun Gao, Tegawendé F. Bissyandé, and Jacques Klein. 2019. Mining Android Crash Fixes in the Absence of Issue- and Change-Tracking Systems. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3293882.3330572

1 INTRODUCTION

Nowadays, computing on Android-powered devices is pervasive across all age and status segments in society. A concern that is commonly shared by the whole user base, however, is that Android apps are prone to crash [45]. This is due to many reasons, including compatibility issues with devices, poorly tested functionalities, etc. A number of studies [25, 45] on the reviews that users provide on Google Play even list this proneness to crash as one of the most recurrent complaints about apps, especially in comparison with apps on the main concurrent system, namely iOS.

A major advantage of Android in attracting developers is that its maintainers provide a Software Development Kit (SDK) which builds on familiar programming concepts, languages and tools. An extensive open source Application Programming Interface (API) is further provided to facilitate the exploration of device resources, and programming features are regularly extended for building ever-fancy mobile apps. In this context, and given time-to-market pressure, most apps are shipped to the market without being fully tested. A situation that is exacerbated by the fragmentation of the Android operating system: a given app may eventually have to run on over 20, 000 unique device models [14] functioning at different API levels [45]. This, added to the fact that most Android app developers are not professional developers, leads to the situation where apps inevitably crash during operations.

Android app crashes yield stack traces pointing to a raised exception that developers must address. Framework crashes are the most difficult to debug, because the exception is thrown within the Android framework code, and developers can easily get lost in the enormous code maze. Although detailed documentation on API usage is provided by Google, and many testing platforms, as well as stack analysis tools, have been proposed by the research community, it is still far from being trivial to fix a crash. Nevertheless, with the huge interest in automated program repair in the software engineering community, there is an opportunity for building and applying some simple but effective online repair approaches targeted at Android recurring crashes.

Unfortunately, as Fan et al. [12] recently pointed out, there is a lack of comprehensive datasets of true crash fixes. This is a major obstacle which has prevented extensive research on crash analysis within the mobile community. Currently, authors turn to open source development code bases to track crash fixes based on the crashes reported in issue tracking systems. Fan et al. managed to identify 194 such issues for conducting a seminal study on Android crash fixes [12]. Such a process, along with the resulting dataset, presents several major limitations:

- There is a threat to external validity as only open source apps can be concerned. The dataset is not representative since a number of crashes may never be reported in issue tracking systems, and yet fixes have been applied to address them.
- The collection process of crash fixes is not scalable. Authors build crawlers to analyze GitHub repositories and select potential closed issues. Then, they must manually verify in the code that the issue is real and that the provided fix is indeed related to the announced crash.
- Open source apps often deal with simple functionality scenarios, and generally, have a smaller code base compared to commercial ones. Thus there may be fewer occurrences of crashes.
- Finally, this process cannot be replicated in commercial development settings, which do not provide useful information on bug reports, or means to reproduce bugs, and information on how they were eventually fixed. The limited information available is often within release notes where developers may vaguely mention that a bug fix was performed.

This paper. Our work in this paper deals with the automation of mining Android crash fixes in the wild. We propose to undertake an expensive testing campaign in a dataset of 450 thousand apks to replicate crashes in app lineages and retrieve fixes. Overall, the contributions of this paper are as follows:

(1) We design a workflow for collecting crash fix datasets in a fully automated and scalable manner in the absence of issue-tracking systems.
(2) We implement the necessary toolset and experiment on a large-scale dataset of apps from the AndroZoo repository [1] and benefit the community with a benchmark comprised of app crashes and scripts for automatic test input replication.
(3) We show that the fix templates mined are effective in patching crashed Android apps.

The remainder of this paper is organized as follows. Section 2 motivates this work with a concrete example. Section 3 presents the overall methodology of this work. Section 4 depicts the dataset we adopt and the results our approach obtains. We then evaluate our approach in Section 5, followed by a discussion and related work in Section 6 and Section 7, respectively. Finally, Section 8 concludes this paper.

2 MOTIVATING EXAMPLE

To better motivate this work, we now provide a concrete example demonstrating a crash and its fix that is, eventually, automatically obtained by our approach presented in this paper.

When random testing the *WhatsApp* app (release version of 30-09-2013), our approach observes a runtime crash. Listing 1 illustrates the crash message that can be retrieved (e.g., via the *logcat* command) from the device running the app. The crash message starts with an exception type, here *java.lang.SecurityException* (cf., line 1), which is thrown by the security manager to indicate that there is a security violation. Then, a detailed message will be given to explain the possible reason behind the crash. In this example, the reason is that neither user 10074 nor the current process has been granted with the following Android permission: READ_ PRECISE_PHONE_STATE (cf., line 2). After this detailed explanation, a call chain (cf. lines 3-8) will be listed showing the crash point (which is normally the first method in the call chain, cf. line 3) and the methods that are successively called until reaching the crash point. Observant readers might have already noted that the methods in the call chain include framework APIs (e.g., line 6, *android.telephony.TelephonyManager.listen*) and app methods (e.g., line 7, *com.whatsapp.RegisterPhone.onCreate*) that are actually written by app developers.

Listing 1: Crash Message of *WhatsApp.*

```
1 java.lang.SecurityException: Neither user 10074 nor current process
        has android.permission.READ_PRECISE_PHONE_STATE
2 at android.os.Parcel.readException
3 at android.os.Parcel.readException
4 at com.android.internal.telephony.
        ITelephonyRegistry$Stub$Proxy.listenForSubscriber
5 at android.telephony.TelephonyManager.listen
6 at com.whatsapp.RegisterPhone.onCreate
7 at android.app.Activity.performCreate
```

After observing the aforementioned crash, one can apply the same testing strategy (i.e., the same test inputs) to a later version (released on 30-04-2014) of the *WhatsApp* app. Interestingly, the app does not crash anymore. Hence, we can suppose that the later version of the *WhatsApp* app has somehow fixed the crash issue illustrated in Listing 1. To further verify the existence of modifications, we leverage SimiDroid [22] and perform a method-level pairwise comparison of these two app versions. Because framework APIs cannot be changed by app developers, we focus on app methods appearing in the call chain. Listing 3 summarises the *diff* message resulted from the pairwise comparison.

Fortunately, in this example, there is only one app method appearing in the call chain, which is *com.whatsapp.RegisterPhone.onCreate*. In this method, a framework API is used. The invoked API method is *android.telephony.TelephonyManager.listen*. This method registers a listener object to receive notification of changes in specified telephony states. In the crashed version, *int* -1 is passed as the second parameter while in the fixed version the value of the second parameter changed to 1535. By investigating the Android API documentation, we understand that the second parameter is the outcome of the bitwise `OR` operation of multiple `LISTEN_Flag`, representing the code for the state change that the app wants to listen to. As specifically mentioned in the documentation, some state changes are protected by permissions, i.e., developers need to declare the appropriate permission in order to access the state change information. The value, -1, is actually the outcome of the combination of all the possible state changes, where permission READ_PRECISE_PHONE_STATE is required. Unfortunately, this permission is not declared in the *AndroidManifest* configuration file, leading to crashes as expected. In the fixed version, the value of the second parameter is changed to 1535, after decoding it to binary, representing a combination of four flags, where none of them requires permission `READ_PRECISE_PHONE_STATE`.

Listing 2: Illustration of Mined Fix from *WhatsApp*.

```
void onCreate(android.os.Bundle $Bundle){
   ...
   - $TelephonyManager.listen($PhoneStateListener, -1)
   + $TelephonyManager.listen($PhoneStateListener, 1535)
   ..
}
```

This motivating example demonstrates that it is possible to learn practical fixes to Android runtime crashes by learning from the code contributed by app developers (e.g., during the evolution of a given Android app). In this work, we call the two app versions ($Apk_{30.09.2013}$, $Apk_{30.04.2014}$) as a crashed-fixed pair. Our objective in this work is hence to mine crashed-fixed pairs from real-world Android apps so as to learn fix hints from them and benefit the research community with such dataset.

3 APPROACH

Figure 1[1] presents an overview of the proposed `CraftDroid` approach. It is composed of two phases.

Phase 1: The *Fix Mining* phase consists of 3 steps. The objective of this phase is to mine fixes to Android crash-inducing bugs through a carefully designed approach leveraging both testing and static analysis.

Phase 2: The *Fix Grouping and Fix Template Abstraction* phase utilizes another 2 steps, to group the aforementioned fixes based on testing results feature identification and a manual abstraction method to abstract fix templates.

The crashed apks set as well as the test inputs replication scripts from the *Fix Mining* phase further form a benchmark called *ReCBench*, and will be utilized to showcase the effectiveness of our approach, as well as a contribution to the community.

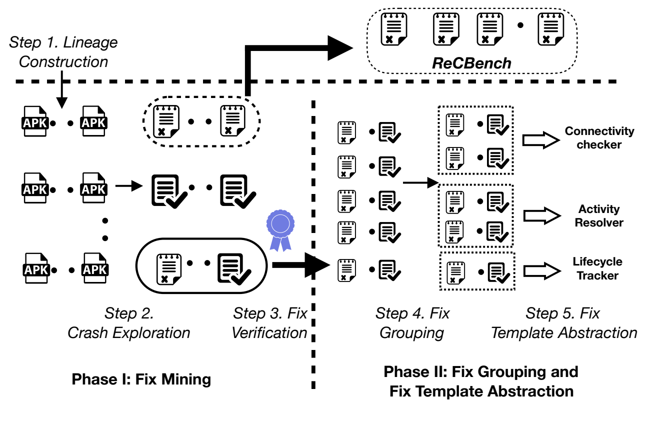
Step 1. Lineage Construction
ReCBench
APK
APK
Connectivity checker
Activity Resolver
Lifecycle Tracker
Step 2. Crash Exploration
Step 3. Fix Verification
Step 4. Fix Grouping
Step 5. Fix Template Abstraction
Phase I: Fix Mining
Phase II: Fix Grouping and Fix Template Abstraction

Figure 1: Overview of CraftDroid.

[1]Icons from www.flaticon.com

3.1 Phase I: Fix Mining

The first phase unfolds in three steps: (1) App Lineage construction, (2) Crash Exploration and (3) Fix Verification. Each of these steps is implemented in a module whose details are provided below.

Step 1 - App Lineage Construction. The concept of app lineage (i.e., a series of apk releases of a given app) was first introduced by Gao et al. [13]. In this step, we use the same approach to construct app lineages from AndroZoo apps. Overall, the app lineages are constructed via the following process: (1) identify unique apps, where APKs sharing the same *package name* are considered to be the same app, and (2) link and order the different app versions of the same app. As a result, an app lineage contains a set of Android apps that share the same package name while are totally ordered based on their release time. Note that a lineage can be sparse given that AndroZoo is not exhaustive in the collection of app versions. Nevertheless, as we will show in the next phases, this does not impact the soundness of `CraftDroid`.

Step 2 - App Crash Exploration. This step aims to pinpoint real-world apps that crash at runtime. The most straightforward and reliable approach to achieve that objective is to launch a testing campaign on the apps. In our case, we generate diverse UI inputs, aiming to automatically and dynamically explore Android apps. Ideally, in order to cover as many crash cases as possible, we should employ different testing strategies. However, as dynamic analysis tools are known to be time-intensive, it is practically hard to adopt all possible random testing tools available in the Android community. Moreover, during our testing step, we need (1) to ensure that the testing tools generate the same sequence of testing inputs, (2) to ensure that the testing environment, i.e., the Android device emulator and the state of the operating system are identical for the apks of the same lineage (3) to ensure that our approach is scalable in order to quickly explore all apks for a huge amount of lineages within a limited time budget. We now detail these three requirements.

Replicative Testing Inputs. In order to achieve optimized crash exploration and to apply the same execution scenario, we utilized 3 testing policies from 2 testing tools. Monkey is a built-in random testing generation tool for Android emulators. The testing procedure will be reproducible if the selected seed is identical. Droidbot [28] is a light-weight test input generation tool for Android apps. It generates input events based on the analysis of *AndroidManifest.xml* and runtime objects. However, *Droidbot* currently does not have an option for reproducibility. For the purpose of this work, we contribute to DroidBot by implementing a reproducibility module. This module essentially records the sequence of events generated for the first app version of any lineage. If needed, the module can apply the recorded events to new app versions so as to reproduce the events fired previously. We utilized two of *Droidbot*'s testing strategies, *dfs_greedy* and *bfs_greedy* for better coverage.

Restored Testing Environment. To ensure that each app version is tested in an identical environment, we used the same emulator instance for all app versions of any lineage. In order to mitigate the potential influence of remnant files and caches from both app installation time and runtime, we uninstall the previous app versions and carefully clear all temporal files prior to new app version testing steps. To make sure the test log collected is confined for each testing procedure, we also clear the buffer for the logging system in the emulators.

Scalable Testing Process. Since the testing process across different lineages is fully parallelized, we leverage the abilities of multi-core computing machines. `CraftDroid` detects the number of CPU cores on the computing device and starts as many Android emulator instances. Each lineage will be tested on a single instance and the whole process will scale to the number of available cores.

After the test procedure finishes on each apk (for each different test strategy/tool), we use the built-in logging tool *Logcat* from Android emulator to filter out the testing information. The log will contain basic information about the app, e.g., app package name, process id, etc. Especially, when the app crashes during the testing procedure, the crash log, including the exception type, crash message and stack trace will also be logged. Furthermore, we collected 200 crashed apks, each from a unique lineage, together with the scripts to reproduce them to form a benchmark called *ReCBench* (*Re*producible *C*rash *Bench*). *ReCBench* will be used in below sections to the evaluation of the effectiveness of our approaches, and also to benefit the community.

Step 3 - App Fix Verification. After the testing process finishes in *Step 2*, we then start to look for app crash-inducing bug fixes.

Analyzing Testing Logs. First, we analyze the test logs. To start, we give the definition of fixes present in lineages. Given an Android app a_1 that crashes following test inputs t, if a_1's subsequent version a_2 does not crash following the same test inputs t, we hypothesize that a_2 has been released with fixes to the crash appearing in a_1. In other words, a_2 contains practical code changes addressing the crash issue observed in a_1. Given an Android app a_1 with its crash information (i.e., stack trace), the objective of this step is to identify app a_2, which must thus satisfy two conditions:

- a_2 is released after a_1.
- a_2 no longer crashes following the same test input.

If such an app is identified, we mark it as the fixed version and hence put it with app a_1 into a pair, i.e., the potential "crashed-fixed" app pair (a_1, a_2). To avoid flaky cases, we further impose restrictions on our approach to only consider such cases where the subsequent app versions throughout the end of the current lineage do not crash anymore under the same tests.

Inspecting Code Changes. Then, with the potential crashed-fixed app pair identified, our approach performs a pairwise comparison between the two apps and records all the changes as potential fix changes. Unlike when working with change-tracking systems, app version updates represent a series of changes performed for various reasons, including adding new functionality to the app, fixing various functional and non-functional bugs, etc. Thus, not all recorded changes are relevant as crash fix modifications. Therefore, we limit ourselves to such changes that are made to the methods appearing in the stack trace of the runtime crash.

A stack trace, as illustrated in Listing 1, includes details on the method during whose execution the crash occurs, as well as the path (i.e., call chain) of execution flow that led to this point.

In this work, we consider all the methods that are from the Android framework as framework methods, and those that are not from the Android framework as app methods which should be considered to look for modifications that fix crashes in apps. Depending on the signaler, i.e., the framework method which instantiates and throws the exception type, app crashes are divided into *framework-specific* crashes and *app-specific* crashes. In this paper, we only study framework-specific crashes. We consider app-specific crashes to be out of the scope of our research interest, given that they mainly fall into below three categories, which can be readily diagnosed and whose fix patterns are straightforward to craft:

(1) Most exceptions thrown are *NullPointerException*. This exception happens when the program tries to invoke a method on a null object reference, it often arises because of the developers forget to initialize variables or adding null checkers.
(2) *ClassCastException* is thrown when developers try to cast a runtime instance to a nonmatching type.
(3) Exceptions created and thrown by app methods. This means that the developer is aware of the condition when the exception is thrown and the fix will be very much app specific.

On the contrary, we find that framework-specific crashes and their fixes are more challenging and valuable for mainly 3 reasons:

(1) The signalers that throw the exception instance are system methods, and are unseen to developers unless developers are very aware of AOSP (Android Open Source Project).
(2) There might be long call layers between the signaler and the API that developer called, making the condition for triggering the exception more complicated.
(3) The Android documentation is never fully clear about when and how an exception will be thrown upon using an API.

Since our apps are all tested on the same Android emulator instance, we consider that the framework method implementations are identical among stack traces, and thus we only need to compare the app methods. For each app method (e.g., *DevMethod1* in Figure 1), we perform a pairwise comparison at the code level between the two apps in the crashed-fixed pair and generate a *diff* snippet representing the changes made by the fixed app version. If any modification is present, we will manually check whether the modification is a true fix to the prior crash.

3.2 Phase II: Fix Grouping and Fix Template Abstraction

After collecting true fixes in Phase I from `CraftDroid`, we extract features from the stack trace of crashes, and group the fixes into buckets based on the features. Subsequently, we shall abstract fix templates from the groups. These fix templates are patterns of code change actions that can be applied to crashed apks in the cases of specific crash types.

Step 4 - Fix Grouping. For each crashed-fixed app pair (a_1, a_2), we analyze the crash stack trace from a_1 and collect 3 feature information: *Exception* is the exception class thrown that caused the crash; *Signaler* is the topmost system method that created and threw the exception; *CrashLoc* is the framework API that the app method called and which passed back the exception. Let us take Listing 1 as an example. The extracted features are *(SecurityException, readException, listen)*. Note that in this example, for better writing structure we used only method names, wherein actual work we use full-length identifiers. Fan et al. have also constructed similar tuple for grouping and finding the root causes for Android crashes. However, our tuple is different since we adopted *CrashLoc* instead of crash message. Because we consider the fixes are closely related to the context of the usage for the APIs. With the features extracted for each crashed-fixed pair, we then group fixes simply based on comparing their feature tuple and build buckets for these fixes. In other words, all fixes with the same extracted features from the crash stack traces (i.e, the three elements $Exception, Signaler, CrashLoc$ are identical) are grouped into the same bucket.

Step 5 - Fix Template Abstraction. For each bucket, we follow the below protocols to manually look for fixes.

(1) Based on the stack trace, we start from studying the code changes in the developer method which calls the crashed Android framework API, i.e., the crash location.
(2) If fix is not found, we turn to the next developer method towards the bottom of the stack trace. We repeat this, until we find a fix or we reach the last developer method.
(3) Once we find a fix, we turn to official Android development documentation, as well as online discussion forums to validate our finding.
(4) If no fix is validated after studying all developer methods, we consider in this work that the code changes are irrelevant.

From the validated fixes, we summarize the most common fix template. These templates are fine-grained since they target to only solve crash-inducing bugs related to the misuses of specific framework APIs. Although automatically mining of templates is researched in the literature (cf. recent work of Koyuncu et al. [19]) they often lead to noises about patterns unrelated to repair changes due to the issues of tangled commits [9, 15, 30, 40]. We leave automation in this part as future work.

3.3 Patching Crashed Apks

Towards demonstrating the usefulness of `CraftDroid`, as well as the fix templates generated, we propose to apply the templates on crashed apks and evaluate the patched apks. As shown in Figure 2, we take crashed apks from ReCBench (cf. Section 3.1). As shown in Figure 1, this benchmark comes from *Phase I*. ReCBench is built by collecting the crashed apks and the associated testing inputs. For a given crashed apk a_c from ReCBench, to evaluate our fix templates, we follow a three-step process: 1) First, the crashed apk a_c is categorized into one of the buckets we collected by extracting features from the crash stack trace of a_c. 2) Then we use Soot[44] to decompile a_c into Jimple files[2] and retrieve the file that contains the app method which called the API. We manually apply the fix template associated with the corresponding bucket. 3) We re-compile the files into apk and run the test with the patched apk. This process was first introduced in [43]. If the patched apk does not crash with the same testing inputs, we consider our template valid.

[2]Jimple is the intermediate representation of Soot

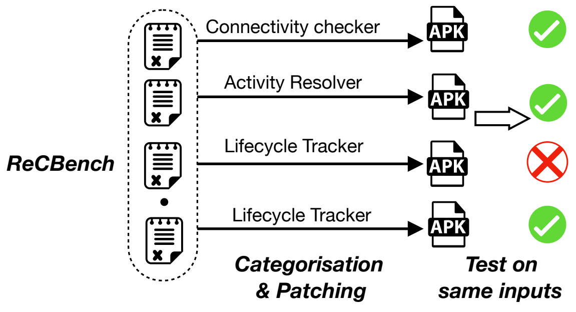

Figure 2: Patching and Evaluation.

4 DATASET AND STATISTICS

4.1 Crash Fixes from Lineages (from Phase I)

Lineage Construction. We start by retrieving a maximum number of apps from the AndroZoo repository [1] using the API key provided by the maintainers of AndroZoo. We managed to collect 4 million apps to start our experiments. Although AndroZoo currently makes available over 8 million apps[3], we believe that the collected dataset is largely representative of this study. From the collected app set, we eventually re-constructed $28K$ app lineages containing around 450 thousand apks: although we could find many more lineages, we focused on cases where there are at least 10 app versions. The purpose of setting this threshold is that we want to mine for code changes among released versions that would result in contrary testing outcomes. Shorter length of lineage would not be able to reflect such changes, and would pose obstacle of filtering out flaky outcomes.

Test Environment Set-up. In order to set up the testing environment for the lineages, we instantiate multiple Android emulators using the same image so that we can do testing parallelly and make full use of multi-core computing machines. Note that we have chosen x86 as ABI (Application Binary Interface) for these emulators. Since we are running these emulators on x86-based servers, choosing x86 over arm64-v8a as ABI makes emulators run substantially faster, because the instruction set is shared among the virtual machine and the host machine. The drawback is that apps making use of the Android Native Development Kit (NDK) are not installable and would report an installation error complaining about non-matching ABI. During our experiment, we have also observed other kinds of installation errors (e.g., the builds are too old w.r.t. API level). In total, around 56% apks are installable and later tested on the emulators. Although the percentage for the set of installable apks over all apks are not ideal, we consider it acceptable since emulators run much faster than real-world Android devices. Moreover, the scalability of emulators makes it capable of running a large collection of lineages in reasonable time budget. During our experiments, we have observed that if one version in a lineage cannot be installed on the emulator, its succeeding releases (app versions) often cannot be installed as well. However, conversely, when a version can be installed, its succeeding releases often can be installed as well. As is illustrated in Figure 3 the median of the

[3]https://androzoo.uni.lu/

total number of apks in a lineage should be close to the median of the number of installable apks in a lineage. Note that since every lineage contains at least 10 apks, the minimum number of apks is 10. For better illustration, lineages that contain no installable apks or no crashes are not considered. Outliers are also removed since in rare cases lineages have great length, i.e., the version updates are very frequent.

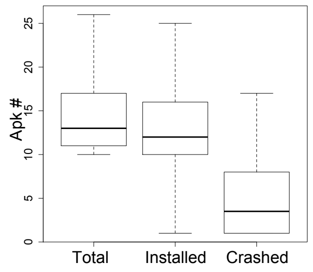

Figure 3: Distribution of Total, Installed and Crashed Numbers of Apks in Lineages.

Test Strategy Application. For the lineages with apps that are installable on our emulators, we launch Monkey as well as Droidbot (using two strategies in the case of this tool) to automatically explore their functionalities, aiming at obtaining runtime crashes. Figure 3 presents in the box-plot the distribution of the number of apps that have framework-specific crashes revealed in each lineage. The median number of crashed apks in a lineage is around 3. We then show the results for the 3 testing strategies in Figure 4. The *dfs_greedy* strategy of the tool *DroidBot* exposed most lineages with at least one apk having framework-specific crashes. Among those 1160 lineages, there are 371 where successive apks passed the same test inputs and do not crash any more. For the *bfs_greedy*, a similar number of crashed lineages and fixed lineages can be observed. However, with random tests using seed $NO.12$ from *Monkey*, we can see that the number of lineages identified to have crashed apks are only half the total number of that of the *dfs_greedy* or *bfs_greedy* strategies of *DroidBot*. This difference in the numbers is because *DroidBot* analyses the *AndroidManifest.xml* and other layout definition files in the apk before generating UI inputs, this gives *DroidBot* higher chance of triggering code execution compared to Monkey. Better coverage, in turn, will result in more crashes being detected. In the meantime, we also see from Figure 4 that all 3 strategies would detect similar portion of passed lineages out of lineages with crashes, this indicates that `CraftDroid` is insensitive in the testing tools for mining fixes from the same number of crashed lineages.

Fix Statical Identification. For the lineages selected from the previous steps, we construct a *crashed-fixed* app pair for each lineage. The crashed apk is the last apk in the lineages that has crashed during testing, the fixed apk is the first apk that has passed with the same testing inputs. The reason for the selection is that we consider two adjacent versions will contain the smallest set of code changes related to the app methods that caused the crash. We then use the static analysis tool Soot to decompile both apks

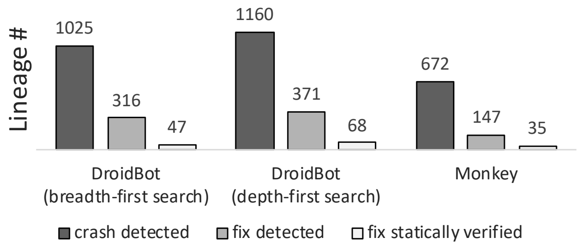

Figure 4: Count of Lineages Crashes per Testing Strategy.

and compute diffs on the app methods in the stack trace. For example, from Listing 1, the app method that will be compared is *com.whatsapp.RegisterPhone.onCreate*. After static analysis finishes, we succeeded to verify 150 fixes in total, as can be seen in Figure 4 (47 + 68 + 35 = 150). We then manually inspected these *diffs*, we finally distilled 104 useful fixes. We removed other 46 fixes for mainly two reasons, 1) The API call which throws the exception in the crashed apk is no longer used in the new apk version, 2) The API call appears multiple times, e.g., in a switch-case clause and is uncertain in which call statement the exception was thrown. Although programmatically those two kinds of code changes are still valid in fixing crashes, we consider them not helpful for the next phase of our approach.

4.2 Fix Buckets & Fix Templates (from Phase II)

In the second phase, as described in Figure 1 and explained in SubSection 4.2, we group the fixes into buckets and abstract fix templates from each bucket.

Grouping for Fix Buckets. From the previous step, we have confirmed 104 true fixes. We extract features to form a tuple (*Exception*, *Signaler*,*CrashLoc*) from the stack trace of the log of the crashed apk of the *crashed-fixed* pair. When features are the same, fixes are grouped into the same buckets. As a result, we grouped these fixes into 35 buckets. Due to the limited space in this paper, we are not listing the full detail of the buckets and the features related. We rather summarized the bucket count where same exception classes are thrown in Figure 5. The most recurrent exception is *NullPointerException*, taking 8 buckets in total, arising from passing null parameters in API calls. The main reason for the crashes are programs trying to call methods on null object references. The second most common one is *WindowManager$BadTokenException*, with a total of 4 buckets. This exception is thrown when the hosting activity upon which the dialog wants to show its message has entered the finishing state. *RuntimeException* is the super class of the unchecked exceptions, it reflects general problem when running the Android app. *Resources$NotFoundException* is thrown when the correspondent resources, e.g., String, Figure, are not found in the apk. *IllegalArgumentException* is thrown when the passed in parameter for the API cannot be handled by the Android system and will be considered illegal. *ActivityNotFoundException* is thrown when no activity is found to handle intents. Android uses intents to start new activities, both inside the current app and activities in other apps in the device. An implicit intent is used when the current app wants to start the activity from other apps on the device that can

Table 1: Fix Templates

Bucket ID	Acronym	Description	Target Exception
1	Provider Checker	Check if the specified provider is enabled with location manager.	IllegalStateException
2-3	Activity Resolver	Resolve intent for Activity existence before starting new Activity	ActivityNotFoundException
4-6	Resource ID Updater	Switch resource ID with another one	Resouces$NotFoundException
7-10	Lifecycle Verifier	Query the state of the hosting activity upon showing dialog	WindowManager$BadTokenException
11	Sleep not Wait	Call sleep() instead of wait() on Thread	IllegalMonitorStateException
12	Thread Finisher	Switch from call of stop() to interrupt() to end thread	UnsupportedOperationException
13	Redundance Trimmer	Trims redundant call of prepare() on same Thread	RuntimeException
14	State Checker	Check check value of isAlive() on Tread before calling start()	IllegalThreadStateException
15	Package Settler	Set package name for intent upon biding service	IllegalArgumentException
16	Permission Checker	Check permission before sensitive operation	SecurityException
17-19	Range Checker	Check on index range before indexing	(String)/(Array)IndexOutOfBoundsException
20	Emptiness Checker	Check if String is Empty before parsing for int	NumberFormatException
21	Path Verifier	Check the file path existence for database	SQLiteCantOpenDatabaseException
22	Hardware Checker	Check the hosting device has specified hardware feature	RuntimeException
23	Canvas Preconcator	Pass preconcated $Canvas to unLockCanvasAndPost()	IllegalArgumentException
24-31	Nullable Checker	Check for parameters that is not nullable	NullPointerException
32-35	Try-catcher	Surround the statement with try-catch	Exception

perform the required *Action* by the intent. However, there is no guarantee that at least one activity exists on the device that performs the action. Improperly handling these scenarios will cause the *ActivityNotFoundException* being thrown. *SecurityException* is thrown when the required permission has neither been granted to the app at installation time nor runtime. The app *WhatsApp* in the motivation example takes this category. Other exceptions, although occurred only once, also arise from key defects in the app codes.

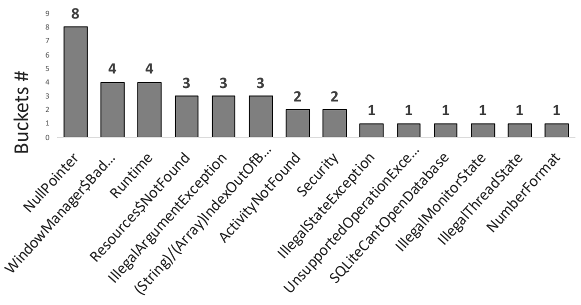

Figure 5: Bucket Count for Exceptions.

Abstracting Fix Templates. We then manually extract templates from buckets. Note that similar or same fix templates can be extracted for different buckets. This is especially true when the exception and the *Signaler* is the same. As a result, we are able to settle down 17 fix templates, as listed in Table 1. We give an acronym for each fix template for easier reference in the rest of the paper. *Provider* is a source for providing location information, e.g., network, gps e.t.c.. *Provider Checker* checks on the return value of *isProviderEnabled("provider")* before the app executes *requestLocationUpdates("provider).* Omitting this when the provider is not enabled will make the system throw *IllegalArgumentException.* The Android system uses *Intent* to start new activities, especially, implicit intents to filter for activities in other apps that can fulfill the required action. However, when no such activity exists in current device, *ActivityNotFoundException* will be thrown. *Activity Resolver* checks for the existence of targeting *Activity* before the app tries to start one. Android apps widely use multi-threading technique to divide light-weight UI response logic and heavy load background logic like downloading. Novice developers will thus be easily confused and make mistakes when handling the threads. We also established 4 templates for fixing thread related crashes. *Sleep not Wait* updates the call from wait() to sleep() on the thread when the app requires a background thread to block other access requests for a certain time period. *Thread Finisher* updates the call from stop() to interrupt() when developer intends to put an end to the thread. *Redundance Trimmer* aims to trim redundant call to prepare() on a thread since it poses *RuntimeException. State Checker* aims to check if the thread is alive before calling start() on the thread. *Package Settler* aims to set package name for intent upon biding service, failing this will make the system throw *IllegalArgumentException.* Fix templates 10-15 contain a set of checkers and verifiers for file existence, parameter, permission e.t.c. These are all easily neglected due to uncareful coding. *Canvas Preconcator* aims to preconcate the current canvas instance before calling unLockCanvasAndPost() by calling translate() on the canvas first. The last fix template is *try-catch.* We use *Exception* to summarize that this fix is widely used across multiple buckets by developers. It is often used to deal with corner cases that do not have many occurrences. Developers tend to be lazy for these scenarios. Similar to Tan et al.[43], we consider this template a hard fix, and will not recommend it since it actually suppresses the root cause of the exception.

4.3 ReCBench for Evaluating Bug Patches

As explained in SubSection 3.3, we take from Phase I the lineages whose testing results contain only crashed apks and propose ReCBench, a benchmark for evaluating patches for crash-inducing bugs. This benchmark contains 200 crashed apks, the crash message filtered out from *LogCat*, and the scripts to automatically reproduce the crashes. The script can be used to re-generate test inputs on the patched apk and filter out log information to decide whether the patch is valid. The dataset is accessible at https://craftdroid.github.io

5 EVALUATION

To evaluate the proposed workflow CraftDroid and the fix templates generated, we further address 3 research questions.

- **RQ1:** How do the automatically explored crashes compare with the ones reported in issue-tracking systems?
- **RQ2:** Are state-of-the-art benchmarks capable of evaluating patches generated for crashed Android apps?
- **RQ3:** How good are our mined fix templates in patching crashed Android apps?

5.1 Explored Crashes vs. Reported Crashes

In Phase I of CraftDroid, we adopted 3 automatic testing strategies to explore inputs that would crash Android apps. However, there is already datasets in the research community that was collected by scanning the issue-tracking systems on open-source App repositories, like the one proposed by Fan et al. [12]. This dataset contains 194 app crashes reported by developers or general users throughout their daily development and usage. We want to compare this dataset with ours to compare the automatically explored crashes and reported crashes in answering 1) Whether our dataset is realistic in reflecting a portion of the crashes cared by users and 2) Whether the automatic testing tools can provide more crash cases omitted by human beings. Please be noted that we utilize our dataset of 104 lineages equipped with fixes, rather than all crashes explored. Similarly, the dataset from Fan et al. also contains crash reporting issues that are closed eventually, i.e., fixed by developers rather than open issues that are not necessarily fixed. The purpose of putting these constraints (i.e., lineages with fixes for CraftDroid, and closed issues for Fan et al.) is that we consider these two sets of crashes valid in the sense that they are cared by developers and actually fixed in the later versions and(or) commits. For comparison, we also applied the feature extraction and fix grouping method as presented in Step 4 on the dataset of Fan et al. Recall that the study of Fan et al. collected stack traces in issue tracking systems (although, in the absence of input information, the associated crashes are not reproducible, unlike in CraftDroid). As can be seen from Table 2, there are in total 9 buckets shared by both datasets. Moreover, CraftDroid is able to spot 26 exclusive buckets not reported by issue-tracking systems. Although there are another 117 buckets from the issue-tracking system that we did not cover, given the fact that the authors claimed that they have exhaustively collected all such reported crashes from the online repository collection F-Droid [11] for open source Android apps, we give credit to CraftDroid of being scalable and having the potential of exposing more such crashes, e.g., with other testing tools for larger code coverage.

Table 2: Buckets Count between Fan et al. and CraftDroid

Dataset	Total Bucket	Exclusive Buckets	Shared buckets
Fan et al. [12]	126	117	9
CraftDroid	35	26	

Answer to RQ1: CraftDroid is able to explore android app crashes that are also reported by issue-tracking systems, with the ability to also cover more crashes that are not yet reported by developers or users.

5.2 Benchmarks for Evaluating Patches

Several benchmarks containing Android app crashes have been proposed recently, e.g., Fan et al [12] collected 194 closed issues. Tan et al. also proposed Droixbench [43]. Droixbench is proposed to evaluate the effectiveness of the automatic patch generation tool Droix. The benchmark contains 24 reproducible app crashes. To reproduce the UI inputs sequences, a runnable script is provided for each crashed apk. However, we found out that Droixbench contains only 9 framework-specific crashes, where the other 15 are app-specific crashes. Since we focus on framework-specific crashes and their fixes, Droixbench does not meet our need. Further, we discover that only 3 out of 9 framework-specific crashes are fixed by Droix, resulting in a fix rate of only 33%. This is much lower than the fix rate of app-specific crashes, where 12 out of 15 crashes are fixed, with the fix rate of 80%. This also supports our observation that framework-specific crashes takes great effort to fix and should draw more attention.

As mentioned in RQ1, Fan et al. [12] also propose a collection of closed issues for framework-specific Android app crashes. We found 3 obstacles in using this dataset. 1) There is no guarantee of apk or version number associated with the issue, people who study this dataset often need to take into consideration the opening time of the issue track, and deduce the version by her/himself. 2) The testing environment, i.e., the specific phone model, the phone states when the crash happens are not easy to re-establish. 3) Most importantly, there are often no clear reproducing steps attached to help reproduce the crash. In most cases, only natural language descriptions are provided. Such shortcomings make it almost impossible to evaluate the patches for the same test inputs.

We thus propose *ReCBench*, a benchmark containing 200 reproducible framework-specific crashes and the scripts to reproduce them. As can be seen in Table 3, *ReCBench* contains the higher number of framework-specific crashes among the three benchmarks. Moreover, the total number of app is also 200, which means that it provides higher diversity in terms of the categories of apps. In comparison, the dataset of Fan et al. only contains 44 apps. All the 200 crashes in *ReCBench* are framework-specific. Most importantly, *ReCBench* provides scripts to automatically reproduce the UI inputs that triggered the crashes. Finally, *ReCBench* is largely extensible: in this study, time constraints for ensuring, via extensive experiments, that crashes are reproducible limited the dataset to 200 samples. Our crash exploration phase yielded thousands of crashes which could be later validated and included in *ReCBench*.

Table 3: Comparison among benchmarks

Benchmark	App#	Size	Framework	Reproducible
Droixbench	15	24	No	Yes
Fan et al.	44	194	Yes	No
ReCBench	200	200	Yes	Yes

Answer to RQ2: ReCBench contains 200 framework-specific crashes, and stands out in automatic reproducibility, diversity in app categories and the total size of crash collection.

5.3 Evaluating Fix Templates on ReCBench

The objective of RQ3 is to assess the quality of the 17 fix templates yielded by `CraftDroid`. To that end, we first consider crash samples from *ReCBench* that are selected based on the below criteria:

(1) We focus on crash samples that are relevant to the universe of crashes for which the fix templates were inferred. Concretely, we sample crashes from the 35 buckets yielded in Phase I.
(2) Diversity over quantity. We select crashes to consider as many buckets as possible.
(3) After a sample of each represented bucket is selected, we consider more samples corresponding to most represented buckets (in this case, *ActivityNotFoundException*-related crashes).
(4) To avoid bias, we ensure that we consider crashes for which no fix was ever found after crash exploration.

Due to time and repair execution constraints, we eventually selected 20 crashed apks as shown in Table 4. For the 20 crashed apks, 17 were fixed, meaning that the patched apk does not crash in same testing inputs. However, there are still 3 apks that did not pass the test. The original version of *WordPress* crashed because it attempted to get index on a substring that does not show in the target string. Although we managed to do a range check and prevented this inappropriate call, succeeding call to the same API again crashed the app. This indicates that the developers tend to make the same mistakes on the same API usage. The patched *SetCPU* fails because we are trying to check whether the string is empty by calling isEmpty() on the string before it is used to parse for int. However, the fix template did not anticipate that the string itself is never instantiated, so the NPE(*NullPointerException*) was thrown and still crashes the app. *FingerWQ* was not even successfully patched because Soot was not able to create the *Jimple* file of the targeted app code from the apk, so the patching process was not carried. Overall, the 20 crashed apks were intended to be patched thanks to 9 out of our 17 fix templates.

In Listing 3, we list 4 patches successfully generated and evaluated to be effective. Since we are using Soot to patch apks, the patches are in fact in *Jimple* code. However, for readability purpose, we illustrate in an equivalent *java* format.

The first app is *AutoHome*, it is a forum app for discussion and information sharing for car fans. On showing the dialog alerting no network connection is present, it encounters *WindowManager$BadTokenException*, it indicates that the activity where the dialog wants to show above has been destroyed. The correct way is to check the activity's lifecycle before calling the show() method, it is therefore fixed by our *Lifecycle Verifier* template. This exception exists in 4 buckets we collected in Section 4. The bug reasons as well as the correspondent fix template are also same for the buckets. The second app is a train tracking app which require location update information. However, since network is turned off for the emulators, requesting location updates from network will make the *IllegalArgumentException* being thrown. The correct way is to check the provider before requesting. This app is fixed by our *Provider Checker* template. The third app is a photography app that has over 100*k* installs from Google Play. We found that it crashes when it tries to bind to the app billing service. The root cause is that it did not set package name of the binder class for the intent instance. It is therefore fixed with correctly setting the package name, with our *Package Settler*. The fourth app is *Fruit Mahjong*, a gaming app. It was fixed by replacing from call of stop() to call of interrupt() to thread to prevent *UnsupportedOperationException* being thrown.

Table 4: Patch Evaluation on RecBench

App Name	Bucket	Applied Template	Fix	Remark
AutoHome	8	LifeCycle Verifier	yes	
PI	8	LifeCycle Verifier	yes	
JadwalKA	12	Thread Finisher	yes	
Fruit Mahjong	12	Thread Finisher	yes	
Flashlight	1	Provider Checker	yes	
areain!	1	Provider Checker	yes	
WordPress	18	Range Checker	no	crash with succeeding API
Android Optimizer	18	Range Checker	yes	
Mine_mine	20	Emptiness Checker	yes	
SetCPU	20	Emptiness Checker	no	String null throw NPE
FingerWQ	23	Canvas Preconcator	no	app method non existing
BTCfx	15	Package Settler	yes	
MapCam	15	Package Settler	yes	
Baby Piano	2	Activity Resolver	yes	
GK in Gujarati	2	Activity Resolver	yes	
Reflection	2	Activity Resolver	yes	
UK Lotto	2	Activity Resolver	yes	
Agile Buddy	2	Activity Resolver	yes	
HiYou Park	33	Try-catcher	yes	
Sohu Weibo	33	Try-catcher	yes	

Listing 3: Fix Examples for Crashed Apps.

```
//App: AutoHome (forum app for car fans)
//Exception: WindowManager$BadTokenException
//Applied template: Lifecycle Verifier
+ if (!activity.isFinishing){
        alertDialogBuilder.show();
+ }
//App: areain! (train tracking app)
//Exception: IllegalArgumentException
//Applied template: Provider Checker
+ if(locationManager.isProviderEnabled("network")){
     locationManager.requestLocationUpdates("network"
                                              ,5000L,100.0F,this);
+ }
//App: MapCam (photography app)
//Exception: IllegalArgumentException
//Applied template: Package Settler
  Intent intent = new Intent("com.android.vending
                              .billing.InAppBillingService.BIND");
+ intent.setPackage("com.android.vending");
  this.bindService(intent, serviceConnection,1);
//App: Fruit Mahjong (gaming app)
//Exception: IllegalArgumentException
//Applied template: Thread Finisher
- thread.stop();
+ thread.interrupt();
```

Answer to RQ3: Our mined fix templates are in general efficient in patching crash-inducing bugs caused by wrong usage of framework APIs, although in minor cases they might cause regression.

6 THREATS TO VALIDITY

External validity. Our findings may be biased due to the app dataset (of free apps) that we used for mining as well as the testing tools that were leveraged in CraftDroid. Nevertheless, we tried to mitigate these threats by considering the largest repository of Android apps available to researchers. We also tried three different test strategies to improve coverage.

Internal validity. Our method of localizing crash-inducing bugs includes some threats to validity. We utilize stack trace for fault localization, and assume that the crash-inducing bug arises in one of the app methods in the stack trace. However, it is possible that such bug can also reside in app methods that do not show in the stack trace. For example, the *NullPointerException* can be thrown because a field variable is not initialized during the instantiation of the type. In this case, the bug location is different from the crash location. However, we consider our work still valid since it has been reported by this empirical study [46] that 59% to 67% crash-inducing bugs actually reside in stack traces. Although it indicates that we might miss fixes during the mining process, it does not affect the validity of CraftDroid.

7 RELATED WORK

Android app analysis has been a hot topic for many years. Different approaches, including both static analysis and dynamic analysis, have been proposed to tackle various issues in the mobile realm such as privacy leaks detection [4, 21], repackaged apps identification [27], etc. Our approach, targeting the runtime crashes of Android apps and their potential fixes, has adopted both static and dynamic analysis techniques.

Android Crash Analysis. The most closely related works to ours are by Tan et al. [43] and Fan et al. [12]. Tan et al. [43] present a benchmark of Android app crashes and their potential fixes to the community for exploring automated crash fixes. Their benchmark, namely DroixBench, which is built by mining open-source projects, contains only 24 samples, which are unfortunately not representative to real-world app crashes and fixes since over half of the thrown exceptions are *NullPointerException*. Similarly, Fan et al. [12] have also mined open-source project tracking systems for identifying reported crashes and associated fixes. Both of these two approaches leverage open-source projects to identify crash fixes. Our work, however, is different given that we attempt to identify crash fixes from closed-source apps. We thus complement the state-of-the-art with more dataset of crash fixes and further provide new means to harvest crash fixes in closed-source settings. The fact that a large majority of Android mobile apps are closed-source [1] suggests that CraftDroid has a high potential for researchers to mine ever more crash fixes than with the previous approaches.

State-of-the-art works are also interested in reproducing runtime crashes [7, 42, 47]. For example, Xuan et al. [47] leverage mutation testing to reproduce crashes while Soltani et al. [42] leveraging genetic algorithms on the inputs to reproduce crashes. Recently, Moran et al. [39] proposed CRASHSCOPE, a testing tool for automatically discovering crashes, saving log related information, and generating useful reports. Along with the tool, the authors released only 8 crashes from 20 Android apps together with reproducing scripts and detailed reports. Such a tool could, however, be integrated into CraftDroid to further enhance the crash exploration. Note that, in our work, we use the same test inputs to feed different lineage app versions in order to locate potential crash *fixes*.

Works that tackle the compatibility of Android apps [25, 26, 45] are also relevant to our study. Indeed, there are many reasons that may cause Android apps to be incompatible to some devices. However, the resulting consequence of incompatibilities is often the same: runtime crashes.

Android App Testing and Analysis. Several automated app testing approaches have been proposed to dynamically analyze Android apps [2, 3, 5, 8, 10, 16, 36, 37, 48]. For example, Mao et al. [37] have introduced an approach that combines random fuzzing, systematic and search-based exploration, exploiting seeding and multi-level instrumentation techniques to perform multi-objective automated testing for Android apps. Among the top 1,000 Google Play apps, Sapienz is able to find 558 unique, previously unknown crashes. As summarized by Kong et al. [17] in their recent systematic literature review, there are more than 100 works proposed by the community to tackle the problem of automated app testing. Our approach is orthogonal to these approaches, where we can leverage more automated testing approaches to further pinpoint runtime crashes of Android apps.

Static analysis has been also a popular technique to dissect Android apps [23, 24, 38, 41]. For example, researchers have used static taint analysis to discovery privacy leaks in Android apps [4] and leveraged model checking techniques to verify Android apps in terms of their security properties [6]. In this work, we leverage basic static analysis techniques to identify crash fixes in Android apps. We believe that more advanced static analysis techniques could be leveraged to improve the accuracy of our crash fix identification approach.

8 CONCLUSIONS AND FUTURE WORK

In this work, we target a new research direction attempting to mine Android crash fixes from Android market apps, which usually do not have an open change-tracking or issue-tracking system. We successfully generatred 17 fine-grained fix templates, which were evaluated to be effective in patching 17 out of 20 real-world crashed apps. CraftDroid can benefit the automatic program repair (APR) community [18, 20, 29–35] in establishing a new means to augment datasets. CraftDroid can also benefit the developer community in recommending effective patches to fix their crashed apps.

As future work, we plan to integrate more automated testing tools so as to enrich our set of fix templates. We also plan to implement analyzers to 1) scan user reviews to retrieve a more accurate set of apps to be tested for runtime crashes and 2) study the GUI changes to filter out false negatives. Finally, beyond the current manual abstraction methods, we aim to apply state-of-the-art automatic fix mining approaches to make our approach fully automatic.

ACKNOWLEDGMENTS

This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under CORE projects CHARACTERIZE C17/IS/11693861 and RECOMMEND 15/IS/10449467, and AFR-PhD project 11620657.

REFERENCES

[1] K. Allix, T. F. Bissyande, J. Klein, and Y. L. Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th IEEE/ACM Working Conference on Mining Software Repositories (MSR)*. ACM, 468–471.

[2] 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.

[3] 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, 1–11.

[4] 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.

[5] 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.

[6] Guangdong Bai, Quanqi Ye, Yongzheng Wu, Heila Botha, Jun Sun, Yang Liu, Jin Song Dong, and Willem Visser. 2018. Towards model checking android applications. *IEEE Transactions on Software Engineering* 44, 6 (2018), 595–612.

[7] Ning Chen and Sunghun Kim. 2015. Star: Stack trace based automatic crash reproduction via symbolic execution. *IEEE transactions on software engineering* 41, 2 (2015), 198–220.

[8] 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.

[9] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *Proceedings of the 2015 IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 341–350.

[10] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. 2018. FraudDroid: Automated Ad Fraud Detection for Android Apps. In *The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*.

[11] F-Droid. 2019. Open Source App Repository. https://f-droid.org/en/. Accessed: 2019-05-31.

[12] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-scale analysis of framework-specific exceptions in Android apps. In *Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 408–419.

[13] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. 2018. On Vulnerability Evolution in Android Apps. In *Proceedings of The 40th International Conference on Software Engineering, Poster Track (ICSE 2018)*.

[14] Google. 2018. Supported Devices -Google Play Help. https://support.google.com/googleplay/answer/1727131?hl=en-GB. Accessed: 2018-08-24.

[15] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 121–130.

[16] 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*. ACM, 67–77.

[17] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. 2018. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Transactions on Reliability* (2018).

[18] Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2017. Impact of tool support in patch construction. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 237–248.

[19] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2018. Fixminer: Mining relevant fix patterns for automated program repair. *arXiv preprint arXiv:1810.01791* (2018).

[20] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: Bug Report driven Program Repair. In *Proceedings of the 27the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM.

[21] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*.

[22] Li Li, Tegawendé F Bissyandé, and Jacques Klein. 2017. SimiDroid: Identifying and Explaining Similarities in Android Apps. In *Proceedings of the 16th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-17)*. IEEE, 136–143.

[23] Li Li, Tegawendé F Bissyandé, and Jacques Klein. 2019. Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark. *IEEE Transactions on Software Engineering (TSE)* (2019).

[24] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. 2017. Static Analysis of Android Apps: A Systematic Literature Review. *Information and Software Technology* (2017).

[25] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. CiD: Automating the Detection of API-related Compatibility Issues in Android Apps. In *Proceedings of The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*.

[26] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2018. Characterising Deprecated Android APIs. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR 2018)*.

[27] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. 2017. Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting. *IEEE Transactions on Information Forensics & Security (TIFS)* (2017).

[28] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 23–26.

[29] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Anil Koyuncu, Kisub Kim, Taeyoung Kim, Suntae Kim, and Yves Le Traon. 2019. Learning to Spot and Refactor Inconsistent Method Names. In *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering*. IEEE.

[30] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Shin Yoo, and Yves Le Traon. 2018. Mining Fix Patterns for FindBugs Violations. *IEEE Transactions on Software Engineering* (2018).

[31] Kui Liu, Dongsun Kim, Anil Koyuncu, Li Li, Tegawendé F Bissyandé, and Yves Le Traon. 2018. A Closer Look at Real-World Patches. In *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution*. IEEE, 275–286.

[32] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *Proceedings of the 12th IEEE International Conference on Software Testing, Verification and Validation*. IEEE.

[33] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. [n. d.]. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*. 456âĂŞ467.

[34] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM.

[35] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F. Bissyandé. 2018. LSRepair: Live Search of Fix Ingredients for Automated Program Repair. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference*. 658–662.

[36] 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*. ACM, 224–234.

[37] 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*. ACM, 94–105.

[38] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. 2017. A survey of app store analysis for software engineering. *IEEE transactions on software engineering* 43, 9 (2017), 817–847.

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

[40] Ward Muylaert and Coen De Roover. 2018. Untangling Composite Commits Using Program Slicing. In *Proceedings of 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 193–202.

[41] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. 2017. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering* 43, 6 (2017), 492–530.

[42] 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.

[43] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing crashes in android apps. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 187–198.

[44] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. IBM Corp., 214–224.

[45] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 226–237.

[46] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. 2014. CrashLocator: locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 204–214.

[47] 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.

[48] 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.