Generating Reproducible and Replayable Bug Reports from Android Application Crashes

Martin White, Mario Linares-Vásquez, Peter Johnson, Carlos Bernal-Cárdenas, and Denys Poshyvanyk
Department of Computer Science
College of William and Mary
Williamsburg, Virginia 23187–8795
Email: {mgwhite, mlinarev, pj, cebernal, denys}@cs.wm.edu

***Abstract*—Manually reproducing bugs is time-consuming and tedious. Software maintainers routinely try to reproduce unconfirmed issues using incomplete or noninformative bug reports. Consequently, while reproducing an issue, the maintainer must augment the report with information—such as a reliable sequence of descriptive steps to reproduce the bug—to aid developers with diagnosing the issue. This process encumbers issue resolution from the time the bug is entered in the issue tracking system until it is reproduced. This paper presents CRASHDROID, an approach for automating the process of reproducing a bug by translating the call stack from a crash report into expressive steps to reproduce the bug and a kernel event trace that can be replayed on-demand. CRASHDROID manages traceability links between scenarios' natural language descriptions, method call traces, and kernel event traces. We evaluated CRASHDROID on several open-source Android applications infected with errors. Given call stacks from crash reports, CRASHDROID was able to generate expressive steps to reproduce the bugs and automatically replay the crashes. Moreover, users were able to confirm the crashes faster with CRASHDROID than manually reproducing the bugs or using a stress-testing tool.**

***Keywords*—Android, crash and bug reports, reproducibility**

I. INTRODUCTION

The burgeoning mobile device and application (app) market is fueled by performant hardware and evolving platforms, which support increasingly complex functionality. Many of the mobile apps have practically the same features as their desktop variants. Furthermore, mobile platforms such as Android and iOS enable user interaction via a touchscreen and a diverse set of sensors (e.g., accelerometer, ambient temperature, gyroscope, and light) that present new challenges for software testing and maintenance. Despite these new platform-specific challenges, the development and maintenance processes for desktop and mobile apps are quite similar with a few notable exceptions [1]–[4]. Nonetheless, high-quality bug reports are equally important for maintaining desktop and mobile apps [5]–[7]. Ideally, bug reports should contain a detailed description of a failure and occasionally hint at the location of the fault in the code in the form of patches or stack traces [8], [9]. However, the quality of bug reports (and user reviews) for Android apps in the wild varies considerably as they are often incomplete or noninformative [10]. Finding and fixing issues in modern mobile apps is difficult; verifying the solution, without being able to reproduce the bug using the bug report, compounds this complexity.

The adverse impact of noninformative bug reports on maintaining mobile apps is aggravated when the issue concerns bugs that require gestures (e.g., swiping, zooming, and pinching the touchscreen) or interactions that use sensors. Equivalently, precise locations of the gesture and temporal parameters may be required to reproduce the bug. Gestures, sensors, precise spatial coordinates, and interarrival times between events/gestures are only some of the challenges that may prevent a maintainer from reproducing an issue even if she is given a report with a complete sequence of steps to reproduce the bug. For example, it would be challenging to unambiguously describe, in natural language terms, a sequence of steps to reproduce a bug in Google Maps that requires pinching and zooming the touchscreen and simultaneously uses input from the sensors to render the map. Thus, it is vital to provide replayable scripts that can precisely demonstrate how to reproduce a bug in addition to complete descriptions. These replayable scripts are critical for cases where issues may be difficult to manually reproduce because of particular spatial or temporal requirements like exceedingly fast interarrival times to trigger a fault when stress-testing a network interface. Recently, several tools for systematic exploration [11]–[20] and record and replay [21] have been proposed, but none of the existing approaches provides a complete solution for automatically producing natural language bug reports with replayable scripts for an incoming crash report.

In this paper, we present CRASHDROID, a novel approach to automatically generating reproducible bug reports given the call stack from an incoming crash report, and these bug reports can be replayed on-demand to demonstrate the crash. Our solution relies on the fact that stakeholders of the application under test (AUT) can compose application usage scenarios in natural language terms that use different features of the app. Afterward, kernel event traces (i.e., live dumps of kernel input events) of the scenarios are recorded while executing those scenarios, with minimal overhead (approximately 1%) using an engine based in part on RERAN [21]. Thus, recording these scenarios is no different from typical usage. Given a dataset of natural language descriptions and kernel event traces, CRASHDROID translates these traces into higher-order user actions (e.g., click, long click, and swipe) and establishes traceability links between each sentence in the description and the corresponding high-order user action. CRASHDROID replays each low-level event trace several times on a physical test device (a Google Nexus 7 tablet) and collects precise execution profiles.[1] Subsequently, the user actions are aligned with sequences of methods in execution profiles using time stamps, indirectly enabling traceability between sentences in

[1]By *execution profile*, we mean the sequence of calls to APIs and app-specific methods during the execution of a scenario.

the descriptions and method sequences in execution profiles. Afterward, given the call stack from an Android app crash report, CRASHDROID queries the existing database of execution profiles to identify matching calling contexts and ranks candidate profiles using the longest common subsequence (LCS) algorithm [22]. If a candidate profile is identified, CRASHDROID uses traceability links between profiles and scenario descriptions to generate a bug report using sentences from the descriptions. Next, CRASHDROID validates each candidate profile by running a representative event trace on the AUT to check whether the AUT's Application PacKage (APK) file crashes in that context. After the crash is automatically reproduced and confirmed, CRASHDROID returns the generated bug report, which includes natural language steps to reproduce the bug with the crashing stack trace and a replayable script. We evaluated CRASHDROID with five open-source Android apps in a study designed to measure the effectiveness and expressiveness of automatically generated bug reports. We found that CRASHDROID compared favorably to both human-written reports and other automated testing tools in terms of both time and the ability to crash an app.

II. Related Work

CRASHDROID is the only approach, to the best of our knowledge, that is able to automatically generate natural language bug reports and reproduce/replay crashes in Android apps using only a call stack and an APK. However, CRASHDROID's record and replay mechanisms are related to test case generation approaches, keyword-action approaches, and event-driven approaches.

A. Approaches for Crashing and Re-crashing Software

It should be noted that all the techniques in this subsection are not yet available for the Android platform; however, we discuss them here since they are similar to our work in terms of their goal, i.e., detecting and reproducing crashes.

One related technique to CRASHDROID is ReCrash [23], an approach for reproducing software crashes by monitoring and storing partial copies of method arguments in memory during a program's execution and then using this information to create test cases designed to reproduce the failure after observing crashes. While CRASHDROID and ReCrash have the same goal of reproducing crashes, they achieve it in two different ways. ReCrash is a dynamic monitoring approach that was designed for Java (ReCrashJ [23]), whereas CRASHDROID is an approach designed to deal with complex crashes pertinent to the Android platform. CRASHDROID can handle reproducing crashes involving gestures, e.g., click, long click, swipe, and type. CRASHDROID also supports recording and replaying that ReCrash was not designed to support.

Another set of approaches is represented by JCrasher [24], Eclat [25], DSDCrasher [26], and Randoop [27]. These are test-case generation techniques, which exploit random inputs to find crashes. Compared to CRASHDROID, these techniques generate test cases that are not necessarily representative of actual app usage, whereas CRASHDROID is designed to reproduce real crashes via actual program executions.

Crash reporting systems send stack traces or core dumps back to developers [28]–[32]. The idea behind these approaches is similar to an idea behind CRASHDROID insofar as stack traces are used as input to identify the problem. ReBucket [32] is designed to cluster crash reports by measuring call stack similarity to aid debugging and facilitate diagnosis. However, CRASHDROID is solving a problem these aforementioned approaches are not able to solve—reproducing crashes with minimal developer effort by generating both bug reports and replayable scripts.

The next important step of debugging a crash report is to identify faulty functions, which is usually a challenging and labor-intensive task. A number of techniques for fault-localization based on crash reports have been proposed [33]–[43]. However, a detailed comparison and discussion of these approaches is beyond the scope of this paper since our work is concerned with generating reproducible and replayable crash reports to be used as a reliable starting point for developers rather than identifying exact root causes of crashes.

B. Approaches for GUI-based Testing of Android Apps

The closest work to ours is RERAN [21] since CRASHDROID uses RERAN as an underlying engine. The key difference between CRASHDROID and RERAN is the fact that CRASHDROID not only records/replays kernel event traces but also captures execution profiles and links them to natural language descriptions and the steps in those descriptions. CRASHDROID uses an incoming call stack as an input and matches it against a set of existing profile traces. If it finds a match for a given calling context, CRASHDROID generates a bug report with expressive steps to reproduce and a replay script. Moreover, before presenting the bug report to the user, CRASHDROID validates it by replaying the script and checking the validated crash context against the incoming call stack.

Keyword-action tools for the Android platform include MonkeyRunner [13], Robotium [15], and GUITAR [16]. Notably, keyword-action tools depend on the availability of the app's layout to identify GUI components and handle IDs. The key difference between our approach, which relies on RERAN, and these tools is that CRASHDROID is agnostic to the layout and availability of user-defined GUI elements. Additionally, keyword-action tools do not support non-trivial touchscreen gestures nor other input sensors. In our experience with using MonkeyRunner, it is quite tedious to design scripts that include long touches, and the support for swiping, zooming, and pinching is limited. Another widely used tool for testing Android apps is the Android GUI Monkey [14]. With Monkey, developers/testers generate random actions (e.g., touches, swipes, and other system events) inside the app.

Another related approach is Dynodroid [12], an event-driven technique that enables Android app testing by observing not only GUI-based events but also system events. The main idea behind Dynodroid is explained by an observe-select-execute cycle [12] in which Dynodroid first observes which events are related to the app before selecting relevant ones and executing them. While the GUI-based aspect has been captured in other tools, Dynodroid extends it by handling system events such as an incoming message or a low battery [12]. But Dynodroid does not handle multi-touch gestures nor input from a wide range of sensors, which are vital for reproducing/replaying intricate bugs pertinent to mobile apps.

Finally, examples of approaches and tools relying on systematic GUI-based testing strategies are the work by Takala et al. [44], and the tools VanarSena [45], AndroidRipper [46], ORBIT [19], A^{3}E [47], and SwiftHand [18]. Examples of approaches aimed at GUI-based testing using event sequences are Collider [48], EvoDroid [17], and MobiGUITAR [49]. However, unlike CRASHDROID, none of these tools/approaches are aimed at automtically generating natural language bug reports and replaying crashes in Android apps.

III. System Architecture

CRASHDROID relies on two phases, an offline workflow and an online dataflow, that provide a database of artifacts and traceability links between scenarios' descriptions and kernel input events (workflow) and reproducible/replayable bug reports for target crashes (dataflow). Fig. 1 depicts the CRASHDROID architecture. The first part (Fig. 1-a) describes the offline workflow where CRASHDROID links scenarios' natural language descriptions, replay scripts, and app execution profiles. The offline workflow is a precondition for the online dataflow. The second part (Fig. 1-b) describes the online dataflow, where CRASHDROID automatically transforms call stacks from incoming crash reports into sets of expressive bug reports and replay scripts for a software maintainer.

A. Offline CrashDroid Workflow

Alg. 1 specifies the offline CRASHDROID workflow, which receives a set of scenario descriptions D representing app usages provided by the stakeholders and produces parallel corpora comprising replay scripts S and app execution profiles P. The workflow's guiding design principle is to manage transparent traceability links among narratives in descriptions, event streams, and profiles.

Algorithm 1: Offline CRASHDROID Workflow

```
   Input: $D$
   Output: $S, P$
1  begin
2      $S = P = \emptyset$;
3      foreach $d \in D$ do                          /* collect */
4          repeat
5              $g \leftarrow$ record getevent log for $d$;
6          until $num\_sentences(d) == num\_actions(g)$;
7          $s \leftarrow$ translate $g$;
8          $S \leftarrow S \cup s$;
9      foreach $s \in S$ do                          /* capture */
10         $p \leftarrow$ profile $s$;
11         $P \leftarrow P \cup p$;
12     foreach $p \in P$ do                          /* munge */
13         filter Dalvik virtual machine calls from $p$;
14         serialize the method calls in $p$;
15         filter unique elements of $p$;
```

1) Collecting Natural Language Descriptions: The first stage in the offline phase is to collect descriptions of well-formed natural language sentences for app features from developers, testers, beta-users, or even ordinary users (Fig. 1 as (1)). There is no requirement on expertise to build a description. These descriptions should simply specify scenarios that use one or more features, but CRASHDROID places a few practical constraints on the content. For instance, the scenario is required to start from the app's main activity. Additionally, each sentence in the description is required to contain a subject and a verb and may contain components as well as any number of adjectives, adverbs, prepositional phrases, and participial phrases describing the verb or component. The subject may simply be the first-person singular pronoun "I." The verb is the action, which is limited to "touched," "longtouched," "swiped," and "typed." Currently, CRASHDROID does not support the gestures "zoomed" and "pinched." However, it is possible to add these actions to CRASHDROID since the system is based in part on RERAN, which supports recording and replaying these gestures, and CRASHDROID can use the kernel's multi-touch protocol to parse the event traces into actions.

The *swiped* action is coupled with a direction, either "up," "down," "left," or "right," and the *typed* action is coupled with a string literal constrained to one or more lowercase letters, digits, or spaces. The *typed* action is essentially a macro that expands one sentence in the description into a sequence of sentences where the number of sentences in the macro expansion is equal to the number of characters in the string literal, but the sentence that precedes the *typed* action must describe how the user brought focus to the component. For example, consider the following description: *I touched the "Search" text field at the top of the touchscreen. I typed "crash droid" in the "Search" text field. I touched the "CrashDroid" list item.* The first sentence (i.e., *I touched... touchscreen.*) is necessary because it describes how the user brought focus to the text field. The second sentence (i.e., *I typed... text field.*) would be expanded into a sequence of several (11 in the example) sentences, one sentence for each character in the string literal "crash droid" since the user enters input by touching keys on the keypad and every touch is a user action.

CRASHDROID limits descriptions to the following set of components: list item, grid item, button, text field, checkbox, radio button, toggle button, spinner, picker, seek bar, and image. Each component in this list is a valid View object in the Android programming model [50]. The adjectives, adverbs, prepositional phrases, and participial phrases qualify verbs or components, e.g., the location on the touchscreen where the action was performed, the color of the component, or any text on the component. Finally, the descriptions are stored in the CRASHDROID database.

2) Compiling Replay Scripts for Descriptions: The next stage in the offline phase, represented in Alg. 1 lines 3–8 and Fig. 1 as (2), is to generate a replay script for a description. The user passes the description as a command line argument to CRASHDROID which begins recording kernel input events as the stakeholder performs, in a mobile device, steps from the description. CRASHDROID uses the Getevent tool [51], like RERAN, to read device files and write a log of streaming events on the device. RERAN is a tool that facilitates recording and replaying Android hardware events. It replays the events with timing that is faithful to the recording, so an event trace is virtually indistinguishable from a real user interacting with the device—from the kernel's perspective. We forked the project and implemented several necessary updates [52]. Specifically,

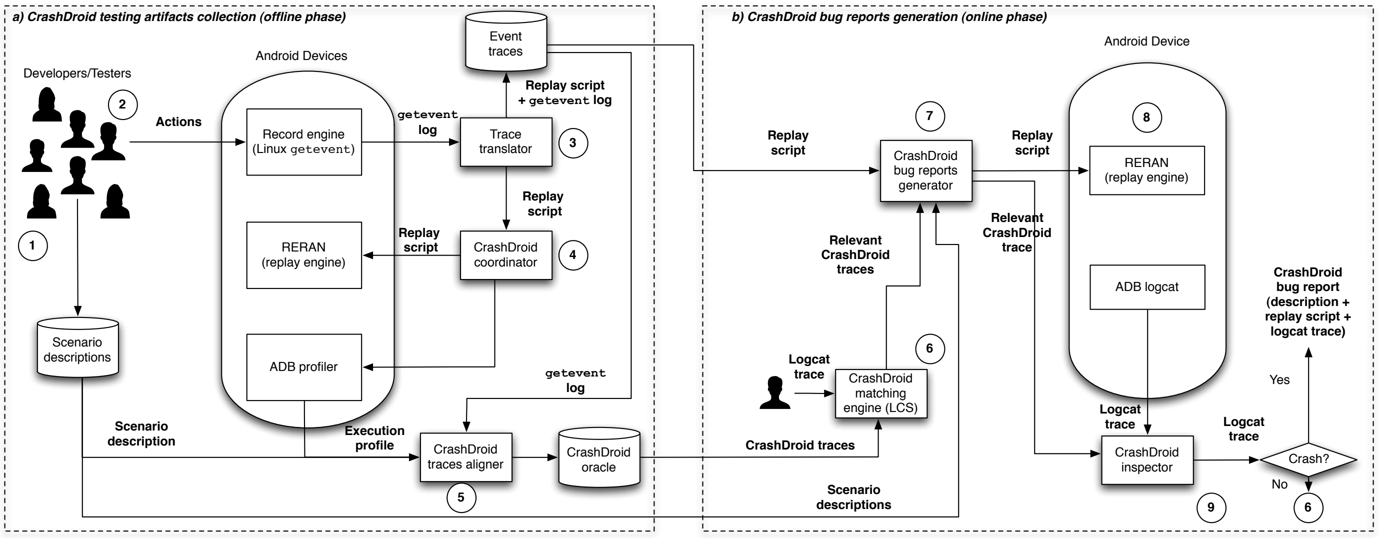

Fig. 1. CRASHDROID comprises an offline workflow and an online dataflow. In the workflow, developers, testers, or users build scenario **descriptions** and record replay **scripts** for the descriptions. CRASHDROID consumes the replay scripts and produces execution **profiles** for the scripts. In the dataflow, given a crash report, CRASHDROID will compute similar profiles using the LCS algorithm. Given these "similar" profiles, CRASHDROID will query the corresponding descriptions and replay scripts to automatically generate bug reports. Finally, CRASHDROID will automatically verify its list of candidate bug reports (sorted by similarity) to verify that its suggestions crash the app, before presenting its reproducible/replayable bug reports to the user.

we fixed a bug that prevented longer event traces from being replayed, and we updated the tool to be compatible with Android APIs newer than Gingerbread.

Kernel input events are represented as a 5-tuple with the following fields: time, device, type, code, and value. When users interact with an app, hardware components write events to component-specific device files, which serve as an interface to the kernel. RERAN records these event streams stemming from hardware components using the Getevent tool via an Android Debug Bridge (adb) shell [53]. These logs are translated into an intermediate representation to be interpreted by a replay agent running on the device in its own process. Essentially, the replay agent writes the recorded events to device files with microsecond accuracy. In addition to the replay agent's highly precise timing, another one of its salient features is "time warping." The replay agent enables *fast forwarding* execution by decreasing the time delays between consecutive events.

When the stakeholder has exhausted all the steps in the description, she signals CRASHDROID to indicate the recording is finished. The system's signal handler pipes the Getevent log to a CRASHDROID utility that translates the entire event stream into a sequence of user actions, e.g., touches, long touches, and swipes. Then CRASHDROID validates the number of actions in the Getevent log equals the number of sentences in the description: Each sentence maps to one and only one action. Thus, CRASHDROID can represent a natural language sentence as a kernel event stream, or it can describe sequences of events as natural language sentences. If the number of kernel events equals the sentences in the description, CRASHDROID uses a compiler, packaged with the RERAN archive, to translate the Getevent log into a replay script that can be replayed on the device via the RERAN replay interpreter. CRASHDROID tags this replay script to maintain a traceability link with the natural language description and saves the file to disk (Fig. 1 as (3)). Incidentally, if the number of sentences is not equal to the number of events, then CRASHDROID notifies the user. The user can choose to record a new Getevent log, or CRASHDROID can run the replay script, so the user can observe discrepancies between the recorded scenario and the sentences in the description.

3) Capturing App Execution Profiles of Replay Scripts: The Android SDK includes a profiler for capturing execution traces, and these traces contain information about the execution time for each method, including methods in AUT classes and API calls, methods' nesting levels, etc. Linares-Vásquez et al. [54] used the profiler to capture execution traces while measuring the energy consumption of Android apps. CRASHDROID uses it to profile a running replay script and log scenarios' dynamic method invocations (Alg. 1 lines 9–11). First, CRASHDROID pushes the replay script from the host to the device using adb [53] (Fig. 1 as (4)). CRASHDROID uses adb am [53], the adb activity manager utility, to launch the AUT on the device and start the profiler. With the profiler running in the background, CRASHDROID uses the replay interpreter on the device to run the replay script automatically while the profiler logs the execution profile. When the scenario finishes, CRASHDROID stops the profiler, closes the AUT, pulls the execution profile from the device, and pipes it to dmtracedump [55], another Android SDK tool, to translate the binary format of the execution profile into human readable format. Then CRASHDROID queries the event traces database for the corresponding Getevent log and translates the kernel input events into actions as it did when it validated the log. However, this time it notes the duration of each action. CRASHDROID takes this sequence of actions with their durations and superimposes this "paned window" on the execution profile to cluster the calls in the profile. This enables CRASHDROID to label every time slice in the profile with the corresponding action in the Getevent log and—transitively—a sentence in the description. Thus, CRASHDROID specifies a traceability link between sentences in the description and a set of complementary subsequences of method calls from the execution profile. This alignment process is represented in Fig. 1 as (5). Finally, CRASHDROID tags the profile to maintain traceability with the replay script and the corresponding description.

4) Munging App Execution Profiles: Ultimately, CRASHDROID is designed to take information (e.g., the call stack in a crash report) and automatically produce natural language descriptions and replay scripts for a software maintainer. To this end, the last stage in the offline workflow (Alg. 1 lines 12–15) is to scrub the profiles in the CRASHDROID database to reconcile the lexemes in the execution profiles with the expected format of incoming call stacks. CRASHDROID filters the API calls from the execution profile to produce a streamlined log of AUT class method calls. For instance, for a generic calculator app with main package `com.android.calculator`, CRASHDROID would select only the method calls in the `com.android.calculator` namespace.

The time resolution of the CRASHDROID profiling mechanism is limited to milliseconds, which is relatively course-grained. Consequently, there may be multiple methods executed in a time slice, which necessitates transforming the profile into a proper sequence of method calls. In cases where there are multiple calls in a time slice, CRASHDROID uses a sliding window (three time slices wide) to attempt to reliably sequence the calls. Suppose (x_1, x_2, x_3) is a 3-tuple indicating the number of calls in each time slice in a particular window. $(1,1,1)$ windows are literal translations, since the profiler logged one and only one call in each time slice. For $(1,2,1)$ and $(1,3,1)$ windows, where the window straddles a time slice with more than one call, CRASHDROID uses the endpoints, i.e., the calls in the first and third time slices, to serialize the multiple calls in the middle. For example, for $(1,2,1)$ windows, each call in the second time slice will be matched to an endpoint (if possible) to determine their serial order. Additionally, there may be more than three methods in a time slice or multiple methods in consecutive time slices. CRASHDROID randomly sequences the methods in these rare cases: This heuristic would not be needed if the profiler could log methods with a higher sampling rate. Care was taken to build a highly modular design, so the profiling mechanism can easily be replaced without impacting any of the other components in the architecture. Finally, CRASHDROID produces a sanitized execution profile with unique methods.

B. Online CrashDroid Dataflow

Alg. 2 specifies the online CRASHDROID dataflow, an online process for taking the call stack σ from an incoming crash report (and an optional threshold τ) and producing a set B of natural language descriptions for reproducing the bug as well as a set R of replay scripts to automatically replay the bug. The online dataflow requires a database of scenario descriptions, event logs, and execution profiles, all derived from the offline workflow.

First, CRASHDROID filters the API calls from the incoming call stack and extracts the top m methods from this filtered call stack. Then CRASHDROID queries the profiles in its database that contain the most recent AUT class method pushed onto the crash call stack, i.e., peek(σ'), and at least n relevant methods; this collection of profiles is denoted by P' in Alg. 2. For each

Algorithm 2: Online CRASHDROID Dataflow

```
   Input: $D, S, P, \sigma, n, m, \tau$
   Output: $B, R$
1  begin
2      $B = R = T = \emptyset$;
3      filter API calls from $\sigma$;
4      $\sigma' \leftarrow$ extract top $m$ methods from $\sigma$;
5      $P' \leftarrow \{\pi \in P : \text{peek}(\sigma') \in \{\pi\} \wedge |\{\pi\} \cap \{\sigma'\}| \geq n\}$;
6      foreach $\pi \in P'$ do
7          $\rho \leftarrow \text{len}(\text{LCS}(\pi, \sigma')) \div \text{span}(\text{LCS}(\pi, \sigma'))$;
8          if $\rho \geq \tau$ then
9              $T \leftarrow T \cup \pi$;
10     $B, R \leftarrow$ query $d \in D, s \in S$ linked to $\pi \in T$;
11     verify $B, R$;
```

profile that meets these criteria, CRASHDROID computes the length of the LCS containing relevant methods and normalizes the length by dividing by span$(\text{LCS}(\pi, \sigma'))$, which is the number of calls in the profile between the first and last method of the LCS (Fig. 1 as (6)). Naturally, there may be multiple "longest" subsequences. In this case, CRASHDROID optimistically chooses the ones with the shortest span to score the profile. While measuring the similarity of profiles to the call stack, CRASHDROID accumulates the best candidate profiles in a collection T. "Best" depends on an empirically-tuned threshold τ on the profiles' scores that is tuned to optimize the trade-off between the true positive rate and the false positive rate; each description either reproduces a bug or it does not reproduce a bug. The scoring function—like the profiling mechanism—we use to rank profiles is decoupled from the rest of the system, so it is trivial to swap in a new scoring engine for measuring the similarity between information in crash reports and artifacts in the database.

Since CRASHDROID manages traceability links between profiles and descriptions, CRASHDROID can use the ranked list of profiles to build a ranked list of descriptions, which serve as *recommended descriptions* to reproduce the issue. CRASHDROID traverses the list of recommended descriptions (Fig. 1 as (7)), running each replay script automatically (Fig. 1 as (8)) to verify the replay script generates a crash. The resulting call stack is compared to the given call stack σ to confirm the particular crash. CRASHDROID filters the recommendations that do not reproduce the bug before presenting the recommendations to the user (Fig. 1 as (9)). Again, one of the replay agent's salient features is its ability to fast-forward execution. When CRASHDROID automatically replays the script, it can exploit this feature by rushing execution to the bug to confirm the issue. The sentences from the descriptions linked to the events in the script, the replay script, and the stack trace are presented as a reproducible/replayable bug report.

IV. Empirical Study Design

The goal of the study was to evaluate the effectiveness and expressiveness of bug reports generated by CRASHDROID with the purpose of providing better tools and automated techniques to aid software maintenance and evolution. The context consisted of Android devices, six open-source Android apps, natural language descriptions defined by study participants,

TABLE I. STATISTICS ON THE AUTS FOR THE USER STUDY

AUT	Version	Category	LOC	# Classes	# Methods	User Actions	# Scenarios	# Contributors
Calculator	3.4.2	Office	3,940	79	427	touch, swipe	36	7
Frex	1.2.2	Science and Education	2,544	105	426	touch, swipe	11	5
GameMaster Dice	0.1.5	Games	598	19	85	touch, longtouch, swipe	14	6
Simple Deadlines	3.1.2	Productivity	985	36	84	touch, longtouch, swipe	15	3
Stickeroid	1.1.2	Productivity	841	13	82	touch, longtouch, swipe	12	2
Units Converter	1.0.0	Tools	3,793	66	419	touch, swipe	14	3

and Getevent logs from executing the scenarios. We used four unlocked and rooted Google Nexus 7 tablets, each equipped with a quad-core Qualcomm Snapdragon S4 Pro processor running Android 4.4.2 (kernel version 3.4.0.gac9222c). We chose the Android platform for two notable reasons. First, the open-source AUTs can be freely downloaded enabling the reproducibility of our experiments. Second, Android SDK tools enable the remote execution of apps from a host connected to a device for debugging, replaying event traces, and profiling. The quality focus of the study concerned the generation of natural language descriptions and replay scripts for reproducing/replaying bugs.

A. Research Questions

Our study sought to empirically determine whether bug reports generated by CRASHDROID are effective and expressive for reproducing crashes in Android apps. We examined the following research questions:

RQ1 Is CrashDroid an effective approach for identifying scenarios that are relevant to a sequence of methods describing crashes in Android apps?

RQ2 How do CrashDroid bug reports compare to human-written bug reports in terms of readability, conciseness, and reproducibility?

The **independent variable** for the research questions was BugID, representing bugs manually injected into the apps' source code to crash the apps. RQ2 also considered whether the bug report was human-written or generated by CRASHDROID. The **dependent variables** for RQ1 were the total number of scenarios replayed to reach a bug and the total time to reproduce a bug. The **dependent variables** for RQ2 were the time it took participants to reproduce crashes, when using human-written reports and CRASHDROID reports, and the readability, conciseness, and reproducibility of the reports. Readability, conciseness, and reproducibility were each measured with a five-point Likert item.

B. Methodology and Data Collection

The AUTs, listed in Tab. I, were downloaded from the F-Droid market [56]. We selected these apps based on several specific criteria. Notably, we sought apps that did not require authentication and used several different types of View objects [57]. To simulate the offline CRASHDROID workflow, we enlisted eight graduate students from the College of William and Mary. First, we asked them to become familiar with the AUTs on an emulator or device. We wanted the students to use the AUTs and transcribe their scenario descriptions offline, so their event traces for the study were "natural" rather than introducing unnatural delays to record their actions. We provided participants with some guidance on composing descriptions to build a well-formed corpus (Sec. III-A1). After gaining some familiarity with the AUTs, we provided them with a pre-configured Google Nexus 7 tablet. They were instructed to set the device on a flat surface (to minimize input from auxiliary sensors) and start recording their scenarios from the AUT's main activity. They were required to perform each scenario at least three times on the devices, and we also asked the participants to clear the app's cache/data before executing each scenario. In response, participants provided us with a set of descriptions, each coupled to multiple Getevent logs. Each archive submitted by a participant was thoroughly validated before it was pushed to the CRASHDROID repository, a database of descriptions and Getevent logs, used to obtain the execution profiles of replay scripts.

Afterward, one of the authors injected five crashes into each one of the AUTs and built one "tainted" APK for each bug. By design, this author was neither involved in collecting descriptions nor compiling replay scripts. Tab. II lists the bugs injected into the AUTs. The crashes were introduced using the statement `throw new AssertionError(...)`. We used errors instead of exceptions to avoid the handling process by try-catch blocks. Thus, the errors can be seen by the tester/developer, and the applications stop when the errors are thrown. Then, for each crash, we collected the logcat traces and executed the CRASHDROID online phase, using the logcat traces and the tainted APKs to generate a list of bug reports to be validated.

In order to collect human-written bug reports (RQ2) for the injected crashes, we asked two graduate students (outside the College of William and Mary) with experience as real developers, to generate bug reports for 15 of the injected bugs from Tab. II. We provided them with only logcat traces representative of the crashes and the apps' tainted source code exhibiting the crash (one different version per crash). The two participants were never involved in the data collection for the CRASHDROID offline phase, and never read the guidelines for the natural language descriptions (Sec. III-A1). This design decision was based on the fact that we wanted to have bug reports as close as possible to descriptions in the wild. We also asked them to measure the time it took them to reproduce a crash and write steps to reproduce each bug.

C. Validating CrashDroid

We designed two experiments to answer the research questions. In the first experiment, we measured effectiveness (RQ1) by analyzing the amount of time to recommend a natural language description with a replay script and the number of scenarios replayed by CRASHDROID. CRASHDROID uses a heuristic designed to tease the bug producing replay scripts from the false alarms in a particular context. This rule-of-thumb, the normalized length of the LCS, scores the candidate bug reports in the document database, so the bug producers rise to the top of the list. However, it is possible for multiple reports to have the same similarity score according to the heuristic, yet the replay scripts for these reports may have different execution times. For instance, if there are five reports with a score of one, then CRASHDROID can place any one of these five reports at the top of the ranked list of candidates, yet the replay scripts' execution times for each of these reports may be very different. To reliably measure the amount of time to recommend a natural language description with a replay script using the ranked list of profiles, we designed a Monte Carlo simulation to randomly sort the subsequences of the ranked list where there were profiles with the same score. While CRASHDROID is configured to examine each and every scenario in its document database, the purpose here is to reduce the amount of time it takes CRASHDROID to sweep down its internal list of candidate scenarios verifying its proposals before presenting a final list of recommendations to the user. This amounts to optimizing the trade-off between the sensitivity and the specificity. *However, from the user's perspective, the precision is always 100%. CrashDroid will never recommend a scenario that it did not verify.*

For the first experiment, we also compared the effectiveness of CRASHDROID to Monkey in terms of time and the number of trials and events required to crash each app version. Monkey was used for validation in previous studies [12]. In this study, it served as a baseline. Monkey generates pseudo-random user and system events according to a set of user-defined arguments. In our case, we executed Monkey with 1,000 events, interleaving delays of 10 milliseconds. This design decision was aimed at validating whether CRASHDROID's effectiveness outperformed a random sequence of events.

Our second experiment was designed to measure the expressiveness (RQ2) of bug reports in terms of readability, conciseness, and reproducibility. We conducted a user study involving ten computer science graduate students from the College of William and Mary to simulate an issue reproduction process, using bug reports generated by CRASHDROID and bug reports written by humans. None of the students in this study were involved in the offline workflow to build the descriptions nor were they familiar with CRASHDROID. Each participant received a package of 15 bug reports, including CRASHDROID reports and human-written reports for 15 different crashes. We provided each participant with different bug reports in such a way that the CRASHDROID bug report and the two human-written bug reports for a particular crash were evaluated by (at least) one different participant. Specifically, we implemented a simple linear program to maximize redundancy in the assignments of participants to bug instances.

We provided anonymized bug reports, so the participants did not know whether they belonged to the CRASHDROID set or the human-written set, with call stacks and a device with the corresponding apps. Then we asked participants to measure how much time it took them to reproduce bugs using a particular bug report. The participants also rated the readability, conciseness, and reproducibility of the reports using a five-point Likert item. Consequently, for RQ2 we determined whether there was a statistically significant difference between the two sets of results, for each measure, using the Mann-Whitney U test [58]. In every test, we looked for statistical significance at $\alpha = 0.05$ (two-tailed). We also computed the effect size (Cliff's δ) [59] to measure the magnitude of the difference in each test. We followed practical guidelines [59] to interpret effect sizes: "negligible" for $|\delta| < 0.147$, "small" for $0.147 \leq |\delta| < 0.330$, "medium" for $0.330 \leq |\delta| < 0.474$, and "large" for $|\delta| \geq 0.474$. We chose these nonparametric methods, because we did not assume population normality nor homoscedasticity.

TABLE II. CRASHES MANUALLY INJECTED INTO THE AUTS

AUT	Bug Location
Calculator	· Calculator.java (Touch = button) · Calculator.java (Touch advanced menu) · CalculatorEditable.java (Touch dot button) · EventListener.java (Touch +row button) · MatrixModule.java (Transpose matrix)
Frex	· FrexActivity.java (Touch zoom) · SettingsActivity.java (Touch settings option) · FrexActivity.java (Touch fractal-type spinner) · ManagerActivity.java (Delete fractal) · SeekBarConfigurer.java (Modify using seek bar)
GameMaster Dice	· NumberPicker.java (Touch die's sides picker) · GameMasterDice.java (Touch about app in menu) · GameMasterDice.java (Touch more option) · NumberPickerButton.java (Touch modifier picker) · DiceSet.java (Roll die with modifiers)
Simple Deadlines	· Settings.java (Touch settings button) · DeadLineListFragment.java (Delete deadline) · DeadLineAdapter.java (Touch deadline checkbox) · EditorDialogFragment.java (Create deadline) · DayCounterView.java (Modify day)
Stickeroid	· CreateCollection.java (Create collection) · EditCollection.java (Touch filter option) · Main.java (Back up with no data) · Utils.java (Show toast) · Main.java (Touch about app in menu)
Units Converter	· Product.java (Operation with products) · Units.java (Clean fields) · Value.java (Convert invalid units) · Units.java (Longtouch backspace) · ValueGui.java (Transform any unit)

D. Empirical Study Reproducibility Package

All the experimental material used in our study is publicly available in our online appendix [60]. In particular, we provide the original APKs and source code, tainted APKs and source code, descriptions and artifacts used for the user study, logcat traces for the injected bugs, human-written bug reports and CRASHDROID reports, and the results of our experiments.

V. EMPIRICAL RESULTS

CRASHDROID is designed to automatically verify its recommendations, a set of expressive bug reports, before presenting them to the user by running the generated replay scripts on the device. Reliably classifying execution profiles in the CRASHDROID database is critically important to this end. The LCS-based heuristic is designed to push a scenario producing the crash to the top of the list of candidates. In this context, the key concern is to reduce the amount of time it takes to recommend a set of valid suggestions to the user by minimizing the number of scenarios to run before finding at least one

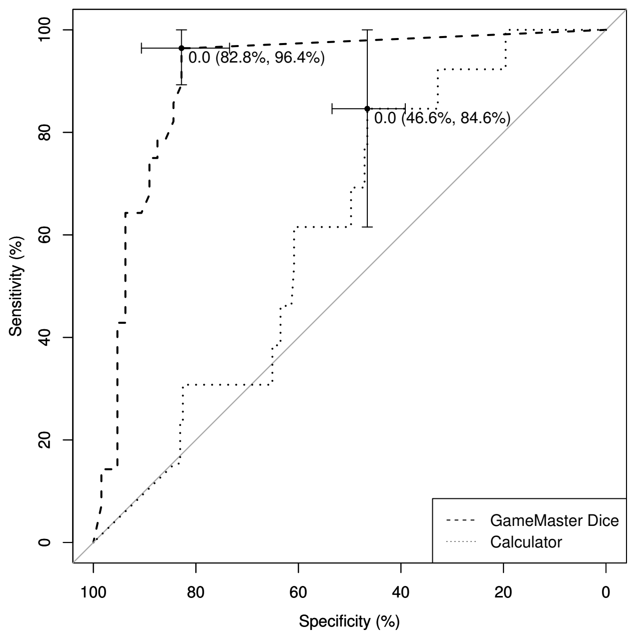

Fig. 2. Given a call stack from a crash report, CRASHDROID computes the similarity of the call stack to each profile in its database (Sec. III-B). Accordingly, the profiles can be sorted and passed to a binary classifier to label each scenario (using a user-defined threshold) as either a bug-producing scenario or not. The performance of this binary classifier can be measured with a ROC curve, depicting the true positive rate (or "sensitivity") versus the true negative rate (or "specificity"). High *sensitivity* suggests a low Type II error rate, and high *specificity* suggests a low Type I error rate. The classifier's "lift" measures its effectiveness relative to randomly labeling the scenarios. The LCS-based heuristic provides virtually no lift for *Calculator*, but it provides some lift for *GameMaster Dice*.

scenario that reproduces the issue. It is also important for the valid suggestions to be readable, concise, and unambiguous to ensure reproducibility. Examples of the CRASHDROID bug reports are in Table III and our online appendix [60].

A. RQ1 Evaluating CrashDroid's Effectiveness

In many cases, the normalized length of the LCS did not effectively prioritize the profiles in the CRASHDROID database such that the profiles' scores were strictly decreasing. The similarity measure's performance is best illustrated with a receiver operating characteristic (ROC) curve. Fig. 2 represents the trade-off between the true positive rate and the true negative rate for two apps, *Calculator* and *GameMaster Dice*. Given the call stack from a crash report, there were typically several profiles with the same score. Thus, if 10 profiles score a 1.0, any one of the 10 corresponding scenarios could be the first to be placed in the ranked list of candidate profiles. The ROC curve represents a run of profiles with the same score (and different bug-producing classifications) as a line segment with a positive slope. Clearly, the normalized length of the LCS does not provide any lift for *Calculator* bugs. In other words, the probability that a randomly chosen positive instance will be ranked ahead of a randomly chosen negative instance is 0.5. The slope of the plot in the bottom left-hand corner suggests that the top profiles all have the same score, yet there is entropy in the target. Indeed, in this particular case, several profiles have one and only one method in common with the call stack, yielding a score of 1.0. This means the heuristic is no better than randomly sampling from these "top" profiles without replacement until the AUT crashes, which discounts the normalized length of the LCS as an effective classifier for identifying bug-producing profiles.

TABLE III. EXAMPLES OF STEPS TO REPRODUCE A CRASH

Calculator Bug 1
Human 1: 1. User inserts a number using numeric keypad; 2. User selects a mathematical operator; 3. User inserts another number using numeric keypad; 4. User presses the equal sign; 5. System crashes without showing the result **Human 2:** 1 - Add a correctly formed mathematical expression to be evaluated by the calculator. If the expression is not correctly formed the bug won't be reproduced and the calculator will produce an 'Error' message which is the correct behavior. 2 - Press the '=' key. **CRASHDROID:** 1. I touched the 6 button on the touchscreen. 2. I touched the * button on the touchscreen. 3. I touched the 7 button on the touchscreen. 4. I touched the + button on the touchscreen. 5. I touched the 7 button on the touchscreen. 6. I touched the = button on the touchscreen.
GameMaster Dice Bug 1
Human 1: 1. User long-presses one of the dice-button; 2. System shows the options available; 3. User selects the "More…" option; 4. System shows a window where the user can config. the dice; 5. User presses "+" button in order to customize the dice; 6. System crashes without showing the result **Human 2:** 1 - Open the 'Configure Dice' menu via one of the following options: Long press (for about half a second) any dice button, choose 'More' in the popup dialog, Tap the 'more' button (last one on the right side), choose 'More' in the popup dialog. 2 - Tap the increment button for the second value (the plus sign on the top-right position). The app should crash. **CRASHDROID:** 1. I touched the "…" button at the bottom of the touchscreen. 2. I touched the "More…" list view item in the window that appears. 3. I touched the plus button on the left side of the touchscreen.

Naturally, while these profiles may have the same score, they likely take different amounts of time to replay. For five of the six apps listed in Tab. I, we timed each replay script in the CRASHDROID database and linked these times to the corresponding profile. To measure the amount of time for CRASHDROID to crash the AUT, we designed a Monte Carlo simulation to measure the expected time for CRASHDROID to produce a bug report that affirmatively crashed the app. The purpose of the simulation was to shuffle the order of profiles with the same score to measure CRASHDROID's expected performance—running 100 trials for each bug and counting the number of seconds to crash the AUT and the number of scenarios replayed. Tab. IV summarizes the results for RQ1. Column (3) lists the median number of replay scripts that were played by CRASHDROID to crash the AUT. This column highlights the poor lift for *Calculator* bugs as opposed to *GameMaster Dice* bugs. Column (4) lists the median number of kernel events fired by CRASHDROID before the AUT crashed. Columns (5)–

TABLE IV. RQ1 RESULTS IN TERMS OF TIME/EVENTS REQUIRED BY CRASHROID, HUMAN PARTICIPANTS, AND MONKEY TO CRASH AN APP

		CrashDroid								Humans		Monkey		
AUT	BugID	(3) Scripts	(4) Events	(5) Min	(6) Q_1	(7) Q_2	(8) Q_3	(9) Max	(10) μ	(11) H_1	(12) H_2	(13) Time	(14) Events	(15) Crashes
Calculator	1	1	66	1.1	3.5	7.8	11.2	65.1	9.6	480	360	N/A	N/A	0
	2	27	124	48.9	159.9	343.3	511.8	1,214.3	387.2	1,680	480	37.4	614	1
	3	2	28	19.5	19.5	19.5	19.5	19.5	19.5	120	-	31.5	478	3
	4	17	61	7.9	104.6	231.4	519.8	1233.9	332.2	180	480	N/A	N/A	0
	5	9	263	7.4	47.7	96.3	198.9	365.3	127.4	420	720	N/A	N/A	0
GameMaster Dice	1	1	32	2.3	2.3	5.8	10.8	16.7	6.8	300	780	25.9	375	1
	2	2	16	1.3	4.4	13.4	40.7	89.2	24.1	300	180	30.1	564	2
	3	2	30	0.3	2.1	11.4	32.1	154.4	22.0	300	420	11.2	196	2
	4	1	32	2.3	2.3	3.3	10.8	11.9	5.3	780	420	N/A	N/A	0
	5	1	44	5.4	5.4	5.4	5.4	5.4	5.4	480	300	15.4	238	3
Simple Deadlines	1	1	16	0.8	0.9	1.2	1.5	26.3	2.8	-	-	54.3	766	4
	2	43	N/A	799.9	799.9	799.9	799.9	799.9	799.9	-	-	N/A	N/A	0
	3	1	160	12.3	12.3	17.4	17.9	17.9	15.8	-	-	N/A	N/A	0
	4	1	163	7.7	12.8	16.2	22.2	45.0	18.1	-	-	N/A	N/A	0
	5	1	8	0.3	0.3	0.3	0.3	0.3	0.3	-	-	12.4	170	3
Stickeroid	1	1	92	6.7	7.2	9.0	10.2	19.1	9.8	-	-	55.1	823	3
	2	1	189	13.6	13.6	15.5	17.3	17.3	15.5	-	-	N/A	N/A	0
	3	15	N/A	344.1	344.1	344.1	344.1	344.1	344.1	-	-	N/A	N/A	0
	4	37	79	510.6	510.6	510.6	510.6	510.6	510.6	-	-	N/A	N/A	0
	5	11	214	25.0	97.2	160.1	251.9	527.4	185.9	-	-	N/A	N/A	0
Units Converter	1	37	N/A	659.8	659.8	659.8	659.8	659.8	659.8	-	-	N/A	N/A	0
	2	4	180	23.2	61.7	83.0	112.3	192.4	87.9	-	-	24.6	278	1
	3	37	N/A	659.0	659.0	659.0	659.0	659.0	659.0	-	-	2.7	23	1
	4	37	N/A	658.7	658.7	658.7	658.7	658.7	658.7	-	-	N/A	N/A	0
	5	4	180	23.2	61.0	72.6	102.3	194.0	83.9	-	-	N/A	N/A	0

(10) list descriptive statistics for the amount of time, measured in seconds, for CRASHDROID to crash the AUT. Columns (11)–(12) list the amount of time, measured in seconds, for the two participants (outside the College of William and Mary) to reproduce the issue and compose a bug report. Column (13) lists the median number of seconds for Monkey to crash the AUT. Column (14) lists the median number of events fired by Monkey for the trials where the AUT crashed; otherwise, it lists "N/A" when the app did not crash before a timeout. Qualitatively, the CRASHDROID statistics suggest the system is reasonably effective, in this context, compared to humans and Monkey. Note that the time required for Monkey includes short delays of 10 ms interleaved between events. On the other hand, we did not use *fast forwarding* to speed up CRASHDROID's execution. This mechanism may drastically reduce the execution time of replay scripts [21]. Moreover, CRASHDROID was able to reproduce the bugs in every case, whereas Monkey could not reproduce 14 of the 25 bugs.

B. RQ2 Evaluating CrashDroid's Bug Reports

Readability: How readable is the bug report? The second quartiles of human-written reports Q_2^h and CRASHDROID reports Q_2^c were 4.5 and 5.0, respectively (Tab. V). We ran a Mann-Whitney U test to evaluate the difference in the responses to a five-level Likert item. We did not find a statistically significant difference at $\alpha = 0.05$ (two-tailed). This is a positive result since CRASHDROID's bug reports are as readable as those produced by developers.

Conciseness: How concise is the bug report? The second quartiles of human-written reports Q_2^h and CRASHDROID reports Q_2^c were 4.5 and 4.0, respectively (Tab. V). The Mann-Whitney U test aimed at evaluating the difference in the responses for conciseness did not report a statistically significant difference at $\alpha = 0.05$ (two-tailed). This is another positive result since CRASHDROID's bug reports are as concise as those produced by developers.

Regarding conciseness of human versus CRASHDROID reports, we obtained further evidence from the participants. For instance, in the case of *Calculator Bug 1*, the human-written report included the sentence *"1. Add a correctly formed mathematical expression... the correct behavior."* One participant assessed the report as two, on the five-level Likert item, with the comment that *"The report is too wordy (the corrected behavior sentence in step 1) and does not describe what happens when = is key."* The CRASHDROID report included detailed steps ending with a sentence that triggered the crash: *"I touched the '6' button on the touchscreen... I touched the '=' button on the touchscreen."*

Reproducibility: How easy was it to reproduce the bug report? The second quartiles of human-written reports Q_2^h and CRASHDROID reports Q_2^c were both equal to 5.0 (Tab. V). Again, the Mann-Whitney U test did not report a statistically significant difference at $\alpha = 0.05$ (two-tailed). Additionally, if the study participant was able to reproduce a bug, we also asked them to document the number of times it took them to go through the steps before they could reproduce the crash. It is worth noting that every CRASHDROID report only required one pass whereas human-written reports required up to 12 passes in some cases. A qualitative review of study participants' amplifying comments suggested that human-written reports are prone to ambiguity when describing certain events. For example, in the case of *GameMaster Dice Bug 1*, one participant wrote *"...the instructions should clarify which of the three + signs to press to be sure this is one that will surely crash the app..."* when referring to a human-written report. The human-written report includes the step *"5. User presses + button in order to customize the dice"* while the corresponding step in the CRASHDROID report is *"I touched the plus button on the left side of the touchscreen in the window."* Another example is *GameMaster Dice Bug 5*, where a human-written report was assessed as two, on the five-level Likert item, for each of the three properties. The participant explained that the report *"...needs to explain what a non-zero modifier is. For step B1, is the second value the sides or the modifier? I created several zero modifier dice and pressing ok did not crash the app."* Another participant assessed the same report as three for each of the three properties and provided a similar argument around the "non-zero modifier" ambiguity: *"The instructions were terrible, and I guess they assumed that the user would know how to get a die with a non-zero modifier."* The human-written report described the steps in terms of a die with a non-zero modifier whereas the CRASHDROID report had a specific step that triggered the bug *"I touched the 1d6+4 button."*

Human-written reports for *Calculator Bug 2* were ranked less than three in all cases whereas CRASHDROID reports were ranked greater than three in two cases. The reason is that human instructions did not include a clear description of how to show the "Advanced panel" option in the menu, but the CRASHDROID report included the step (change from the basic to the Hex panel) that activates the option in the menu: *"I swiped right on the top half of the screen: I touched the "Hex" button...I touched the 'Advanced panel' menu option."*

Therefore, the qualitative and quantitative results suggest that CRASHDROID reports are as effective natural language descriptions as those produced by developers in this context.

C. Threats to Validity

Threats to **construct validity** concern the relationship between theory and observation. For the first study we only had five crashes for six apps (30 in total). We are aware that the types of crashes we injected may not necessarily be representative of real-word crashes. However, we tried to inject crashes into likely locations. Moreover, crash injections were done by an author who has significant Android app development experience and investigated a number of open-source crash bug reports. For the second study, we asked two graduate students to manually compose bug reports. While this could have impacted the results, the two subjects are experienced developers who have significant industrial experience, including writing and resolving bug reports.

Threats to **internal validity** concern co-factors that could influence our results, and they are mainly related to various measurement settings used in our study. For the second study we asked participants to time themselves. Obviously, we are not expecting to get results with second level accuracy since our comparison between CRASHDROID-generated reports and human-written reports is rather relative.

Threats to **external validity** concern generalization of the obtained results. The main threat is related to recruiting students for the bug comparison study. While most of these students have only academic programming experience, most of them took a mobile app development class. Finally, we limited the second study to bug reports from two apps, because we could provide only a limited number of bug reports that participants could complete in two hours.

TABLE V. MANN-WHITNEY U TEST RESULTS FOR RQ2

Measure	Q_2^h	Q_2^c	n^h	n^c	U	p-value	Cliff's δ
Readability	4.5	5.0	42	42	753.0	0.2501	0.1254
Conciseness	4.5	4.0	42	42	730.5	0.1770	0.1474
Reproducibility	5.0	5.0	42	42	870.5	0.9203	0.0107

VI. CONCLUSION AND FUTURE WORK

We presented CRASHDROID, a system for automatically producing expressive bug reports and replay scripts given a stack trace from a crash report. The system compared favorably in terms of readability, conciseness, and reproducibility to human-written reports. It also compared favorably in terms of time and its ability to crash AUTs to Monkey. Moreover, the system is designed to achieve 100% precision since it validates its proposals before presenting any results to the user.

In computational linguistics, statistical machine translation concerns the automatic translation of text or speech from one natural language to another using, for instance, the noisy channel model of sentence pair generation. We plan to examine how traceability links between natural language sentences, AUT class method traces, and kernel event streams can leverage a translation engine for dynamically constructing ad hoc scenarios. One purpose of this engine may be to "bootstrap" both descriptions and replay scripts from existing data. These new scenarios can be added to the CRASHDROID database to automatically improve the coverage that descriptions provide for an app's functionality.

ACKNOWLEDGMENT

We would like to thank Conner Kasten for his contribution to early versions of this work that used Dynodroid as an underlying engine. We would like to acknowledge Santiago Baldrich and Michele Tufano for writing bug reports for the tainted APKs. Next, we would like to acknowledge Zhaoliang Duan, Boyang Li, Qi Luo, Kevin Moran, Du Shen, Christopher Vendome, and Kyle Wallace for producing feature descriptions with replay scenarios for the apps used in the study. Finally, we would like to thank all the students from the Universidad Nacional de Colombia who helped us with CRASHDROID debugging, data collection, and the user study. This work is supported in part by the NSF CCF-1218129 and NSF-1253837 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

REFERENCES

[1] R. Minelli and M. Lanza, "Software analytics for mobile applications–insights & lessons learned," in *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, ser. CSMR '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 144–153.

[2] I. J. M. Ruiz, "Large-scale empirical studies of mobile apps," Master's thesis, Queen's University, 2013.

[3] M. D. Syer, M. Nagappan, A. E. Hassan, and B. Adams, "Revisiting prior empirical findings for mobile apps: An empirical case study on the 15 most popular open-source Android apps," in *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '13. Riverton, NJ, USA: IBM Corp., 2013, pp. 283–297.

[4] C. Roy and A. A. Moamen, "Exploring development practices of Android mobile apps from different categories," in *8th International Workshop on Software Quality and Maintainability*, ser. SQM '14, 2014.

[5] C. Hu and I. Neamtiu, "Automating GUI testing for Android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. New York, NY, USA: ACM, 2011, pp. 77–83.

[6] P. Bhattacharya, L. Ulanova, I. Neamtiu, and S. C. Koduru, "An empirical analysis of bug reports and bug fixing in open source Android apps," in *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, ser. CSMR '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 133–143.

[7] J. Kochhar, J. Keng, and T. Biying, "An empirical study on bug reports of Android 3rd party libraries," *Singapore Management University*, 2013.

[8] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008, pp. 27–30.

[9] A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in *MSR*, 2010, pp. 118–121.

[10] N. Chen, J. Lin, S. C. H. Hoi, X. Xiao, and B. Zhang, "AR-miner: Mining informative reviews for developers from mobile app marketplace," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE '14. New York, NY, USA: ACM, 2014, pp. 767–778.

[11] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 641–660.

[12] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE '13. New York, NY, USA: ACM, 2013, pp. 224–234.

[13] (2014) MonkeyRunner. [Online]. Available: developer.android.com/tools/help/monkeyrunner_concepts.html

[14] (2014) UI/Application exerciser monkey. [Online]. Available: developer.android.com/tools/help/monkey.html

[15] (2014) Robotium. [Online]. Available: code.google.com/p/robotium

[16] (2014) GUITAR. [Online]. Available: sourceforge.net/projects/guitar

[17] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of Android apps," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '14. New York, NY, USA: ACM, 2014, pp. 599–609.

[18] W. Choi, G. Necula, and K. Sen, "Guided GUI testing of Android apps with minimal restart and approximate learning," in *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'13)*, 2013, pp. 623–640.

[19] W. Yang, M. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," in *16th International Conference on Fundamental Approaches to Software Engineering (FASE'13)*, 2013, pp. 250–265.

[20] S. Malek, N. Esfahani, T. Kacem, R. Mahmood, N. Mirzaei, and A. Stavrou, "A framework for automated security testing of Android applications on the cloud," in *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability Companion*, ser. SERE-C '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 35–36.

[21] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "RERAN: Timing- and touch-sensitive record and replay for Android," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 72–81.

[22] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[23] S. Artzi, S. Kim, and M. D. Ernst, "ReCrash: Making software failures reproducible by preserving object states," in *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ser. ECOOP '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 542–565.

[24] C. Csallner and Y. Smaragdakis, "JCrasher: An automatic robustness tester for Java," *Softw. Pract. Exper.*, vol. 34, no. 11, pp. 1025–1050, Sep. 2004.

[25] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *Proceedings of the 19th European Conference on Object-Oriented Programming*, ser. ECOOP '05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 504–527.

[26] C. Csallner, Y. Smaragdakis, and T. Xie, "DSD-Crasher: A hybrid analysis tool for bug finding," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 2, pp. 8:1–8:37, May 2008.

[27] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–84.

[28] M. Brodie, S. Ma, G. Lohman, T. Syeda-Mahmood, L. Mignet, N. Modani, M. Wilding, J. Champlin, and P. Sohn, "Quickly finding known software problems via automated symptom matching," in *Proceedings of the 2nd International Conference on Autonomic Computing*, ser. ICAC '05. Piscataway, NJ, USA: IEEE Press, 2005, pp. 101–110.

[29] (2007) Apple crash reporter. [Online]. Available: developer.apple.com/technotes/tn2004/tn2123.html

[30] (2007) Online crash analysis. [Online]. Available: oca.microsoft.com/

[31] (2007) Crash reports. [Online]. Available: talkback-public.mozilla.org

[32] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "Rebucket: A method for clustering duplicate crash reports based on call stack similarity," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 1084–1093.

[33] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "Crashlocator: Locating crashing faults based on crash stacks," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA '14. New York, NY, USA: ACM, 2014, pp. 204–214.

[34] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 14–24.

[35] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, Oct. 2005.

[36] ——, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2002, pp. 1–10.

[37] S. Yoo, M. Harman, and D. Clark, "Fault localization prioritization: Comparing information-theoretic and coverage-based approaches," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, pp. 19:1–19:29, Jul. 2013.

[38] H. Seo and S. Kim, "Predicting recurring crash stacks," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '12. New York, NY, USA: ACM, 2012, pp. 180–189.

[39] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: Statistical model-based bug localization," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE '13. New York, NY, USA: ACM, 2005, pp. 286–295.

[40] S. Kim, T. Zimmermann, and N. Nagappan, "Crash graphs: An aggregated view of multiple crashes to improve crash triage," in *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, June 2011, pp. 486–493.

[41] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 467–477.

[42] W. Jin and A. Orso, "F3: Fault localization for field failures," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA '13. New York, NY, USA: ACM, 2013, pp. 213–223.

[43] ——, "Bugredux: Reproducing field failures for in-house debugging," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 474–484.

[44] T. Takala, M. Katara, and J. Harty, "Experiences of system-level model-based GUI testing of an Android application," in *Fourth International Conference on Software Testing, Verification and Validation (ICST'11)*, 2011, pp. 377–386.

[45] L. Ravindranath, S. nath, J. Padhye, and H. Balakrishnan, "Automatic and scalable fault detection for mobile applications," in *12th annual international conference on Mobile systems, applications, and services (MobiSys'14)*, 2014, pp. 190–203.

[46] D. Amalfitano, A. Fasolino, P. Tramontana, S. De Carmine, and A. Memon, "Using GUI ripping for automated testing of Android applications," in *International Conference on Automated Software Engineering (ASE'12)*, 258-261, Ed., 2012.

[47] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'13)*, 2013, pp. 641–660.

[48] C. S. Jensen, M. R. Prasad, and A. Moller, "Automated testing with targeted event sequence generation," in *International Symposium on Software Testing and Analysis (ISSTA'13)*, 2013, pp. 67–77.

[49] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. Ta, and A. Memon, "Mobiguitar – a tool for automated model-based testing of mobile apps," *IEEE Software*, vol. 99, no. PrePrints, p. 1, 2014.

[50] Google. Android developer guide. http://developer.android.com/guide/topics/ui/index.html.

[51] (2014) Getevent. [Online]. Available: https://source.android.com/devices/input/getevent.html

[52] RERAN at github. https://github.com/nejstastnejsistene/RERAN/.

[53] (2014) Android debug bridge. [Online]. Available: http://developer.android.com/tools/help/adb.html

[54] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy API usage patterns in Android apps: An empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR '14. New York, NY, USA: ACM, 2014, pp. 2–11.

[55] (2014) dmtracedump. [Online]. Available: http://developer.android.com/tools/help/dmtracedump.html

[56] F-doid. https://f-droid.org.

[57] Google. Android API reference - view class. http://developer.android.com/reference/android/view/View.html.

[58] D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, 2nd ed. Chapman & Hall/CRC, 2000.

[59] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[60] M. White, M. Linares-Vásquez, P. Johnson, C. Bernal-Cárdenas, and D. Poshyvanyk. Crashdroid - online appendix. http://www.cs.wm.edu/semeru/data/ICPC15-CrashDroid/.