2016 IEEE International Conference on Software Testing, Verification and Validation

Automatically Discovering, Reporting and Reproducing Android Application Crashes

Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk
College of William & Mary
{kpmoran, mlinarev, cebernal, cvendome, denys}@cs.wm.edu

***Abstract*—Mobile developers face unique challenges when detecting and reporting crashes in apps due to their prevailing GUI event-driven nature and additional sources of inputs (e.g., sensor readings). To support developers in these tasks, we introduce a novel, automated approach called CRASHSCOPE. This tool explores a given Android app using systematic input generation, according to several strategies informed by static and dynamic analyses, with the intrinsic goal of triggering crashes. When a crash is detected, CRASHSCOPE generates an augmented crash report containing screenshots, detailed crash reproduction steps, the captured exception stack trace, and a fully replayable script that automatically reproduces the crash on a target device(s).**

We evaluated CRASHSCOPE's effectiveness in discovering crashes as compared to five state-of-the-art Android input generation tools on 61 applications. The results demonstrate that CRASHSCOPE performs about as well as current tools for detecting crashes and provides more detailed fault information. Additionally, in a study analyzing eight real-world Android app crashes, we found that CRASHSCOPE's reports are easily readable and allow for reliable reproduction of crashes by presenting more explicit information than human written reports.

I. INTRODUCTION

Continued growth in the mobile hardware and application marketplace is being driven by a landscape where users tend to prefer mobile smart devices and apps for tasks over their desktop counterparts. The gesture-driven nature of mobile apps has given rise to new challenges encountered by programmers during development and maintenance, specifically with regard to testing and debugging [41]. One of the most difficult [22], [24] and important maintenance tasks is the creation and resolution of bug reports [35]. Reports concerning application crashes are of particular importance to developers, because crashes represent a jarring software fault that is directly user facing and immediately impacts an app's utility and sucess. If an app is not behaving as expected due to crashes, missing features, or other bugs, nearly half of users are likely to abandon the app for a competitor [12] in a marketplace such as Google Play [10].

Mobile developers heavily rely on user reviews [42], [49], [65], crash reports from the field in the form of stack traces, or reports in open source issue tracking systems to detect bugs in their apps. In each of these cases, the bug/crash reports are typically lacking in information [27], [41], containing only a stack trace, overly detailed logs or loosely structured natural language (NL) information regarding the crash [23]. This is not surprising as previous studies showed that information, which is most useful for a developer resolving a bug report (e.g., reproduction steps, stack traces and test cases), is often the most difficult information for reporters to provide [33]. Furthermore, the absence of this information is a major cause of developers failing to reproduce bug/crash reports [22]. In addition to the quality of the reports, some other factors specific to Android apps such as hardware and software fragmentation [3], API instability and fault-proneness [21], [48], the event-driven nature of Android apps, gesture-based interaction, sensor interfaces, and the possibility of multiple contextual states (e.g., wifi/GPS on/off) make the process of detecting, reporting, and reproducing crashes challenging.

Motivated by these current issues developers face regarding mobile application crashes, we designed and implemented CRASHSCOPE, a practical system that automatically discovers, reports, and reproduces crashes for Android applications. CRASHSCOPE explores a given app using a systematic input generation algorithm and produces expressive crash reports with explicit steps for reproduction in an easily readable natural language format. This approach requires only an `.apk` file and an Android emulator or device to operate and requires no instrumentation of the subject apps or the Android OS. The entirety of the CRASHSCOPE workflow is completely automated, requiring no developer intervention, other than reading produced reports. Our systematic execution includes different exploration strategies, aimed at eliciting crashes from Android apps, which include automatic text generation capabilities based on the context of allowable characters for text entry fields, and targeted testing of contextual features, such as the orientation of the device, wireless interfaces, and sensors. We specifically tailored these features to test the common causes of app crashes as identified by previous studies [26], [45], [79]. During execution, CRASHSCOPE captures detailed information about the subject app, such as the inputs sent to the device, screenshots and GUI information, exceptions, and crash information. This information is then translated into detailed crash reports and replayable scripts, for any encountered crash.

This paper makes the following noteworthy contributions:

1) We design and implement a practical and automatic approach for discovering, reporting, and reproducing Android application crashes, called CRASHSCOPE. To the best of the author's knowledge, this is the first approach that is able to generate expressive, detailed crash reports for mobile apps, including screenshots and augmented NL reproduction steps, in a completely automatic fashion. CRASHSCOPE is also one of the only available fully-automated Android testing approaches

978-1-5090-1827-7/16 $31.00 © 2016 IEEE
DOI 10.1109/ICST.2016.34

that is practical from a developers' perspective, requiring no instrumentation of the subject apps or OS. Our approach builds upon prior research in automated input generation for mobile apps, and implements several exploration strategies, informed by lightweight static analysis that are able to effectively detect crashes and exceptions;

2) We perform a detailed evaluation of the crash detection abilities of CRASHSCOPE on 61 Android apps as compared to five state-of-the-art Android input generation tools: Dynodroid [53], Gui-Ripper [17], PUMA [36], A^{3}E [20], and Monkey [7]. Our results show that CRASHSCOPE performs nearly as well as current tools in terms of detecting crashes, while automatically generating detailed reports and replayable scripts;

3) We design and carry out a user study evaluating the *reproducibility* and *readability* of our automatically generated bug reports through comparison to human written crash reports for eight open source apps. The results indicate that CRASHSCOPE reports offer more detail, while being at least as useful as the human written reports;

4) We make our experimental dataset and crash reports available in our online appendix [58].

II. Related Work & Motivation

In this section, we overview the current landscape of automated testing and input generation tools for Android, discussing limitations of these approaches while illustrating CRASHSCOPE'S novelty in context. Several approaches for detecting and reproducing crashes are available in literature [30]–[32], [39], [40], [43], [51], [52], [63], [64], [69], [74], [78], [80]–[82]; however, we forgo discussion of these approaches, as they are not presented in the context of mobile apps, and hence do not consider the unique associated challenges.

A. Input Generation & Testing Tools for Mobile Apps

Approaches for automated input generation can be broadly grouped into three major categories [29]: *random-based input generation* [7], [11], [53], [68], [76], *systematic input generation* [18]–[20], and *model-based testing* [18], [20], [28], [36], [75], [79]. We outline some features of several mobile testing and input generation tools in Table I.

Random-Based Input Generation techniques rely on choosing arbitrary GUI or contextual events; they can adapt such a strategy through biasing the selection with the frequency of past events and purposefully select those previously un-exercised. Dynodroid [53] introduces such an approach by maintaining a history of event execution frequencies in a context-sensitive manner that tracks relevant events throughout the execution. Intent Fuzzer [68] generates intents to test apps by taking into consideration information extracted in an offline static analysis phase. However, it does not easily scale for large apps due to the number of paths that need to be stored in memory for processing, and only tests app intents. VanarSena [66] is a tool for Windows Phone that instruments app binaries in order to test apps for externally-inducible faults using random exploration and the injection of adverse contextual conditions. VanarSena is capable of generating crash reports containing a stack trace and exception. Compared to VanarSena, CRASHSCOPE does not require any type of instrumentation and is able to generate more detailed crash reports that include screenshots, natural language reproduction steps, and replayable scripts.

Systematic Input Generation approaches execute input events according to a pre-defined heuristic such as Breadth or Depth First Search (BFS/DFS). For instance, *Android*Ripper dynamically analyzes an app's GUI with the aim of obtaining a fireable sequence of events. It then extracts task lists according to the GUI hierarchy of an app and systematically executes the events in the generated lists. A^{3}E [20] leverages static bytecode, taint-style, dataflow analysis in order to construct a high-level control flow-graph that captures allowable transitions between app screens (activities). The tool implements two exploration strategies based on this graph, a simple DFS, and targeted exploration. ACTEve [19] is a concolic-based testing approach for smartphone apps that symbolically tracks events from the point where they originate to the point where they are handled. The tool is similar in efficacy to concolic testing while avoiding the path-explosion problem, but requires Soot for instrumentation [13].

Model-Based Input Generation approaches attempt to build detailed models (e.g., capturing relevant screen and event flows) of apps in order to be able to explore them more effectively. MobiGUItar [17] is the evolution of AndroidRipper and models the states of an Android app's GUI, executing feasible events by observing and updating the current GUI state of an app. Swifthand [28] uses active learning to infer a model of an app and uses the learned model to generate inputs that drive the execution of the app towards unexplored states. Additionally, this approach aims to minimize app restarts, due to restart overhead, by exploring all screens that can be reached from the initial state of the app first. QUANTUM [79] represents an approach for automatically generating GUI-based app-agnostic oracles for the detection of defects in Android applications. The tool implements a set of app-agnostic features, namely: rotation, killing and restarting, pausing and resuming, and back button. It is able to present a visual representation of the app to the developer for verification after exercising the previous features at targeted locations during testing. While QUANTUM can not produce detailed crash reports, it can produce replayable Junit/Robotium test cases. ORBIT [75] uses static analysis to extract GUI-components in the layout files and locate any handlers or event listeners in the source files in order to extract possible fireable actions. Then, it uses dynamic GUI crawling to construct a model of an app. MonkeyLab [50] is an approach that mines application usages to extract *n*-grams representing the GUI and usage models. MonkeyLab is capable of generating unseen and actionable sequences of events that achieve orthogonal code coverage compared to the application usages mined to generate the model. Tonella et. al. [71] also applied interpolated *n*-grams to model based testing, but not in the context of mobile apps.

Other Approaches in Mobile Testing and Bug Reporting encompass emerging work that attempts to solve specific problems related to testing and bug reporting in a mobile

TABLE I
AN OVERVIEW OF FEATURES IN AUTOMATED TESTING APPROACHES FOR MOBILE APPLICATIONS

Tool Name	Instrumentation	GUI Exploration	Types of Events	Crash Resilient	Replayable Test Cases	NL Crash Reports	Emulators, Devices
Dynodroid [53]	Yes	Guided/Random	System, GUI, Text	Yes	No	No	No
EvoDroid [54]	No	System/Evo	GUI	No	No	No	N/A
*Android*Ripper [18]	Yes	Systematic	GUI, Text	No	No	No	N/A
MobiGUItar [17]	Yes	Model-Based	GUI, Text	No	Yes	No	N/A
A^3E Depth-First [20]	Yes	Systematic	GUI	No	No	No	Yes
A^3E Targeted [20]	Yes	Model-Based	GUI	No	No	No	Yes
Swifthand [28]	Yes	Model-Based	GUI, Text	N/A	No	No	Yes
PUMA [36]	Yes	Programmable	System, GUI, Text	N/A	No	No	Yes
ACTEve [19]	Yes	Systematic	GUI	N/A	No	No	Yes
VANARSena [66]	Yes	Random	System, GUI, Text	Yes	Yes	No	N/A
Thor [16]	Yes	Test Cases	Test Case Events	N/A	N/A	No	No
QUANTUM [79]	Yes	Model-Based	System, GUI	N/A	Yes	No	N/A
AppDoctor [37]	Yes	Multiple	System, GUI^2, Text	Yes	Yes	No	N/A
ORBIT [75]	No	Model-Based	GUI	N/A	No	No	N/A
SPAG-C [46]	No	Record/Replay	GUI	N/A	N/A	No	No
JPF-Android [72]	No	Scripting	GUI	N/A	Yes	No	N/A
MonkeyLab [50]	No	Model-based	GUI, Text	No	Yes	No	Yes
CrashDroid [73]	No	Manual Rec/Replay	GUI, Text	Manual	Yes	Yes	Yes
SIG-Droid [56]	No	Symbolic	GUI, Text	N/A	Yes	No	N/A
CrashScope	**No**	**Systematic**	**GUI, Text, System**	**Yes**	**Yes**	**Yes**	**Yes**

context. PUMA [36] exposes high-level events for which users can define handlers, in order to create a programmable UI-Automation framework for which developers can implement their own exploration strategies. This system relies on the instrumentation of app binaries and implements a basic random input generation approach. Thor [16] relies on existing test cases for an app and systematically triggers adverse contextual conditions during the execution of the sequences in given test case. AppDoctor [37] takes an "approximate execution" approach by using a side loaded instrumentation app to hook directly into event handlers that are associated with various application GUI-components. While this approach does offer the ability to replay the bugs and presents the developer with stack traces, it must replay crash traces to prune false-positives and it does not offer highly detailed and expressive reports as CRASHSCOPE does. SIG-Droid [56] infers test inputs by relying on symbolic execution, and combining inputs with a GUI model extracted automatically from the source code. SPAG-C [46] is aimed at determining whether or not a given device is capable of correctly displaying an app that is assumed to be working correctly. This approach combines Sikuli, for UI Automation, and Android screencast for capturing the frame buffer of a device. JPF-Android [72] verifies Android apps by running android code on the JVM using a collection of different event sequences and then detecting when certain errors occur using Java PathFinder (JPF). This approach introduces overhead due to the stack manipulation, listeners and logging used by JPF, and is severely limited in terms of the types of events it can properly execute. Evodroid [54] expands upon a typical systematic strategy and leverages evolutionary algorithms that use Interface and Call-Graph models.

The motivation for CRASHSCOPE stems in part from our previous work in mobile bug and crash reporting [59], [60], [73]. CRASHDROID [73] is capable of translating a call stack from an app crash into expressive steps to reproduce a bug. However, it has two noteworthy limitations: first, it is not able to uncover crashes automatically, but instead relies on pre-existing stack traces; second, the tool requires expensive collection of manual traces from real users that need to be annotated using natural language descriptions, which are needed to construct the bug reports and replayable scenarios. FUSION [57], [59], [60] is a bug reporting mechanism that leverages program analysis to facilitate users creating expressive bug reports. However, this approach requires users to create bug reports for functional problems in an app. CRASHSCOPE automates the entire crash discovery process and adds information regarding the contextual state of the app at each reproduction step, resulting in a tool that automates a typically expensive maintenance process for developers.

B. Previous Studies on Mobile App Bug/Crashes

Motivating factors from mobile app bug/crash studies aided us in designing CRASHSCOPE. Two studies stand out in terms of providing information to drive design decisions for our approach. First, Ravindranath *et al.* [66] conducted a study of 25 million real-world crash reports collected from Windows Phone users "in the wild" by the "Windows Phone Error Reporting System" (WPER). Although this study was conducted regarding crashes from a different mobile OS, several of the findings reported in this study are relevant in the context of Android, due to platform similarities: 1) a small number of root causes cover a majority of the crashes examined; 2) many crashes can be mapped to well-defined externally inducible faults, for example, HTTP errors caused by network connectivity issues; 3) the dominant root causes can affect many different user execution paths in an app. The most salient piece of information that can be gleaned from the study and applied in the design of CRASHSCOPE is the following: *An effective crash discovery tool must be able to test different contextual states in a targeted manner, while remaining resilient to encountered crashes so as to uncover crashes present in different program event-sequence paths.* We explain how CRASHSCOPE achieves targeted testing of contextual states using program analysis in Sec. III.

In addition, Zaeem *et al.* [79] conducted a bug study on 106 bugs drawn from 13 open-source Android applications, with the goal of identifying opportunities for automatically generating test cases that include oracles. Most notably, the results of this study were formulated as a categorization of different Android app bugs. Specifically, these categorizations were grouped into three headings: Basic Oracles, App-Agnostic

Oracles, and App-Specific Oracles. CRASHSCOPE uses the well-defined oracles of uncaught exceptions and app crashes to detect faults; however, some of the bug categorizations in this study are useful in triggering these, specifically the app-agnostic categorizations of *Rotation*, *Activity Life-Cycle*, and *Gestures*. Specifically, we implemented a targeted (i.e. localized) version of the double-rotation feature [79].

C. Limitations of Mobile Testing Approaches

While significant progress has been made in the area of testing and automatically generating inputs for mobile applications, the available tools generally exhibit some noteworthy limitations that inspired the development of CRASHSCOPE:

- Previous approaches lack the ability to provide detailed, easy-to-understand testing results for faults discovered during automatic input generation, leaving the developer to sort through and comprehend stack traces, log files, and non-expressive event sequences [29];
- Most approaches for automated input generation are not practical for developers to use, typically due to instrumentation or difficult setup procedures. This is affirmed by the fact developers typically prefer manual over automated testing approaches [41], [44], [49]. As we show, instrumentation can contribute to a higher than necessary developer effort in parsing results from automated approaches, because developers must sift through "false positive" crashes caused by instrumentation.
- Few approaches combine different strategies and features for testing apps through supporting different methodologies for user text input and testing of contextual states (e.g., wifi on/off) in a single holistic approach.

These shortcomings contribute to the low adoption rate of automated testing approaches by mobile developers. In the next section of this paper, we clearly describe how CRASHSCOPE'S design addresses these current limitations in automated mobile input generation and testing tools.

III. CRASHSCOPE DESIGN

In this section, we first describe CRASHSCOPE'S novelty by illustrating how it addresses the limitations discussed in the previous section. We then give an overview of the CRASHSCOPE'S workflow, and the salient features in detail.

CRASHSCOPE addresses the general limitations of existing tools. First, no other automated testing approach, is able to automatically generate *expressive* bug reports (and replayable scripts) for exceptions and crashes discovered by automated input generation for mobile apps. CRASHSCOPE accurately detects crashes and is able to generate *easily readable and detailed* reports without any developer intervention. Second, CRASHSCOPE is a practical tool, requiring only an `.apk` file and an instantiated emulator *or* physical device running Android 4.3 and newer, which constitutes 55% of the current Android OS install base [6]. Operating on emulators CRASHSCOPE is able to parallelize testing on multiple emulators with different specifications, versions of Android, and screen sizes, mitigating a major challenge in app development [41]. Third, inspired by existing approaches [16], [66], [79] CRASHSCOPE is able to explore an app through automated input generation while testing varying contextual states. We extend previous context aware testing techniques by leveraging static analysis to extract targeted locations for testing apps in different contextual states. Finally, our approach is *app-crash-resilient*; it can detect a crash and continue testing the unvisited components and states of the GUI after handling the crash.

The overall workflow of CRASHSCOPE is illustrated in Figure 1. Let us consider the 31C3 Schedule app [1] as a running example to explain the CRASHSCOPE workflow; then, we will discuss the salient features in detail. The first step in running CRASHSCOPE is to obtain the source code of the app, either directly or through decompilation, and detect Activities (by means of static analysis) that are related to contextual features (Figure 1-①) in order to target the testing of such features. In other words, CRASHSCOPE will only test certain contextual app features (e.g., wifi off) if it finds instances where they are implemented in the source code. In the case of 31C3 Schedule, the first activity (screen) of the app makes use of network connectivity, so this screen would be marked as implementing this feature. More details about the contextual features detection are provided in Sec. III-A.

Next, the *GUI Ripping Engine* (Figure 1-②) systematically executes the app using various strategies (Section III-D), including enabling and disabling the contextual features (if run on an emulator) at the Activities of the app identified previously. If during the execution, uncaught exceptions are thrown, or the app crashes, dynamic execution information is saved to the CRASHSCOPE's database (Figure 1-③), including detailed information regarding each event performed during the systematic exploration. In the case of 31C3 Schedule, if systematic execution is continued from the first screen when the network is disabled, a crash occurs. This is because the differing contextual condition exposes a state of the app that would not be otherwise seen.

After the execution data has been saved to the CRASHSCOPE database, the *Natural Language Report Generator* (Figure 1-④, Section III-E) parses the database and processes the information for each step of all executions that ended in a crash, generating an HTML based natural language crash report with expressive steps for reproduction (Figure 1-⑤). In addition, the *Crash Script Generator* (Figure 1-⑥, Section III-F) parses the database and extracts the relevant information for each step in a crashing execution in order to create a replayable script containing `adb input` commands and markers for contextual state changes. The *Script Replayer* (Figure 1-⑦, Section III-F) is able to replay these scripts by executing the sequence of `adb input` commands and interpreting the contextual state change signals, in order to reproduce the crash. In the case of the 31C3 Schedule app, this involves turning off the network connection and trying to interact with one of the app menu headers.

A. Extracting Activity and App-Level Contextual Features

CRASHSCOPE uses Abstract Syntax Tree (AST) based analysis to extract the API-call chains that are involved in invocations of contextual features. In particular, it detects

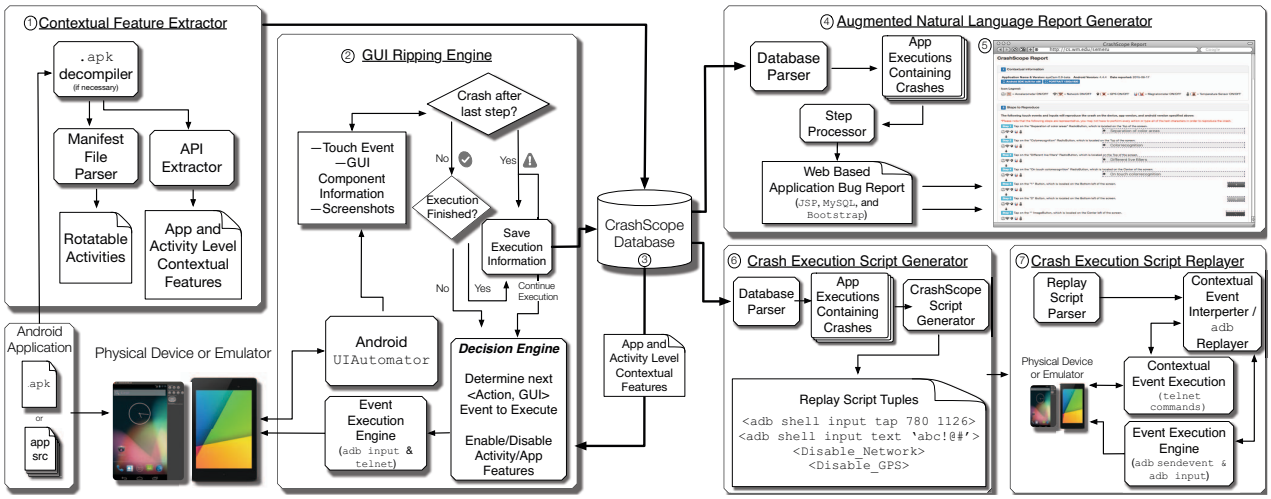

Fig. 1. Overview of CrashScope Workflow

Android API calls related to network connectivity and sensors (i.e., Accelerometer, Magnetometer, Temperature Sensor, and GPS). Because the API calls might not be executed directly by an Activity, CRASHSCOPE performs a call-graph analysis to extract paths ending in a method invoking a contextual API. Because certain API calls may not be traceable through a back-propagated call-chain (e.g., sensor or network implemented as a service), CRASHSCOPE employs two granularities for testing contextual features: activity (screen-) level and app-level. If a particular API call related to one of the contextual features above is able to be traced back to an Activity, then that feature is later tested at the Activity level (i.e., the contextual feature is enabled or disabled when the corresponding Activity is in foreground). If the feature is not able to be linked to an Activity, then the feature is tested at the level of the entire app (i.e., the contextual feature is enabled or disabled at the beginning of the app's execution). To obtain the Activities that are rotatable, CRASHSCOPE parses the AndroidManifest.xml, where rotatable activities must be declared. During testing, if a rotatable activity is encountered while exploring an app, then the screen is rotated from portrait to landscape and back again before any GUI interactions occur to test for proper implementation of the corresponding rotation event-handlers.

B. Exploration of Apps & Crash Detection

To explore an app, CRASHSCOPE dynamically extracts the GUI hierarchy of each app screen visited during the exploration and identifies the clickable and long-clickable components to execute, as well as available components for text inputs (e.g., EditText boxes). The (long-) clickable components are added to a working list to assure that all the clickable components are executed systematically. CRASHSCOPE executes each possible event (i.e., action on an available GUI component) on the current screen according to the GUI hierarchy. If text entry fields are available in a particular app screen, then each text box is filled in before each (long-) clickable component on the screen is exercised. Currently, our *Ripping Engine* supports the *tap*, *long-tap*, and *type* events.

Text entry from the user is a major part of functionality in many Android apps, therefore, CRASHSCOPE's *GUI Ripping Engine* employs a unique text input generation mechanism. CRASHSCOPE detects the type of text expected (e.g., numbers) by a text field, by querying the keyboard type associated to the text field [4]. This is done with the `adb shell dumpsys input_method` command. Once the type of expected input is detected, CRASHSCOPE employs two strategies to generate text inputs: *expected* and *unexpected*. The *expected* strategy generates a string within the keyboard parameters without any punctuation or special characters, whereas the *unexpected* strategy generates random strings with all of the allowable special characters for a given keyboard type. The intuition behind this input generation mechanism is to test instances where a developer may have unknowingly set a keyboard that allows certain characters, but does not properly check for these characters in the code, resulting in a fault. Before the keyboard metadata is read, a *touch* event is executed on the text box to ensure the corresponding keyboard is displayed.

In addition to the text input generation strategies, CRASHSCOPE traverses the GUI hierarchy either from the bottom of the hierarchy up or from the top of the hierarchy down. The rationale for having two such strategies is to generally mimic what a user would do, i.e., executing GUI events without a predefined order. If a transition to another screen is recorded during the exploration, then the GUI-hierarchy of the new screen is detected and the components on the new screen are executed next. The *GUI Ripping Engine* constructs a graph containing all of the possible transition states and uses the back button to return to previous states after the executable components in a particular branch have been exhausted. It also keeps a stack of all the yet-to-be visited components. To detect and capture exceptions, CRASHSCOPE filters the logcat for uncaught exceptions related only to the app being tested. To detect crashes, CRASHSCOPE checks for the appereance of the standard Android crash dialog. If a crash is encountered, the execution information is logged to the database, but because of the transition diagram and stack of unvisited components, execution can continue towards additional remaining program paths without starting the execution from scratch.

C. Testing Apps in Different Contextual States

When the GUI-Ripping begins, CRASHSCOPE first checks for app-level contextual features that should be tested ac-

cording to the exploration strategy. Then, the *GUI Ripping Engine* checks if the current Activity is suitable for exercising a particular contextual feature in adverse conditions. If this is the case, it sets the value of the sensor according to the current strategy. The testing of contextual features works *only on* emulators using telnet commands associated with standard Android Virtual Devices (AVDs) [2]. While the telnet commands do support turning on/off the network for an emulator, they do not support the enabling/disabling of sensors (Accelerometer, Magnetometer, GPS, Temperature Sensor), but it is possible to set the values of these sensors. Therefore, to test for sensor related features in adverse conditions, the network connection is disabled, and unexpected values are set for the other sensors (GPS, Accelerometer, etc) that would not typically be possible under normal conditions. For instance, to test the GPS in an adverse contextual state, CRASHSCOPE sets the value to coordinates that do not represent physical GPS coordinates.

D. Multiple Execution Strategies

One of CRASHSCOPE'S most powerful features is its ability to explore an app according to several different strategies through combinations of its various supported testing features. These strategies stem from three major feature heuristics: 1) the direction in which to traverse the GUI Hierarchy (top-down or bottom-up), 2) the method by which inputs are generated for user text entry fields (*no text*, *expected text*, *unexpected text*), and finally, 3) enabling or disabling the testing of adverse contextual states (e.g., if an activity is found to have utilize wifi, should it be turned on or off?). Different combinations of these strategies have the potential to uncover different types of app crashes. For example, consider the following configuration `<no_text, top_down, enable_all_context_states>`. According to this strategy, CRASHSCOPE will not enter any user text, will exercise the GUI-components in order from the top of the screen to the bottom, and will trigger adverse contextual features in activities where they are detected. This type of strategy has a high likelihood of uncovering crashes like the one described earlier in C13C Schedule in which the change of contextual state triggers a crash. However, the `<unexpected_text, top_down, disable_context_states>` has a better chance of uncovering crashes related to user input being handled improperly by the app. By running an app through all 12 combinations of these three feature heuristics in different strategies, CRASHSCOPE can effectively test for different types of commonly inducible crashes. These strategies can also be parallelized by running several strategies for an app concurrently on a group or cloud of emulator instances, further reducing the testing overhead for the developer.

E. Generating Expressive, Natural Language Crash Reports

CRASHSCOPE generates a Crash Report (Figure 1-⑤) that contains four major types of information: 1) general information including the app name and version, the version of the Android OS, a legend of icons that indicate the current contextual state of the app in the reproduction steps, the device, and the screen orientation and resolution when the

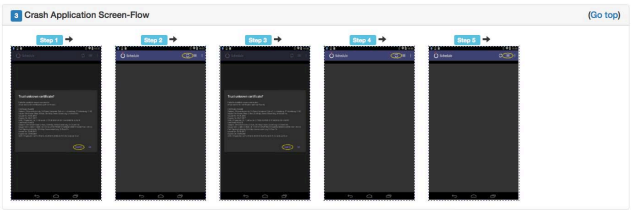

Fig. 2. Crash Screen-Flow

crash occurred; 2) natural language sentences that describe the steps to reproduce a crash using detailed information about the GUI events and contextual states for each step (Figure 3); 3) an app's screen flow that highlights the component interacted with on each screen in the execution scenario for a particular crash (Figure 2); (4) a pruned stack trace containing only the app exceptions that occurred during execution.

The natural language reproduction steps are constructed by the *Report Generator* (Figure 1-④) using the template:

> `<action>` *on* `<component text> <component type>`, *which is located on the* `<relative location>` *of the screen*

For the steps that have text entry associated with them, the `<action>` placeholder is modified into the following: "`Type <text input>` on the..." so as to capture any specific text inputs that may trigger a crash.

F. Generating & Replaying Reproduction Scripts

The *Crash Script Generator* (Figure 1-⑥), parses the saved execution information from the CRASHSCOPE database and generates replayable scripts containing `adb input` commands for touch and text inputs and markers for changes in contextual states. The scripts are generated by parsing the database for all of the GUI events associated with each step in a particular execution. Then, the coordinates of each component that were recorded during the systematic exploration of the app are parsed and the center coordinates are extrapolated based on each component's size. These coordinates are used to generate `adb input` commands to reproduce the GUI event. This approach relies on our previous work in replaying events of test sequences in Android apps [47], [50]. An example of a CRASHSCOPE replayable script can be seen in Fig. 1-⑥. The scripts can be replayed by the *Script Replayer* (Fig. 1-⑦), which executes the `adb input` commands, and interprets the state change markers in the script (e.g.,⟨Wifi_OFF⟩) to execute proper *telnet* commands to set states on an emulator.

IV. EMPIRICAL STUDY 1: CRASH DETECTION CAPABILITY

The *goal* of our first study is to evaluate the effectiveness of CRASHSCOPE at discovering crashes in Android apps as compared to state-of-the-art approaches for testing mobile apps. The *quality focus* of this first study concerns the fault detection capabilities of CRASHSCOPE in terms of locating crashes. The *context* of this study consists of 61 open-source Android apps previously used to evaluate automated testing approaches in [29], as well as five approaches for automated input generation (listed in Table II). We investigated the following research questions (RQs):

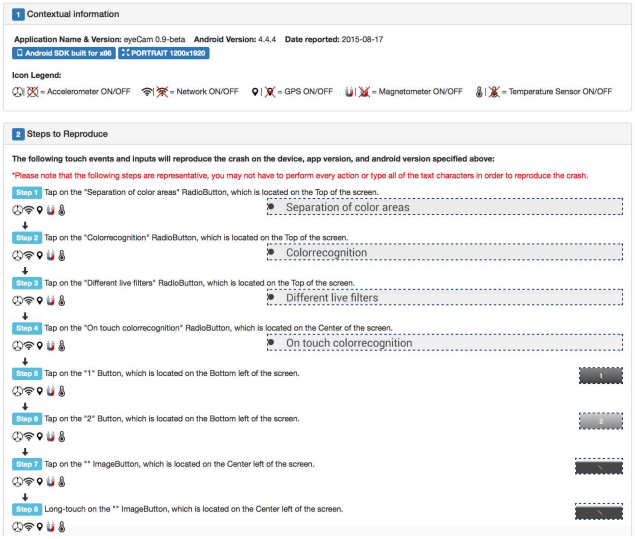

Fig. 3. Example of Contextual Information and Reproduction Steps sections in a generated crash report

- **RQ**$_1$: *What is* CRASHSCOPE'S *effectiveness in terms of detecting application crashes compared to other state-of-the-art Android testing approaches?*
- **RQ**$_2$: *Does* CRASHSCOPE *detect different crashes compared to the other tools?*
- **RQ**$_3$: *Are some* CRASHSCOPE *execution strategies more effective at detecting crashes or exceptions than others?*
- **RQ**$_4$: *Does average application statement coverage correspond to a tool's ability to detect crashes?*

A. Methodology

In order to compare CRASHSCOPE against other state-of-the-art automated input generation tools for Android, we utilized a subset of subject apps and tools available in the Androtest testing suite [8], [29]. We chose to perform this study on a subset of the tools offered by the Androtest artifact due to runtime issues, namely, some tools would not run consistently on the set of provided subject apps (e.g., the tools would launch an emulator but not the app), causing inconsistent results we chose to exclude. However, when contacted, the authors of the tool were helpful in supporting us. We believe the tools tested against constitute a diverse representation of the publicly available Android testing tools. The Androtest suite contains 68 subject applications for testing; however, when recompiling the applications to run the tools and extract the apps from the VM to run with CRASHSCOPE, seven of the subject apps failed to compile with the instrumentation necessary to gather code-coverage results. Therefore, each tool in the suite was allowed to run for one hour for each of the remaining 61 subject apps, five times, whereas we ran all 12 combinations of the CRASHSCOPE strategies once on each of these apps. It is worth noting that the execution of tools in the Androtest suite (except for Android monkey) can not be controlled by a criteria such as maximum number of events.

In the Androtest VMs, each tool ran on its required Android version, for CRASHSCOPE each subject application was run on an emulator with a 1200x1920 display resolution, 2GB of RAM, a 200 MB Virtual sdcard, and Android version 4.4.2 KitKat. We ran the tools listed in Table II, except Monkey, using Vagrant [14] and VirtualBox [15]. The Monkey tool was run for 100-700 event sequences (in 100 event deltas for seven total configurations) on an emulator with the same settings as above with a two-second delay between events, discarding trackball events. Trackball events were discarded to facilitate a fair comparison to the supported events and hardware of the other testing tools. Each of these seven configurations was executed five times for each of the 61 subject apps, and every execution was instantiated with a different random seed [7]. While Monkey is an available tool in Androtest, the authors of the tool chose to set no delay between events, meaning the number of events monkey executed over the course of 1 hour far exceeds the number of events generated by the other tools, which would have resulted in a biased comparison to CRASHSCOPE and the other automated testing tools. To facilitate a fair comparison, we chose to limit the number of events thrown by Android monkey to a range (100-700 events) that corresponds to the average number of events invoked by other tools. In order to give a complete picture of the effectiveness of CRASHSCOPE as compared to the other tools, we report data on both the statement coverage of the tools as well as crashes detected by each tool. Each of the subject applications in the Androtest suite was instrumented with the Emma code coverage tool [9], and we used this instrumentation to collect statement coverage data for each of the apps. Due to space limitations, we report the cumulative coverage for all of the strategies and runs of each tool with a full dataset of detailed statistics available in our replication package in the online appendix [58].

TABLE II
TOOLS USED IN THE COMPARATIVE FAULT FINDING STUDY

Tool Name	Android Version	Tool Type
Monkey	any	Random
A^{3}E Depth-First	any	Systematic
GUIRipper	any	Model-Based
Dynodroid	v2.3	Random-Based
PUMA	v4.1+	Random-Based

The underlying purpose of this study is to compare the crash detection capabilities of each of these tools and answer **RQ**$_1$. However, we cannot make this comparison in a straightforward manner. CRASHSCOPE is able to accurately detect app crashes by detecting the standard Android dialog for exposing a crash (e.g., a text box containing the phrase "application_name *has stopped*"). However, because the other analyzed tools do not support identifying crashes at runtime, there is no reliable automated manner to extract instances where the application crashed purely from the `logcat` [5]. To obtain an approximation of the crashes detected by these tools, we parsed the `logcat` files generated for each tool in the Androtest VMs. Then, we isolated instances where exceptions occurred containing the `FATAL EXCEPTION` key marker, which were also associated with the process id (pid) of the app running during the `logcat` collection. While this filters out unwanted exceptions from the OS and other processes, unfortunately, it does not guarantee that the exceptions signify a crash caused by incorrect application logic. This could signify, among other

TABLE III
UNIQUE CRASHES DISCOVERED WITH INSTR. CRASHES IN PARENTHSES

App	A^{3}E	GUI-Ripper	Dyno-droid	PUMA	Monkey (All)	Crash-Scope
A2DP Vol	1	0	0	0	0	0
aagtl	0	0	1	0	1	0
Amazed	0	0	0	0	1	0
HNDroid	1	1	1	2	1	1
BatteryDog	0	0	1	0	1	0
Soundboard	0	1	0	0	0	0
AKA	0	0	0	0	1	0
Bites	0	0	0	0	1	0
Yahtzee	1	0	0	0	0	1
ADSDroid	1	1	1	1	1	1
PassMaker	1	0	0	0	1	1
BlinkBattery	0	0	0	0	1	0
D&C	0	0	0	0	1	0
Photostream	1	1	1	1	1	0
AlarmKlock	0	0	1	0	0	0
Sanity	1	1	0	0	0	0
MyExpenses	0	0	1	0	0	0
Zooborns	0	0	0	0	0	2
ACal	1	2	2	0	1	1
Hotdeath	0	2	0	0	0	1
Total	**8 (21)**	**9 (5)**	**9 (6)**	**4 (0)**	**12 (1)**	**8 (0)**

things, a crash caused by the instrumentation of the controlling tool. Therefore, in order to conduct a consistent comparison to CRASHSCOPE, the authors manually inspected the instances of fatal exception stack traces returned by the `logcat` parsing, discarding duplicates and those caused by instrumentation problems, and we report the crash results of the other tools from this pruned list. A full result set with both full and pruned `logcat` traces is available in our online appendix [58]. The issues encountered when parsing the results from these other tools further highlight CRASHSCOPE'S utility, and the need for an automatic tool that can accurately detect and in turn effectively report crashes in mobile apps.

B. Results & Discussion

Table III shows the aggregated crash discovery results of each tool over their various runs. This table reports unique crashes (as signified by differing stack traces not caused by app instrumentation) detected by the various approaches, and only includes those apps for which crashes were discovered. For tools other than CRASHSCOPE, we also report crashes (in parentheses) that were caused by instrumentation frameworks (e.g. troyd, Android intsr., junit, Emma), as these represent "false positive" crashes uncovered by the tools. The results highlight four key results. The first observable result is that CRASHSCOPE is about as effective in terms of number of crashes detected, while also providing detailed bug reports. CrashScope discovered fewer crashes compared to Monkey due to the large number of events that this tool is capable of producing. However, it should be noted that Monkey is not able to generate replayable scripts or reports, severely limiting its usefulness form a developers perspective. CRASHSCOPE was able to discover about as many crashes as A^{3}E, GUI-Ripper, and Dynodroid, more than PUMA, without any false positives caused by instrumentation of the app or system. Therefore, we answer $\mathbf{RQ}_1$ as follows: **CrashScope is about as effective at detecting crashes as the other tools. Furthermore, our approach reduces burden on developers by reducing the number of "false" crashes caused by instrumentation and providing detailed crash reports**.

The second observable result is that CRASHSCOPE is able to detect orthogonal crashes compared to the other tools. In order

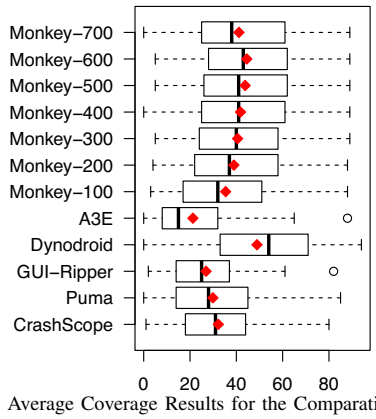

Fig. 4. Average Coverage Results for the Comparative Study

to understand why CRASHSCOPE detected different crashes than the other approaches, the authors manually examined the detected crash reports to determine their causes. Because it might not be possible to determine the exact cause or type of crashes from the other tools, we exclude a discussion here, but we speculate on the differences from CRASHSCOPE'S results. The key finding from this exploration is that *the differing strategies implemented by CrashScope contributed to its ability to detect orthogonal crashes compared to the other tools.* For instance, the crash detected by CRASHSCOPE for the *zooborns* app is triggered by typing unexpected text in a text box. The other tools probably missed this crash because their text generation techniques do not include unexpected inputs. Furthermore, one aspect of this crash highlights the utility of CRASHSCOPE's detection and reporting capabilities, namely, the thrown exception is potentially misleading to a developer. While this crash was caused by text formatting, the exception is for an AsyncTask object, one of Android's thread handling mechanisms, meaning it could be difficult for a developer to reason about the cause of this crash in the absence of a detailed report. Another example of an orthogonal crash discovered by CRASHSCOPE is that for the *PasswordMakerPro* app. While two other tools (Monkey, A^{3}E) found a crash during their exploration of this app, only CrashScope was able to discover a crash caused by a contextual feature, rotation. This highlights the utility of the different exploration techniques. Consequently, $\mathbf{RQ}_2$ can be answered as follows: **The varying strategies of CrashScope allow the tool to detect different crashes compared to those detected by other approaches.**

The third result we see from the the crash detection data is that certain CRASHSCOPE strategies are more effective at uncovering crashes than others. The most effective of the text strategies overall was the *unexpected* heuristic where all of the crashes listed for CRASHSCOPE in Table III were uncovered, but not directly triggered by, utilizing this type of text input. While the different strategies achieved similar overlapping coverage on average, different crashes were discovered during the runs of strategies where contextual features were and were not tested in adverse conditions, as discussed above, suggesting that some errors are only discoverable when contextual features are in normal states. Overall, the *forwards* heuristic for traversing the GUI led to the discovery of more crashes (8 crashes) compared to the *backwards* strategy (7

crashes), with 7 of these crashes overlapping. It should be noted that the GUI-traversal heuristic did not directly trigger any crashes directly (e.g. the changing the order of interacting with components did not lead to crashes), however, these two strategies were useful for exploring different parts of the subject applications. The most effective overall crash discovery strategy was *<contextual_feautres_enabled, forward, unexpected>*. Full coverage and crash results for all the tools, and all 12 of CRASHSCOPE'S strategies on a per app basis are available in our online appendix [58]. Thus, $\mathbf{RQ}_3$ can be answered: **Different combinations of CrashScope strategies were more effective than others, suggesting the need for multiple testing strategies encompassed within a single tool with an emphasis on strategies for contextual features.**

The fourth observable result is that the average statement coverage of the analyzed tools (see Fig. 4) does not necessarily correspond to a better fault discovery capability, as CRASHSCOPE was able to detect about as many crashes with lower average coverage than other tools (i.e., PUMA, Monkey, and Dynodroid). This implies that future testing approaches for mobile apps need to take into consideration other metrics in addition to code coverage to illustrate the effectiveness of the approach. Therefore, our answer for $\mathbf{RQ}_4$ is: **Higher statement coverage of an automated mobile app testing tool does not necessarily imply that tool will have effective fault-discovery capabilities**.

V. STUDY 2: REPRODUCIBILITY & READABILITY

The *goal* of the second study is to evaluate the *reproducibility* and *readability* of the natural language reports generated by CRASHSCOPE compared to original human written reports found in online issue trackers. The *quality focus* of this study concerns the ability of developers to reproduce bugs from CRASHSCOPE's reports. The *context* of this study consists of eight real world Android app crashes and reports, extracted from open source apps and their corresponding issue trackers, as well as reports generated by CRASHSCOPE for these same crashes (details of the crashes and corresponding apps are presented in our online appendix [58]). In the context of this second study we examined the following RQs:

- $\mathbf{RQ}_5$: *Are reports generated with* CRASHSCOPE *more reproducible than the original human written reports?*
- $\mathbf{RQ}_6$: *Are reports generated by* CRASHSCOPE *more readable than the original human written reports?*

A. Methodology

To identify the crashes used for this study, we manually inspected the issue trackers of the apps on F-droid looking for reports that described an app crash. Then, we ran CRASHSCOPE on the version of the app that the crash was reported against to observe whether or not CRASHSCOPE was able to capture the crash on the same emulator configuration as the previous study. While we chose these bugs manually, the goal of this study is not to measure CRASHSCOPE's effectiveness at discovering bugs (unlike the first study). We acknowledge that there are situations in which CRASHSCOPE will not be able to detect a fault and we outline these cases in Sec. VI.

In order to answer $\mathbf{RQ}_5$ and $\mathbf{RQ}_6$, we asked 16 CS graduate students from William and Mary (a proxy for developers [67]) to reproduce the eight crashes (four from the original human written reports, and four from CRASHSCOPE). The design matrix of this study was devised in such as way that each crash for each type of report was evaluated by four participants, no crash was evaluated twice for the same participant, and eight participants saw the human written reports first, and eight participants saw the CRASHSCOPE reports first, all in the interest of reducing bias. The system names were also anonymized (CRASHSCOPE to "System A" and the human written reports to "System B"). The full design matrix can be found in our online appendix [58]. During the study, participants recorded the time it took them to reproduce the crash on a Nexus 7 device for each report, with a time limit of ten minutes for reproduction. If a participant could not reproduce the bug within the ten minute time frame or gave up in trying to reproduce the bug, that bug was marked as non-reproducible for that participant. To mitigate the "flaky-test" problem, where outstanding factors such as Network I/O, varying sensor readings or app delay could cause difficulty of crash reproduction, when manually selecting the crashes and crash reports from the online repositories, the authors ensured that each bug was deterministically reproducible within the confines of the study environment (e.g. Using the proper version of the application that contains the bug and that the bug was always reproducible on the Nexus 7 tablet). Therefore, in order to answer $\mathbf{RQ}_5$, we measured how many crashes were successfully reproduced by the participants for each type of crash report, we also measured the time it took each participant to reproduce each bug (the detailed dataset is available at [58]).

After the completion of the crash reproductions, we had each participant fill out a brief survey, answering questions regarding the *user preferences* (***UP***) and *usability* (***UX***) for each type of bug report. We also collected information about each participants programming experience and familiarity with the Android platform. The ***UP*** questions were formulated based on the user experience honeycomb originally developed by Moville [61] and were posed to participants as free form text entry questions. We forgo a discussion of the free-form question responses due to space limitations, but offer full anonymized participant responses at [58]. The ***UX*** questions were created using statements based on the SUS usability scale by Brooke [25] and were posed to participants in the form of a 5-point Likert scale. We quantify the *user experience* of CRASHSCOPE and answer $\mathbf{RQ}_6$ by presenting the mean and standard deviation of the scores for the responses to the Likert-based questions. The questions regarding programming experience are based on the well-accepted questionnaire developed by Feigenspan *et al.* [34].

B. Results & Discussion

The CRASHSCOPE reports achieved a similar levels of reproducibility compared to the human written reports with 94% (60 out of 64) of the CRASHSCOPE reports being successfully reproduced by participants compared to 92% (59 out of 64) of the original reports. Therefore, $\mathbf{RQ}_5$ can be

TABLE IV
USER EXPERIENCE RESULTS: THIS TABLE REPORTS THE MEAN AVERAGE RESPONSE FROM 16 USERS REGARDING THE USER EXPERIENCE QUESTIONS POSED FOR BOTH CRASHSCOPE GENERATED REPORTS AND THE ORIGINAL HUMAN WRITTEN REPORTS FOUND IN THE APP'S ISSUE TRACKERS.

Question	CrashScope Mean	CrashScope StdDev	Original Mean	Original StdDev
UX1: I think I would like to have this type of bug report frequently.	4.00	0.89	3.06	0.77
UX2: I found this type of bug report unnecessarily complex.	2.81	1.04	2.125	0.96
UX3: I thought this type of bug report was easy to read/understand.	4.00	0.82	3.00	0.97
UX4: I found this type of bug report very cumbersome to read.	2.50	1.10	2.44	0.81
UX5: I thought the bug report was really useful for reproducing the crash.	4.13	0.62	3.44	0.89

answered as follows: **Reports generated by CrashScope are about as reproducible as human written reports extracted from open-source issue trackers**. Due to space limitations, full numerical statistics for Study 2, including time-cost to reproduce crashes, and detailed participant answers, can be accessed in our online appendix [58]. The UX questions and results can be found in Table IV, which show that participants found CRASHSCOPE reports to be more readable and useful than the original reports. Thus, $\mathbf{RQ}_6$ can be answered as: **Reports generated by CrashScope are more readable and useful from a developers' perspective as compared to human written reports**. One interesting case arose from this study. No participant assigned the original report for the C13C Schedule app was able to reproduce the bug, whereas all participants assigned the CRASHSCOPE version of this app were able to reproduce it. This is because the network needed to be disabled for the crash to manifest itself, and this was not captured in the original bug report. This highlights the utility of CRASHSCOPE'S context-aware reports.

VI. LIMITATIONS & THREATS TO VALIDITY

While our empirical evaluation has shown that CRASHSCOPE is effective at detecting crashes in Android apps, our tool has some inherent limitations. *First*, because CRASHSCOPE's systematic execution engine does not implement the swipe gesture, it will not be able to execute GUI components existing within a list that does not fit entirely within the device's screen. This limitation may cause some crashes or exceptions dependent on these types of components to be missed. The *second* limitation is that CRASHSCOPE does not support highly specialized text input. This may limit the exploration capabilities of our tool for certain apps. However, recent approaches in concolic and symbolic executions may prove useful in overcoming this limitation [38], [55], [56], [70], [77]. The *third* limitation of our tool relates to window detection in Android. Android apps are organized into screens based on activities and other windows (e.g., dialogs). Activities are fairly simple to detect, as each has a unique name which acts as an identifier for that activity. However, the same is not true for dialogs, as they have no unique identifier. Each Activity can have multiple dialogs. To solve this problem we use the size of the window with the focus and in foreground as a unique identifier, as through our observations we found that very few activities employ different unique windows of the same size. However, this is an imperfect heuristic and prone to occasional errors. Due to checks in place in our systematic execution algorithm, this never leads to incorrect execution of the app, however, it may mean that less functionality of the app is explored compared to a method that is able to correctly identify all unique windows in an app.

One potential threat to external validity is the fact that we used a set of 61 open source applications to evaluate CRASHSCOPE in the first empirical study, and eight crashes in eight open source applications for the second empirical study. Therefore, we can not generalize our results to Android apps in general due to the limitations of these subject apps. However, we believe that this threat is lessened by the fact that these apps were collected from datasets in previous studies and contain several popular, complex apps. In the context of our empirical studies, one threat to internal validity stem from the potentially surprising effects of participants in the second empirical study. To this end, there is a threat since we approximated graduate students in Computer Science as experienced Android developers. However, this threat is mitigated by the fact that all of these participants indicated that they have extensive programming experience as well as moderate experience with the Android environment, and recent work shows that in carefully controlled experiments experienced graduate students are sufficient proxy's for developers [67]. Another threat to internal validity concerns the manual inspection of log traces from the tools CRASHSCOPE was tested against. However, this threat is mitigated as the process was partially automated to decrease the manual examination set and the authors who examined these logs are well versed in the Android platform and automated testing approaches in research.

VII. CONCLUSIONS

In this paper, we present CRASHSCOPE, a practical approach for discovering, reporting, and replaying Android app crashes. Our tool leverages a powerful algorithm for systematic exploration that is crash resilient, capable of context-aware input and text generation, and runs on a diverse set of devices and emulators. We evaluated CRASHSCOPE with respect to crash and exception detection, as compared to other state-of-the-art automatic input generation tools for Android and show that our tool is able to uncover about as many crashes as these other approaches, while offering more detailed information in the form of NL crash reports containing steps to reproduce the crash, and high-level repayable traces that can reproduce the crash on demand. We also evaluated the *reproducibility* and *readability* of our automatically generated reports and show that they provide for reliable reproduction of crashes while proving more readable and usable for developers. In the future, we aim to investigate techniques to trim bug reports, so that they contain only the necessary steps, as well as improving our systematic exploration strategy for uncovering a higher number of bugs, by adapting promising emerging approaches in model-based GUI testing [62].

ACKNOWLEDGEMENTS

This work is supported in part by the NSF CCF-1218129 and NSF CCF-1525902 grants. We would like to thank the anonymous reviewers for their insightful comments that significantly improved this paper and the authors of the Androtest benchmark tools [29] for their aid in reproducing the results.

REFERENCES

[1] 31c3 schedule application https://github.com/tuxmobil/CampFahrplan.

[2] Android emulator documentation http://developer.android.com/tools/help/emulator.html.

[3] Android fragmentation statistics http://opensignal.com/reports/2014/android-fragmentation/.

[4] Android inputtype specifications https://developer.android.com/reference/android/widget/TextView.html#attr_android:inputType.

[5] Android logcat debugging tool http://developer.android.com/tools/help/logcat.html.

[6] Android platform install base information https://developer.android.com/about/dashboards/index.html.

[7] Android ui/application exerciser monkey http://developer.android.com/tools/help/monkey.html.

[8] Androtest framework http://bear.cc.gatech.edu/~shauvik/androtest/.

[9] Emma code coverage tool http://emma.sourceforge.net.

[10] Google play store https://play.google.com/store?hl=en.

[11] Intent fuzzer https://www.isecpartners.com/tools/mobile-security/intent-fuzzer.aspx.

[12] Mobile apps: What consumers really need and want https://info.dynatrace.com/rs/compuware/images/Mobile_App_Survey_Report.pdf.

[13] Soot java instrumentation framework http://sable.github.io/soot/.

[14] Vagrant virtualbox manager https://docs.vagrantup.com/v2/.

[15] Virtualbox https://www.virtualbox.org.

[16] C. Q. Adamsen, G. Mezzetti, and A. Møller. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 83–93, New York, NY, USA, 2015. ACM.

[17] D. Amalfitano, A. Fasolino, P. Tramontana, B. Ta, and A. Memon. Mobiguitar – a tool for automated model-based testing of mobile apps. *Software, IEEE*, PP(99):1–1, 2014.

[18] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.

[19] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 59:1–59:11, New York, NY, USA, 2012. ACM.

[20] 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*, OOPSLA '13, pages 641–660, New York, NY, USA, 2013. ACM.

[21] G. Bavota, M. Linares-Vásquez, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. The impact of api change- and fault-proneness on the user ratings of android apps. *Software Engineering, IEEE Transactions on*, 41(4):384–407, April 2015.

[22] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 308–318, New York, NY, USA, 2008. ACM.

[23] 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*, MSR '08, pages 27–30, New York, NY, USA, 2008. ACM.

[24] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann. Information needs in bug reports: Improving cooperation between developers and users. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, CSCW '10, pages 301–310, New York, NY, USA, 2010. ACM.

[25] J. Brooke. SUS: A quick and dirty usability scale. In P. W. Jordan, B. Weerdmeester, A. Thomas, and I. L. Mclelland, editors, *Usability evaluation in industry*. Taylor and Francis, London, 1996.

[26] R. Chandra, B. F. Karlsson, N. D. Lane, C.-J. M. Liang, S. Nath, J. Padhye, L. Ravindranath, and F. Zhao. How to the smash next billion mobile app bugs? *GetMobile: Mobile Comp. and Comm.*, 19(1):34–38, June 2015.

[27] N. Chen, J. Lin, S. Hoi, X. Xiao, and B. Zhang. AR-Miner: Mining informative reviews for developers from mobile app marketplace. In *36th International Conference on Software Engineering (ICSE'14)*, page To appear, 2014.

[28] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 623–640, New York, NY, USA, 2013. ACM.

[29] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*, 2015.

[30] C. Csallner and Y. Smaragdakis. Jcrasher: An automatic robustness tester for java. *Softw. Pract. Exper.*, 34(11):1025–1050, Sept. 2004.

[31] C. Csallner, Y. Smaragdakis, and T. Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2):8:1–8:37, May 2008.

[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*, ICSE '12, pages 1084–1093, Piscataway, NJ, USA, 2012. IEEE Press.

[33] M. Erfani Joorabchi, M. Mirzaaghaei, and A. Mesbah. Works for me! characterizing non-reproducible bug reports. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 62–71, New York, NY, USA, 2014. ACM.

[34] J. Feigenspan, C. Kastner, J. Liebig, S. Apel, and S. Hanenberg. Measuring programming experience. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 73–82, June 2012.

[35] Z. Gu, E. Barr, D. Hamilton, and Z. Su. Has the bug really been fixed? In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 55–64, May 2010.

[36] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 204–217, New York, NY, USA, 2014. ACM.

[37] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, effectively detecting mobile app bugs with appdoctor. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 18:1–18:15, New York, NY, USA, 2014. ACM.

[38] J. Jaffar, V. Murali, and J. A. Navas. Boosting concolic testing via interpolation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 48–58, New York, NY, USA, 2013. ACM.

[39] W. Jin and A. Orso. Bugredux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 474–484, Piscataway, NJ, USA, 2012. IEEE Press.

[40] W. Jin and A. Orso. F3: Fault localization for field failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 213–223, New York, NY, USA, 2013. ACM.

[41] M. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*, pages 15–24, Oct 2013.

[42] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan. What do mobile App users complain about? a study on free iOS Apps. *IEEE Software*, (2-3):103–134, 2014.

[43] 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*, pages 486–493, June 2011.

[44] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. Understanding the test automation culture of app developers. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification, and Validation*, April 2015.

[45] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, MobiCom '14, pages 519–530, New York, NY, USA, 2014. ACM.

[46] Y. Lin, J. F. Rojas, E. Chu, and Y. Lai. On the accuracy, efficiency, and reusability of automated test oracles for android devices. *IEEE Transactions on Software Engineering (TSE)*, (99):1–1, 2014.

[47] M. Linares-Vásquez. Enabling testing of android apps. In *IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, volume 2, pages 763–765, May 2015.

[48] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. Api change and fault proneness: A threat to the success of android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 477–487, New York, NY, USA, 2013. ACM.

[49] M. Linares-Vásquez, C. Vendome, Q. Luo, and D. Poshyvanyk. How developers detect and fix performance bottlenecks in android apps. In *Proceedings of 31st IEEE International Conference on Software Maintenance and Evolution (ICSME'15)*, page to appear, 2015.

[50] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. Mining android app usages for generating actionable gui-based execution scenarios. In *12th Working Conference on Mining Software Repositories (MSR'15)*, page to appear, 2015.

[51] 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*, ESEC/FSE-13, pages 286–295, New York, NY, USA, 2005. ACM.

[52] G. Lohman, J. Champlin, and P. Sohn. Quickly finding known software problems via automated symptom matching. In *Proceedings of the Second International Conference on Automatic Computing*, ICAC '05, pages 101–110, Washington, DC, USA, 2005. IEEE Computer Society.

[53] 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*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[54] 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*, FSE 2014, pages 599–609, New York, NY, USA, 2014. ACM.

[55] R. Majumdar and K. Sen. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[56] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek. Sig-droid: Automated system input generation for android applications. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 461–471, Nov 2015.

[57] K. Moran. Enhancing android application bug reporting. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 1045–1047, New York, NY, USA, 2015. ACM.

[58] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk. Crashscope online appendix http://www.cs.wm.edu/semeru/data/ICST16-CrashScope/.

[59] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk. Auto-completing bug reports for android applications. In *ESEC/FSE'15*, page to appear, Bergamo, Italy, 2015.

[60] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk. Fusion: A tool for facilitating and augmenting android bug reporting. In *in Proceedings of 38thACM/IEEE International Conference on Software Engineering (ICSE'16)*, May 2016.

[61] P. Morville. User experience design. http://semanticstudios.com/user_experience_design/.

[62] B. Nguyen and A. Memon. An observe-model-exercise; paradigm to test event-driven systems with undetermined input spaces. *Software Engineering, IEEE Transactions on*, 40(3):216–234, March 2014.

[63] 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*, ECOOP'05, pages 504–527, Berlin, Heidelberg, 2005. Springer-Verlag.

[64] 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*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[65] F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. User reviews matter! tracking crowdsourced reviews to support evolution of successful apps. In *Proceedings of 31st IEEE International Conference on Software Maintenance and Evolution (ICSME'15)*, page to appear, 2015.

[66] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 190–203, New York, NY, USA, 2014. ACM.

[67] I. Salman, A. T. Misirli, and N. Juristo. Are students representatives of professionals in software engineering experiments? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 666–676, Piscataway, NJ, USA, 2015. IEEE Press.

[68] R. Sasnauskas and J. Regehr. Intent fuzzer: Crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, WODA+PERTEA 2014, pages 1–5, New York, NY, USA, 2014. ACM.

[69] H. Seo and S. Kim. Predicting recurring crash stacks. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 180–189, New York, NY, USA, 2012. ACM.

[70] H. Seo and S. Kim. How we get there: A context-guided search strategy in concolic testing. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 413–424, New York, NY, USA, 2014. ACM.

[71] P. Tonella, R. Tiella, and C. D. Nguyen. Interpolated n-grams for model based testing. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 562–572, New York, NY, USA, 2014. ACM.

[72] H. van der Merwe, B. van der Merwe, and W. Visser. Execution and property specifications for jpf-android. *SIGSOFT Softw. Eng. Notes*, 39(1):1–5, Feb. 2014.

[73] M. White, M. Linares-Vásquez, P. Johnson, C. Bernal-Cárdenas, and D. Poshyvanyk. Generating reproducible and replayable bug reports from android application crashes. In *23rd IEEE International Conference on Program Comprehension*, 2015.

[74] 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*, ISSTA 2014, pages 204–214, New York, NY, USA, 2014. ACM.

[75] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated gui-model generation of mobile applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, FASE'13, pages 250–265, Berlin, Heidelberg, 2013. Springer-Verlag.

[76] H. Ye, S. Cheng, L. Zhang, and F. Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, MoMM '13, pages 68:68–68:74, New York, NY, USA, 2013. ACM.

[77] C. C. Yeh, H. L. Lu, C. Y. Chen, K. K. Khor, and S. K. Huang. Craxdroid: Automatic android system testing by selective symbolic execution. In *Proceedings of the 2014 IEEE Eighth International Conference on Software Security and Reliability-Companion*, SERE-C '14, pages 140–148, Washington, DC, USA, 2014. IEEE Computer Society.

[78] S. Yoo, M. Harman, and D. Clark. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Trans. Softw. Eng. Methodol.*, 22(3):19:1–19:29, July 2013.

[79] R. N. Zaeem, M. R. Prasad, and S. Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 183–192, Washington, DC, USA, 2014. IEEE Computer Society.

[80] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 1–10, New York, NY, USA, 2002. ACM.

[81] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[82] 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*, ICSE '12, pages 14–24, Piscataway, NJ, USA, 2012. IEEE Press.