FUSION: A Tool for Facilitating and Augmenting Android Bug Reporting

Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Denys Poshyvanyk

College of William & Mary
Department of Computer Science
Williamsburg, VA 23187-8795, USA
{kpmoran, mlinarev, cebernal, denys}@cs.wm.edu

ABSTRACT

As the popularity of mobile smart devices continues to climb the complexity of "apps" continues to increase, making the development and maintenance process challenging. Current bug tracking systems lack key features to effectively support construction of reports with actionable information that directly lead to a bug's resolution. In this demo we present the implementation of a novel bug reporting system, called FU-SION, that facilitates users including reproduction steps in bug reports for mobile apps. FUSION links user-provided information to program artifacts extracted through static and dynamic analysis performed before testing or release. Results of preliminary studies demonstrate that FUSION both effectively facilitates reporting and allows for more reliable reproduction of bugs from reports compared to traditional issue tracking systems by presenting more detailed contextual app information. Tool website: `www.fusion-android.com` Video url: `https://youtu.be/AND9hOElxRg`

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Experimentation, Design

Keywords

Bug reports, android, reproduction steps, auto-completion

1. INTRODUCTION

It is clear that as smart device usage reaches ubiquitous levels (e.g., 2.7 billion active smartphone users in 2014[20]), developers need tools to support them in maintaining high-quality apps. Software maintenance activities are known to be generally expensive and challenging [24] and one of the most important maintenance tasks is bug report resolution. However, current bug tracking systems such as Bugzilla [3], Mantis [10], the Google Code Issue Tracker [7], the GitHub Issue Tracker [6], and commercial solutions such as JIRA

[9] rely mostly on unstructured natural language bug descriptions. While these descriptions can be supplemented with structured information such as reproduction steps or stack traces, and files such as screenshots, the inclusion of such information typically depends on the reporter's experience and attitude towards providing these details. Previous studies have also shown that the information most useful to developers is often the most difficult for reporters to provide and that the lack of this information is a major reason behind non-reproducible bug reports [19, 15]. Therefore, the reporting process can be cumbersome, and the additional effort means that many users are unlikely to enhance their reports with extra information [16, 18, 17, 13].

The above issues point to a more prominent problem for bug tracking systems in general: the *lexical gap* that normally exists between bug reporters (e.g., testers, beta users) and developers. Reporters typically only have functional knowledge of an app, even if they have development experience themselves, whereas the developers working on an app tend to have intimate code level knowledge. When a developer reads and attempts to comprehend (or reproduce) a bug report, she has to bridge this gap, reasoning about the code level problems from the high-level functional description in the bug report. If the lexical gap is too wide the developer may not be able to reproduce and/or subsequently resolve the bug report.

To address this fundamental problem of making bug reports more useful (and reproducible) for developers, this paper presents the implementation of a novel tool, called FU-SION, that facilitates reporters creating detailed bug reports in order to provide more actionable information to developers. FUSION implements the novel approach that was presented and evaluated in our previous work [23]. FUSION first employs fully automated static and dynamic analysis techniques to gather screen-shots and other relevant information about an app before it is released for testing. Reporters then interact with the web-based report generator using the auto-completion features in order to provide the bug reproduction steps. By linking the information provided by the user with features extracted through static and dynamic analyses, FU-SION presents an augmented bug report to the developer that contains actionable information with well-defined steps to reproduce a bug.

2. THE FUSION REPORTING TOOL

FUSION's current target user base consists of two major groups: *mobile application developers* and *beta* users or *testers* . As such there are two user-facing scenarios for the

tool's operation. From a developer's perspective the work-flow is purposefully simple; the only action required is the submission of an `.apk` of the latest build of the application for which they want to enable bug reporting. Once this file is submitted, Fusion's automated program analysis techniques extract the information necessary to facilitate process of creating a bug from the reporters end. Once reports are filled out, developers can access them through a web portal. From a reporter's perspective, they construct a report using a web interface. After selecting the application for which they would like to report a bug, the reporter enters a brief description and contextual information about the bug (e.g. device used, screen orientation). They then use a series of auto-filled combo boxes to construct reproduction steps, including screenshots.

2.1 FUSION Architecture

Fusion's architecture can be seen in Figure 1. First, Fusion collects information related to the GUI components and event flow of an app using the *Static Application Analyzer* and the *Dynamic Analysis Engine*. Then the tool leverages the information collected during the analysis to facilitate a reporter constructing a detailed bug report with reproduction steps, and screenshots. The static and dynamic analyses must be performed before each version of an app is released for testing or before it is published to end users. Both program analysis components store their extracted data in the Fusion database (Fig. 1 - ③).

2.1.1 Static Analysis

The goal of the *Static App Analyzer* (Fig. 1 - ①) is to extract all of the GUI components and associated information from the app source code. For each GUI component, the *Primer* extracts: (i) possible actions on that component, (ii) type of the component (e.g., Button, Spinner), (iii) activities the component is contained within, and (iv) class files where the component is instantiated. Thus, this results in a universe of possible components within the domain of the application, and establishes traceability links connecting GUI-components to code specific information such as the class or `activity` where they are located.

The *Static App Analyzer* utilizes several tools to extract the information outlined above. First it uses the `dex2jar`[4] and `jd-cmd` [8] tools for decompilation; then, it converts the source files to an XML-based representation using `srcML` [11]. We also use `apktool` [2] to extract the resource files from the app's APK. The `ids` and types of GUI components were extracted from the xml files located in the app's resource folders (*i.e.*, `/res/layout` and `/res/menu` of the decompiled application or src). Using the `srcML` representation of the source code, we are able to parse and link the GUI-component information to extracted app source files.

2.1.2 Dynamic Analysis

The *Dynamic Analysis Engine* (Fig. 1 - ②) is used to glean dynamic contextual information and enhance the database with both run-time GUI and application event-flow information. The goal of the *Engine* is to explore an app in a systematic manner, ripping and extracting run-time information related to the GUI components during execution including: (i) the text associated with different GUI components (e.g., the "Send" text on a button to send an email message), (ii) whether the GUI component triggers a tran-
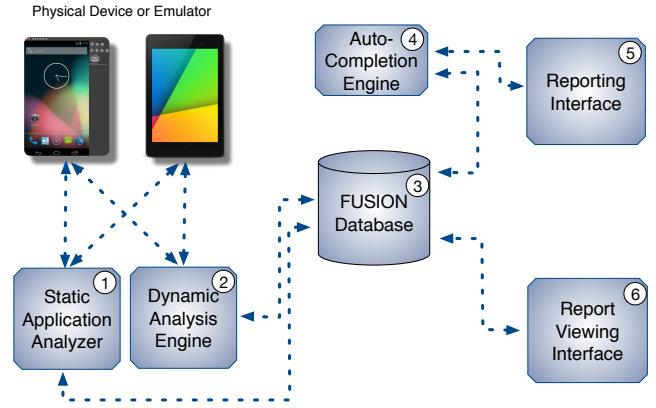

Figure 1: Fusion **Architecture**

sition to a different activity, (iii) the action performed on the GUI component during systematic execution, (iv) full screen-shots before and after each action is performed, (v) the location of the GUI component object on the test device's screen, (vi) the current activity and window of each step, (vii) screen-shots of the specific GUI component, and (viii) the object index of the GUI component (to allow for differentiation between different instantiations of the same GUI component on one screen).

The *Engine* performs this systematic exploration of the app using the `UIAutomator` [1] framework included in the Android SDK. This systematic execution of the app is similar to existing approaches in GUI ripping [12, 14, 22, 21]. Using the `UIAutomator` framework allows us to capture cases that are not captured in previous tools such as pop-up menus that exist within menus, internal windows, and the onscreen keyboard. To effectively explore the application we implemented our own version of a systematic depth-first search (DFS) algorithm for application traversal that performs click events on all the clickable components in the GUI hierarchy reachable using the DFS heuristic.

2.2 Reporting Bugs with FUSION

During the *Report Generation Phase*, Fusion aids the reporter in constructing the steps needed to recreate a bug by making suggestions based upon the "potential" GUI state reached by the declared steps. This means for each step s, Fusion infers — online — the GUI state GUI_s in which the target app should be by taking into account the history of steps. For each step, Fusion verifies that the suggestion made to the reporter is correct by presenting the reporter with contextually relevant screen-shots, where the reporter selects the screen-shot corresponding to the current action she wants to describe.

2.2.1 Report Generator User Interface

After first selecting the app to report an issue for, a reporter interacts with Fusion by filling in some identifying information (i.e., name, device, title) and a brief textual description of the bug in the top half of the UI. Next, the reporter inputs the steps to reproduce the bug using the auto-completion boxes in a step-wise manner, starting from the initial screen of a cold app launch[1], and proceeds until the list of steps to reproduce the bug is exhausted. Accord-

---

[1]Cold-start means the first step is executed on the first window and screen displayed directly after the app is launched.

Figure 2: Fusion **Reporting Interface**

Figure 3: Relative Location Enumeration and Example Augmented Screenshot

First, it differentiates each duplicate component in the list through specifying text "Option #". *Second* Fusion attempts to confirm the component entered by the reporter at each step by fetching screen-shots from the Fusion database representing the entire device screen (e.g., Figure 3).

After the reporter makes selections from the drop-down lists, they have an opportunity to enter additional information for each step (e.g., a button had an unexpected behavior) in a natural language text entry field. For instance, the reporter might indicate that after pressing the "OK" button the pop-up window took longer than expected to disappear.

2.2.2 Report Generator Auto-Completion Engine

The *Auto-Completion Engine* of the web-based report generator (Figure 1-④) uses the information collected up-front during the *Analysis Phase*. When Fusion suggests completions for the drop-down menus, it queries the database for the corresponding state of the app event flow and suggests information based on the past steps that the reporter has entered. Since we always assume a "cold" application start, the *Auto-Completion Engine* starts the reproduction steps entry process from the app's `main` Activity. We then track the reporter's progress through the app using predictive measures based on past steps.

Fusion presents components to the reporter at the granularity of `activities`, or application screens. During the suggestion process, Fusion looks back through the history of reported user interactions and looks for possible transitions from the previous steps to future steps depending on the history of the components interacted with. If Fusion is unable to capture the last few steps from the reporter due to the incomplete application execution model mentioned earlier, then Fusion presents the possibilities from all known screens of the application. Due to the limited nature of the DFS heuristic used by the *Dynamic Analysis Engine*, there may be action-Gui-Component pairs that are not available in the auto-filled combo boxes. To handle these cases gracefully, we allow the reporter to select a special option when they cannot find a component in the auto-completed combo box. When picking this option, the reporter would manually fill in (i) the type of the component, (ii) any text associated with the GUI-component and (iii) the relative location of the GUI-component as denoted in Figure 3.

2.2.3 Viewing FUSION Reports

The *Auto Completion Engine* saves each step to the database as reporters complete bug reports. Once a reporter finishes filling out the steps and completes the data entry process, a screen containing the final report, with an automatically assigned unique ID, is presented to the reporter and saved

ing to the various fields in Figure 2, the reporter would first fill in their (i) *name* (Field 1), (ii) *device* (Field 2), (iii) *screen orientation* (Field 3), (iv) a *bug report title* (Field 4), and (v) a *brief description of the bug* (Field 5).

The first drop down list (see Figure 2 - Field 6) corresponds to the possible actions a user can perform at a given point in app execution. For example, let's say the reporter selects *click* as the first action in the sequence of steps as shown in Figure 2. The possible actions considered in Fusion are *click(tap), long-click(long-touch), type*, and *swipe*. The *type* action corresponds to a user entering information from the device keyboard. When the reporter selects the *type* option, we also present them with a text box to collect the information she typed in the Android app.

The second dropdown list (see Figure 2 - outlined in blue) corresponds to the component associated with the action in the step. Fusion presents the following information, which can also be seen in Figure 2: (i) *Component Type*: this is the type of component that is being operated upon, (e.g. button, spinner, checkbox), (ii) *Component Text*: the text associated with or located on the component, (iii) *Relative Location*: the relative location of the component on the screen according to the parameters in Figure 3, and (iv) *Component Image*: an in-situ (i.e., embedded in the dropdown list) image of the instance of the component. The relative location is displayed here to make it easier for reporters to reason about the on-screen location, rather than reasoning about pixel values. In the example from Figure 2 above, Fusion will populate the component dropdown list with all of the clickable components in the `main` Activity since this is the first step and the selected action was *click*.

Fusion uses two techniques to handle instances of seemingly identical GUI-components appearing on the same screen.
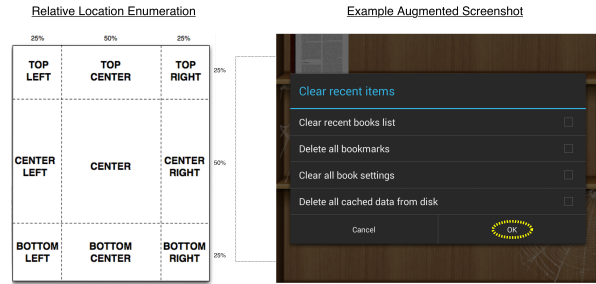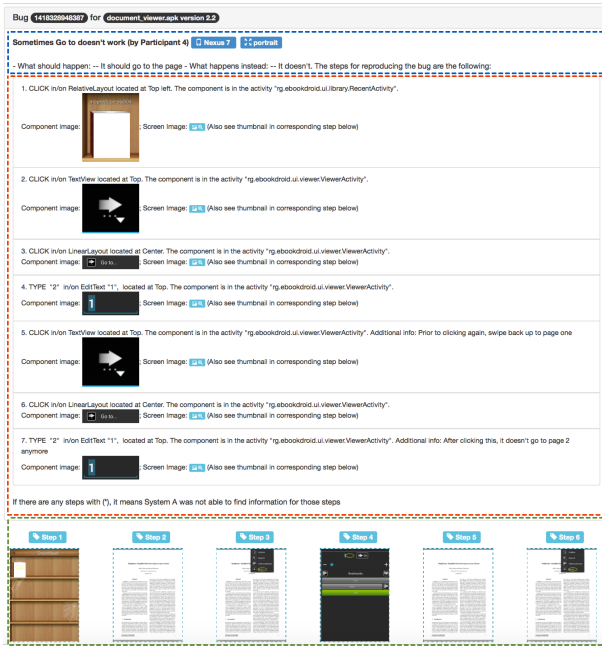
Figure 4: Example FUSION **Bug Report**

to the database for a developer to view later (see Figure 4 for an example report).

The Report presents information to developers in three major sections: First, preliminary information including the report title, device, and short description (shown in Figure 4 in blue). Second, a list of the Steps with the following information regarding each step is displayed (highlighted in blue in Figure 4): (i) The action for each step, (ii) the type of a component, (iii) the relative location of the component, (iv) the `activity` Java class where the component is instantiated in the source code, and (v) the component specific screenshot. Third, a list of full screen-shots corresponding to each step is presented at the bottom of the page so the developer can trace the steps through each application screen (this section is highlighted in green in Figure 4).

3. EVALUATION

To evaluate FUSION (see full details in [23]) we investigated its ease of use, as well as the reproducibility of the FUSION reports compared to reports created using Google Code Issue Tracker (GCIT). First, in a *bug-creation study* we recruited eight students (four undergraduate or *non-experts* and four graduate or *experts*) to construct bug reports using FUSION and GCIT — as a representative of traditional bug tracking systems— for 15 real-world world bugs for 14 open-source apps from F-Droid [5]. We collected survey responses from these participants regarding the ease of use and user preferences of each tool. Next, in a *bug-reproduction study* we evaluated the reproducibility of the FUSION and GCIT reports generated by the first group of participants. These reports (120 for each type) and the original bug reports extracted from the respective app issue trackers were evaluated by a new set of 20 graduate student participants through attempted bug reproduction on physical devices. **The results of this study indicate that developers using** FUSION **reports are able to reproduce more reports compared to traditional bug tracking systems such as the GCIT.**

4. DEMO REMARKS AND FUTURE WORK

In this demo, we presented FUSION, a novel implementation of an enhanced bug reporting system for Android applications. FUSION facilitates reporters crafting detailed reports containing reproduction steps, screenshots, and traceability links to code artifacts, by informing the reporting process with data gleaned from static and dynamic program analyses. As future work, we plan to (i) investigate more sophisticated methods of modeling program behavior, such as using statistical language models, (ii) to improve our DFS engine through supporting more gestures, (iii) and to use FUSION as a tool for reporting feature requests.

5. REFERENCES

[1] Android uiautomator http://developer.android.com/tools/help/uiautomator/index.html.

[2] apktool https://code.google.com/p/android-apktool/.

[3] Bugzilla issue tracker https://bugzilla.mozilla.org.

[4] dex2jar https://code.google.com/p/dex2jar/.

[5] F-droid. https://f-droid.org/.

[6] Github issue tracker https://github.com/features.

[7] Google code issue tracker https://code.google.com/p/support/wiki/IssueTracker.

[8] jd-cmd decompiler https://github.com/kwart/jd-cmd.

[9] Jira bug reporting system https://www.atlassian.com/software/jira.

[10] Mantis bug reporting system https://www.mantisbt.org.

[11] srcml http://www.srcml.org.

[12] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. ASE'12, pages 258–261.

[13] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. pages 298–308, May 2009.

[14] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. OOPSLA '13, pages 641–660. ACM, 2013.

[15] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? SIGSOFT '08/FSE-16, pages 308–318.

[16] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. MSR'08.

[17] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful... really? pages 337–345, Sept 2008.

[18] S. Davies and M. Roper. What's in a bug report? ESEM '14, pages 26:1–26:10.

[19] M. Erfani Joorabchi, M. Mirzaaghaei, and A. Mesbah. Works for me! characterizing non-reproducible bug reports. MSR 2014, pages 62–71.

[20] Ericsson. Ericsson mobility report. http://www.ericsson.com/res/docs/2014/ericsson-mobility-report-november-2014.pdf.

[21] 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)*, to appear, 2015.

[22] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. ESEC/FSE'13, pages 224–234.

[23] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk. Auto-completing bug reports for android applications. ESEC/FSE'15, pages 673–686. ACM.

[24] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002.