MZoltar: Automatic Debugging of Android Applications

Pedro Machado, José Campos, and Rui Abreu
Department of Informatics Engineering
Faculty of Engineering, University of Porto
Porto, Portugal
{machado.pedro, jose.carlos.campos}@fe.up.pt, rui@computer.org

ABSTRACT

Automated diagnosis of errors and/or failures detected during software testing can greatly improve the efficiency of the debugging process, and thus help to make applications more reliable. In this paper, we propose an approach, dubbed MZoltar, offering dynamic analysis (namely, spectrum-based fault localization) of mobile apps that produces a diagnostic report to help identifying potential defects quickly. The approach also offers a graphical representation of the diagnostic report, making it easier to understand. Our experimental results show that the approach requires low runtime overhead (5.75% on average), while the tester needs to inspect 5 components (statements in this paper) on average to find the fault.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Experimentation

Keywords

Mobile software, fault detection, automated debugging.

1. INTRODUCTION

Software reliability can generally be improved through extensive testing and debugging, however this often conflicts with market conditions. Often, software cannot be tested exhaustively, and of the bugs that are found, only those with the highest impact on the user-perceived reliability can be solved before the release. In this typical scenario, testing reveals more bugs than can be solved, and debugging is a bottleneck for improving reliability. Automated debugging techniques can help reducing this bottleneck [2, 15, 21, 25, 26, 29, 30].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DeMobile '13, August 19, 2013, Saint Petersburg, Russia
Copyright 13 ACM 978-1-4503-2312-3/13/08 ...$15.00.

Despite recent advances in fault localization techniques, the development of applications for mobile devices - such as Android apps - still pose interesting challenges [10].

- The large amount of available devices and their large range of specifications (e.g., cpu speed, screen resolution), make it difficult to check the consistency and ensure portability between different devices and platforms;
- Testing apps for each target platform requires the development of several versions. Furthermore, the available testing frameworks have serious limitations for testing mobile specific features;
- Developers claim that better analysis tools and techniques to help debugging apps are seriously needed. In fact, according to the World Quality Report [9], $^2/_3$ of surveyed developers mentioned that they do not have the proper tools to test and debug mobile apps, despite the available tools such as provided by Android Software Development Kit (SDK) and Android Development Tools (ADT) plugin.

Locating a fault is an important step in actually fixing it. Spectrum-based fault localization (SFL) is a technique, which is amongst the best performing techniques, that helps identifying the root cause of observed failures, relying on program execution data and test pass/fail information. GZoltar[1] [8], which focuses on Java programs, and Tarantula [19], which focuses on C programs are examples of tools offering the SFL technique. Since it is lightweight, SFL has been successfully applied in the context of embedded software [31]. However, despite these tools and the increasingly active research in the area of fault localization, to the best of our knowledge, not much has been reported in the area of mobile software. We argue that SFL has the potential to be applicable to perform fault localization in mobile apps.

Furthermore, following the same strategy as GZoltar, our approach provides a visual representation of the diagnostic report to aid the developers in the process of locating the defects in the code. These visualizations are a *translation* of the report into an intuitive representation that can ease and speed up the fault localization process [11, 20, 24].

Our empirical study, using 4 open-source Android applications with single and multiple injected faults, supports the argument that spectrum-based fault localization is well suited for mobile apps. The underlying infrastructure to

[1] GZoltar homepage `http://gzoltar.com`, 2013.

collect the required information entails low runtime overhead (5.75% on average, with standard deviation σ=2.49), while the tester needs to inspect 5 components on average to find the fault.

The main contributions of this paper are:

- We discuss the challenges faced by developers when doing fault localization, highlighting the real-world relevance of the problem;
- We propose a fully automated approach for localizing defects in Android applications. Our approach is based on a well-known spectrum-based fault localization technique and produces a visual report to aid in locating the defects;
- We provide a toolset, MZoltar, embedded into the Eclipse Integrated Development Environment (IDE) providing the proposed fault localization technique;
- We carried out an empirical study to demonstrate the efficiency of MZoltar.

2. CONCEPTS AND DEFINITIONS

In this section, concepts and definitions relevant to this paper are introduced. Throughout this paper, the following terminology is used [6]: a *failure* is an event that occurs when the delivered service deviates from correct service, an *error* is a system state that may cause a failure, while *fault* (defect/bug) is the cause of an error in the *system.* In this paper, this terminology is applied to software programs, where faults are bugs in the program code. Failures and errors are symptoms caused by faults in the program. The purpose of fault localization is to pinpoint the root cause of observed symptoms.

A software program Π (a mobile app in the context of this paper) is formed by a sequence M of one or more statements. A test suite $T = \{t_1, \ldots, t_N\}$ is a collection of test cases that are intended to test whether the program follows the specified set of requirements. The cardinality of T is the number of test cases in the set $|T| = N$. Finally, a test case t is a (i, o) tuple, where i is a collection of input settings or variables for determining whether a software system works as expected or not, and o is the expected output. If $\Pi(i) = o$ the test case passes, otherwise fails.

2.1 Android

Android is an open source Linux-based operating system targeted for mobile or embedded devices. Typically, Android applications are developed in Java. However, native-code languages such as C and C++ may also be used. Note that Android reuses the Java language syntax and semantics, but it does not provide the full class libraries and APIs bundled with Java SE. Java development is mainly supported by a comprehensive set of development tools Android SDK, while native code is supported by the Android Native Development Kit (NDK). Android applications have a specific lifecycle that differs from the Java regular applications' lifecycle. Instead of a main function, Android applications' main components are: *Activities*; *Services*; *Content providers*; *Broadcast receivers.* Furthermore, Android applications are not only comprised of source files, but also include resource files (mainly related to the specification of layouts and translations) and a manifest file (as mentioned before, responsible for providing the necessary information about the application to the Android system).

2.2 Fault Localization

Spectrum-based fault localization (SFL) [2,3] is a statistics-based lightweight fault localization technique and it is considered to be amongst the most effective ones [2], [21], [29]. This technique uses a dynamic analysis approach, as it relies on program execution information (*program spectrum*) from previous runs (passed and failed) to correlate the software components with the observed failures and determine the suspiciousness of each component being faulty. Passed runs are program executions that completed correctly, while failed runs are executions in which an error was detected. A *Program spectrum* is a collection of data that indicates which components of the software were hit during a run [3].

The input of the SFL is constituted by the hit spectra and an error-vector [3]. The hit spectra of N runs constitutes a binary $N \times M$ matrix A. M represents the components of the program. The error-vector, e, is a N-length vector and each position represents the outcome of a run (passed or failed).

After obtaining these two inputs, the resemblance between the error-vector and each column of the hit spectra matrix is calculated by means of a *similarity coefficient.* Although there are many similarity coefficients, the Ochiai coefficient (see Equation 1), also used in the molecular biology domain, is considered one of the best performing similarity coefficients for SFL [3]

$$s_O(j) = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{01}(j)) \cdot (n_{11}(j) + n_{10}(j))}} \quad (1)$$

where $n_{pq}(j)$ is the number of runs in which a component (j) was hit $(p = 1)$ or not hit $(p = 0)$ during an execution, and where that execution failed $(q = 1)$ or was successful $(q = 0)$. $n_{pq}(j)$ is formally defined as

$$n_{pq}(j) = |\{i \mid a_{ij} = p \wedge e_i = q\}| \quad (2)$$

3. MOTIVATIONAL EXAMPLE

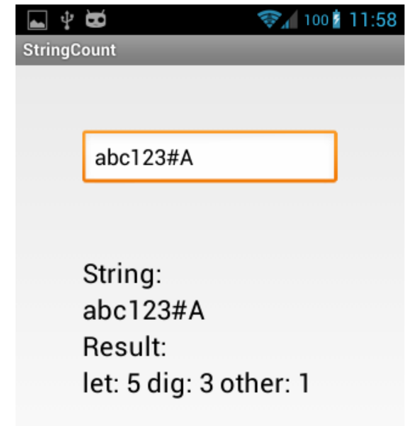
To illustrate the problem addressed in this paper, consider the simple Android application in Figure 1a (based on the example used in [12]). To improve the legibility, the coverage matrix and the error detection vector were transposed.

This running example uses a function *count()* that receives a string as an argument and prints the number of times each type of char (letter, number or other) occurs in that string. A bug has been injected in line 5 (Figure 1a), where the *let* counter should be incremented by just one when the string includes a capital letter, but instead it is being incremented by two. The figure also shows the code coverage information of the 8 executed tests. For each row, a ● appears in the columns that correspond to the tests where that line was touched. The error vector, e is presented at the bottom of the table, showing the passed/failed information of the executed tests. Resorting to this information, the Ochiai coefficient, s_O, is used to calculate the suspiciousness of a given line containing a fault. In this case, SFL has successfully performed the fault localization as the ranking created encourages the developer to inspect the faulty line first.

In Figure 1b the graphical user interface of the application implemented for the Android operating system is presented.

Subject: CharCount	T										
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	s_O
`class CharCount {...`											
`static void count(String s) {`											
1: `int let, dig, other;`	●	●	●	●	●	●	●	●	●	●	0.63
2: `for(int i = 0; i < s.length(); i++) {`	●	●	●	●	●	●	●	●	●	●	0.63
3: `char c = s.charAt(i);`	●	●	●	●	●	●	●		●	●	0.67
4: `if ('A'<=c && 'Z'>=c)`	●	●	●	●	●	●	●		●	●	0.67
5: `let += 2; /* FAULT */`	●		●	●			●				1.00
6: `else if ('a'<=c && 'z'>=c)`	●	●	●	●	●	●	●		●	●	0.67
7: `let += 1;`					●	●	●		●	●	0.22
8: `else if ('0'<=c && '9'>=c)`	●	●	●	●	●	●			●	●	0.53
9: `dig += 1;`	●	●	●	●		●			●	●	0.57
10: `else if (isprint(c))`					●	●					0.00
11: `other += 1; }`					●	●					0.00
12: `System.out.println(let + " " + dig + " " + other); }`	●	●	●	●	●	●	●	●	●	●	0.63
`...}`											
Test case outcome (pass=✓, fail=✗)	✗	✓	✗	✗	✓	✓	✗	✓	✓	✓	

(a) Example of SFL technique with Ochiai coefficient (adapted from [12]).



(b) Android example application with bug in result.

Figure 1: Example.

The application receives a string through a text box and shows the result of the counting algorithm. Also, the result is affected by the bug injected in the previous example. There are 4 letters in the string, but the *letter* counter indicates the value 5, as the capital letter 'A' increments 2 units in the *letter* counter while it should increment just one. The application was also tested using the Android testing framework [2] to assess its functioning.

To retrieve the program spectra that contains the execution information, there is the need to instrument the application. However, as Android devices often possess limited computational resources, the execution times are greatly sensible to instrumentation overhead. Moreover, the embedded nature of Android devices hinders the retrieval of runtime information, used as input to SFL.

Thus, the application of the SFL technique to Android devices presents the following challenge: Given it is a resource-constrained environment, is SFL, and collecting the coverage information, well suited to perform automatic fault localization in mobile apps software?

4. MZOLTAR

There are a few toolsets offering spectrum-based fault localization, such as GZoltar[3] [8], Tarantula [19], or EzUnit [7]. GZoltar and Tarantula show the diagnostic reports visually in an attempt to facilitate the quest for the defects. EzUnit provides a textual ranking of the lines that are most likely to be faulty and assigns a background color to each line matching its failure probability.

In this section, we detail the novel aspects of MZoltar, which make it suitable to the mobile apps testing and debugging phase. Since MZoltar is based on GZoltar, they both provide the same set of visualizations. Figure 2, as an example, shows one of the three GZoltar's visualizations[4], the Sunburst visualization. Sunburst shows the information as an hierarchical structure, taking advantage of the fact software is inherently hierarchical, in particular the Java-based object-oriented software used in the development of

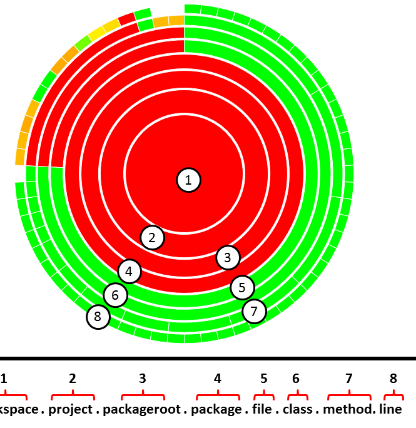


Figure 2: Sunburst visualization (and how to interpret it).

Android applications. Both in GZoltar and MZoltar, the visualizations also allow users to navigate through the code and interact with the source code, trying to ease the task of finding bugs.

4.1 Workflow

As mentioned in Section 3, Android devices present a series of challenges in what it concerns the application of SFL. To surpass those challenges, MZoltar relies on the **`Android testing framework`** and **`JaCoCo`**[5] to run the applications' tests and acquire coverage information. As MZoltar is a plugin for the Eclipse IDE, it also relies on the abstractions provided by the ADT plugin to perform some of the tasks. MZoltar's flow is:

1. Instrument bytecode;
2. Generate application's `apk` and flush into the device;
3. Run tests;
4. Collect code coverage;
5. Run diagnostic report.

[2] Android Testing Fundamentals `http://developer.android.com/tools/testing/testing_android.html`, 2013.
[3] GZoltar homepage `http://gzoltar.com`, 2013.
[4] Other visualization can be seen at `http://www.gzoltar.com`

[5] JaCoCo homepage `http://www.eclemma.org/jacoco/`, 2013.

Table 1: Experimental Subjects.

Subject	Version	Lines of Code (LOC)	Test Cases	Test Cases LOC	Coverage	Resources LOC
CharCount	1.0	148	10	133	92.2%	115
ConnectBot	1.7.1	32911	14	484	0.7%	7673
Google Authenticator	2.21	3659	170	2825	76.6%	5275
StarDroid	1.6.5	13783	187	3029	29.7%	2694

To instrument the application, phase (1), there were two options available: instrument the Dalvik Bytecode directly in the `apk` file or instrument the original Java Bytecode generated by Eclipse (afterwards compiled into the Dalvik Bytecode when the `apk` is built). As Dalvik Virtual Machine (VM) is register based and the available frameworks (such as ASMDex[6]) do not provide a way to automatically reallocate the registers after instrumenting the code, we opted for the Java Bytecode instrumentation. In particular, we used a recently developed feature of `JaCoCo`, the Offline instrumentation[7]. Android test framework has an EMMA[8] code coverage analysis feature that can be replaced by `JaCoCo` offline instrumentation to retrieve the code coverage information of a test execution.

After the Bytecode was instrumented, ADT API was used to build the `apk` file, phase (2), as well as to run the tests in phase (3). In this last step the code coverage information is generated in the device. In phase (4), the IDevice interface provided by ADT is used to pull the coverage data file from the device to be further processed, offline, by MZoltar. Using the `JaCoCo` API, the coverage file is mapped into the input expected by SFL and the diagnostic report is computed (phase 5).

4.2 Eclipse integration

MZoltar is offered as a plugin for the well-known Eclipse IDE. Features such as the visualizations, code navigation, and the editor markers are reused from GZoltar. The existence of an official ADT plugin to Eclipse was taken into account, as it aided in the implementation of the Android related features (see Subsection 4.1). There follows a description of MZoltar's main features.

Visualizations, such as Sunburst (see Figure 2), provide useful information to the user, as they translate the diagnostic reports into an intuitive graphical representation. The similarity coefficient value of each component is used to create a color gradient, that goes from red, for the components that are more likely to be faulty, and ends with green to the less likely ones. The structured visual representation also makes it easier to understand the program structure, hence reducing the effort of locating a given component in the code. The available visualizations and their features are thoroughly described in [13].

Besides the intuitive representation, there are also some interactions that ease the debugging process. To get to a component's location in the code, the user only needs to click on that component in the visualization. Then the editor opens the file and highlights the previously clicked component. It is also possible to change the visualization, by performing a root change or zooming, so the user can focus on a desired set of components.

In MZoltar the embedded nature of Android devices implies a different way of running the tests, collecting coverage information and processing it, therefore, ADT was crucial in its implementation and is the main difference between MZoltar and GZoltar. ADT is an Eclipse IDE plugin that provides the possibility of building Android apps with Eclipse. Some of its features, such as pulling files from the device, building `apk` files, managing apps on the device or running tests, were used in the implementation of MZoltar.

Before being able to execute the diagnostic algorithm, the user has to (i) select the project that is going to be tested, (ii) select the Test Runner to use, and (iii) select whether or not the application should be uninstalled from the device after it is tested. Furthermore, we decided to use the ADT device chooser dialog to make the experience of using MZoltar as similar as possible with the usual experience of developing an Android application with the Eclipse IDE.

5. EVALUATION

In this section, we describe the empirical evaluation we carried out to assess MZoltar's performance, in particular to verify its applicability to the context of mobile apps. We start by describing the experimental setup, followed by a discussion on the observed results. The empirical evaluation aims at answering the following research questions:

RQ1 Is the MZoltar's instrumentation overhead negligible?

RQ2 Does MZoltar yield accurate diagnostic reports under Android device's constrained environment?

5.1 Experimental Setup

Four mobile apps, of different sizes and complexities, were considered to empirically evaluate MZoltar. Table 1 presents further information about the subjects. CharCount is the subject used as a motivational example in Section 3. ConnectBot[9] is an Android Secure Shell (SSH) client. Google Authenticator[10] is a two-step authentication application. StarDroid[11] is sky map open source project. To foster reproducibility and comparability, we report the version number of the subjects used in the evaluation. Lines of Code (LOC) count information was obtained using `Code Analyzer`[12]. Code coverage information was obtained using

[6]ASMDex homepage `http://asm.ow2.org/asmdex-index.html`, 2013.
[7]JaCoCo Offline instrumentation homepage `http://www.eclemma.org/jacoco/trunk/doc/offline.html`, 2013.
[8]EMMA homepage `http://emma.sourceforge.net/`, 2013.
[9]ConnectBot homepage `http://code.google.com/p/connectbot`, 2013.
[10]Google Authenticator homepage `http://code.google.com/p/google-authenticator`, 2013.
[11]StarDroid homepage `http://code.google.com/p/stardroid`, 2013.
[12]Code Analyzer homepage `http://www.codeanalyzer.teel.ws`, 2013.

JaCoCo[13], enabling the coverage flag of the Android tests framework. Then, we used the EclEmma[14] Eclipse plugin to analyse the generated coverage files.

As the subjects are bug-free (with regard to the tests suite), eight common mistakes [14] were injected in each subject. To facilitate the activation of the faults, thus automating the testing process, we built a fault injection framework[15]. This framework allows to enable/disable the faults automatically and use a custom InstrumentationTestRunner[16], named MZoltarTestRunner, that parses an argument (injectedFaults) which indicates which faults should be active per run. Then, each application was executed 30 times for each of the following scenarios $\{(f, g) \mid f \in \{1, 2, 3, 5\} \wedge g \in \{1, 2, 3, 5, 10\}\}$, where f is the number of injected faults and g is the number of tests considered in a transaction [17].

These scenarios make it possible to assess the performance of MZoltar in different situations, being also important to evaluate the tradeoff *effectiveness* vs *time*. Running each test separately may entail a considerable time overhead (since, per test execution, Android terminates and starts a new VM). Our goal is to evaluate the consequences of executing several tests simultaneously, assessing the potential time reduction vs. the potential information and effectiveness losses.

The experiments target device was an emulator running Android 2.2 (API Level 8) with a 4" (480x800 hdpi) screen, an ARM processor, 343MB of RAM and 32MB of VM Heap. The emulator was used on a 3.16GHz Intel® Core™ 2 Duo PC with 2GB of RAM, running Debian 7.0 (wheezy).

5.2 Evaluation Metric

To measure the success of a diagnosis technique we use the diagnostic quality C_d, which estimates the number of components the tester needs to inspect to find the fault [27]. Note that C_d cannot be computed prior to computing the ranking: one does not know the actual position of true-fault candidates in the ranking beforehand. Because multiple explanations can be assigned with the same similarity value, the value of C_d for the real fault d_* is the average of the ranks that have the same similarity value:

$$\theta = |\{j|s_O(m) > s_O(d_*)\}|, \ 1 \leq j \leq M$$

$$\phi = |\{j|s_O(m) \geq s_O(d_*)\}|, \ 1 \leq j \leq M \qquad (3)$$

$$C_d = \tfrac{\theta \ + \ \phi \ - \ 1}{2}$$

In the multiple fault cases we use the one-at-a-time mode, discussed in [27]: one fault is identified and fixed, and then the fault localization process is repeated (including a re-run of the test suite and a re-computation of the input for the fault localization technique). We report C_d for the first fault found, as one can estimate the impact of reducing the number of faults on C_d in the experiments of lower number of injected faults.

[13]JaCoCo homepage http://www.eclemma.org/jacoco, 2013.

[14]EclEmma homepage http://www.eclemma.org/, 2013.

[15]Again, to foster reproducibility and comparability, the injection framework can be obtained at http://www.gzoltar.com/mzoltar/demobile13

[16]Android Instrumentation Test Runner homepage http://developer.android.com/reference/android/test/InstrumentationTestRunner.html, 2013.

[17]We considered a transaction, i.e., a row in the matrix, the set of test cases clustered in each virtual machine execution. Will be explained in more detailed in Subsection 5.3

Table 2: Execution times.

Subject	Original	Instrumented	Overhead
CharCount	1.82s	1.86s	2%
ConnectBot	1.25s	1.35s	8%
Google Authenticator	80.49s	87.26s	8%
StarDroid	14.70s	15.46s	5%

We further use a metric $\bar{\rho}$, the *density of the coverage matrix* [12], that has been used in the past to build confidence on the C_d obtained, and understand whether one can still improve the diagnostic report by adding more tests. The density of a coverage matrix is the average percentage of components covered by test cases. It is defined as follows

$$\bar{\rho} = \frac{\sum_{i=1}^{N} \sum_{j=1}^{M} a_{ij}}{N \ \cdot \ M}$$

where N and M denote the number of test cases and the number of components, respectively. a_{ij} represents the coverage of the component m when the test t_i is executed. Values of $\bar{\rho}$ close to 0 means that test suite touch a small parts of the program, whereas values close to 1 means that test suite tend to cover most components of the program. In [12] it has been shown that $\bar{\rho} = 0.5$ is the best for fault localization, provided that there is a *diversity* in the test cases of the suite.

5.3 Experimental Results

RQ1: Is the MZoltar's instrumentation overhead negligible?

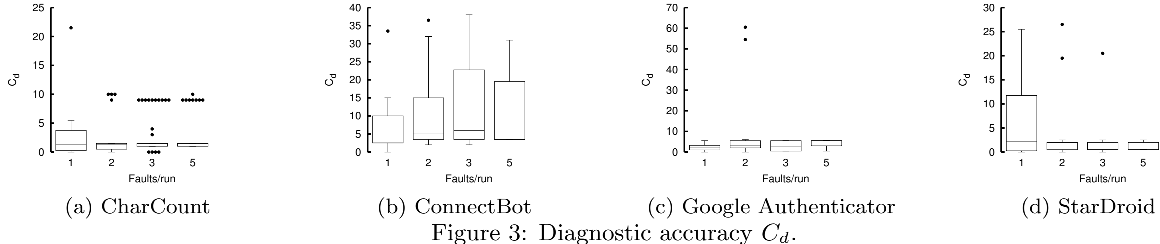
Table 2 shows the execution times for the test subjects. The average execution times of the mobile apps' instrumented versions are, as expected, slightly higher then the original versions. The collected results show that the used instrumentation entails an average time overhead of 5.75% (with standard deviation σ=2.49).

We cannot claim that we have not altered the timing behaviour of the system, but the applications, although slightly slower, are still functioning properly. Therefore, we conclude that the instrumentation overhead is not prohibitive.

RQ2: Does MZoltar yield accurate diagnostic reports under Android device's constrained environment?

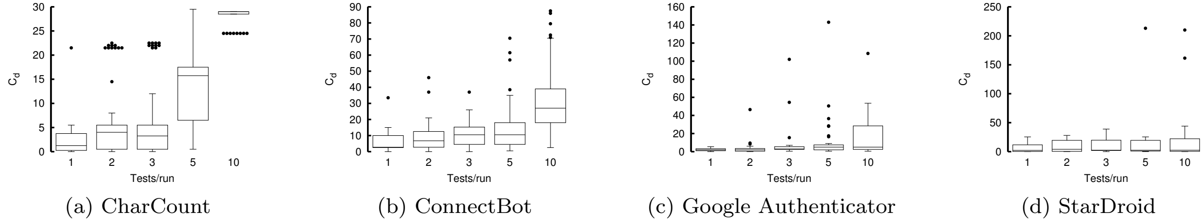
Figure 3 plots the diagnostic accuracy C_d for the following number of injected faults: 1, 2, 3, and 5. The injected faults are 8 faults that are considered to be common [14]. For the single fault scenario, the reported results are on average for the 8 faults, whereas for the multiple fault scenarios we have randomly repeated the experiments 30 times (randomly injecting the faults).

As mentioned before, SFL takes as input the coverage of a transaction. In Android a transaction is composed by all the tests that are executed when the VM is initiated until its tear down. The reason for this notion of transaction (and not one per test case) is that to obtain the individual code coverage information for each test, only one test can be



(a) CharCount (b) ConnectBot (c) Google Authenticator (d) StarDroid

Figure 3: Diagnostic accuracy C_d.



(a) CharCount (b) ConnectBot (c) Google Authenticator (d) StarDroid

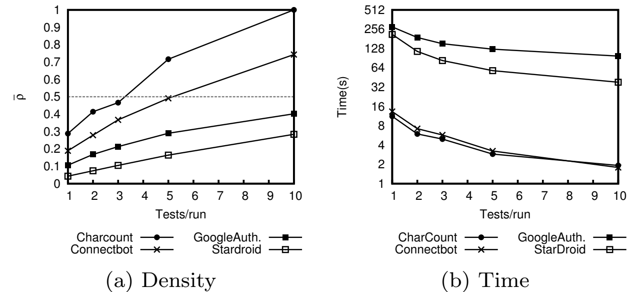
Figure 4: Impact of Grouping Test Cases on C_d.

executed per run (technological limitation). Hence, creating a new VM per test case may impose a considerably high overhead, as a delay is observed between each test execution since the system reboots the VM where the tests are run. This delay was measured to be approximately one second per transaction.

To address this potential bottleneck, we considered to execute multiple test cases per virtual machine. On the one hand, there is a potential reduction in the runtime overhead, but, on the other hand, the noise implied by considered multiple tests as one execution may entail information loss. Consequently, worsening the diagnostic quality C_d. Grouping test cases, i.e., increasing the number of test cases per transaction, decreases the number rows in spectra matrix, N, increasing its density, which is an explanation for the degradation of the diagnostic quality. Figures 5a and 4 plots the impact of grouping test cases in ρ and C_d. For $\rho \geq 0.5$, there is a significant worsening of the diagnostic quality (cf. [12]).

Figure 4 plots C_d when grouping, randomly, several tests per execution (namely, 1,2,3,5, and 10), and the execution overhead is plotted in Figure 5b. For CharCount and ConnectBot, the C_d increases with the number of tests executed per transaction, while for GoogleAuth and StarDroid C_d remains practically constant. These differences are explained by the number of tests each application provide. This way, the percentage reduction of the number of lines (caused by clustering of several tests in each run) of the spectra matrix is higher in the subjects with less tests implemented, thus worsening C_d. Regarding the execution overhead, an exponential reduction was observed with the increase of the number of tests. This overhead reduction is explained by the fact that there is no need to restart the VM.

As an example, when executing 10 test cases per transaction, we observed that grouping test cases reduced the execution overhead in 79% on average (σ=8.36), at the cost of a loss in the diagnostic quality of 74% (σ=12.14). This is mainly due to the density growth that comes along with the increase of the number of tests in a group.



(a) Density (b) Time

Figure 5: Outcomes when grouping multiple tests per transaction.

6. RELATED WORK

ADT and Android SDK provide (mostly manual) debugging capabilities[18] specific to Android, the target operating system of this paper. Dalvik Debug Monitor Server (DDMS) provides port-forwarding services, screen capture on the device, thread and heap information on the device, logcat, process, and radio state information, incoming call and SMS spoofing, location data spoofing, among other features. Plus, the Dalvik VM supports the Java Debug Wire Protocol (JDWP) protocol which enables the user to use any debugger that supports this protocol to debug the apps running on the device. There are also some other tools like the Hierarchy Viewer, used to optimize the application's user interface, the Traceview which is a graphical viewer used to log tracing information about the running application and the Dev Tools application that can be installed on Android devices and provide a way to enable debugging targeted feature in the device.

Proposed recently, GROPG [22] is an on-phone debugger that enables the debugging of the application in real time on top of the running application itself. It provides traditional

[18] Android debugging homepage `http://developer.android.com/tools/debugging/index.html`, 2013.

debugging actions like breakpoints, step into, step over, step out, in-scope variable analysis and thread analysis.

Automatic fault localization techniques exist, but have mainly been investigated in the context of general purpose programs. Amongst the best fault localization techniques is SFL [2, 3, 21, 29], which is the underlying technique of the MZoltar toolset. Although many techniques have been investigated, not many off-the-shelf tools exist offering them.

GZoltar [8] is a graphical debugging tool for the Eclipse IDE, on which MZoltar was based. Both toolsets are very similar, only differing in the support for Android devices. Tarantula [19] is a standalone graphical debugging tool. It presents a visualization that overviews all the source code, representing each line with a color that indicates its failure probability. Tarantula only targets C projects. Vida [15] is an Eclipse plug-in based on Tarantula that suggests places where the developer should place breakpoints to analyse the system. EzUnit4 [7] is another Eclipse plugin with graphical debugging purposes. Like MZoltar it is based on JUnit tests and on the use of statistical analysis to calculate the failure probability. Each line has a background color matching its failure probability (from red for high probability, to green for low probability).

Attempts to automate the process of testing and debugging mobile apps include the following. Bo Jiang *et al.* described a statistical fault localization technique for mobile embedded systems [18], where not only the code is targeted, but also suspicious context providers. Incorporating a fault localization logic into the app makes it able to choose the most reliable context provider, when a crash occurs. The new context provider is chosen from a list of the same group of providers, ordered based on their suspiciousness score.

A system to automatically and systematically generate input events to exercise smartphone apps and its underlying algorithm, based on a concolic testing approach, is described in [5]. Moreover, GUI testing in mobile devices is an active research subject [4, 16, 17, 28].

Pascual *et al.* used a generic algorithm to automatically generate optimal application configurations, based on feature model, at runtime [23]. This optimizes the configuration of the system at runtime according to the available resources. The approach does not entail excessive overhead, and helps the app coping with the resource constrained environment and optimizing its performance.

Embedded systems, category in which we can fit mobile devices, were already targeted to measure SFL's performance in such resource-constrained environments [31]. This study has confirmed that fault diagnosis through analysis of program spectra performs well under harsh conditions and opened corridors to new applications, such as run-time recovery.

Despite the myriad of techniques and approaches, there are still shortcomings when applying these techniques in the context of mobile, resource-constrained apps. Available automated fault localization toolsets do not offer easy integration into the mobile apps world. As a consequence, manual approaches are still prevalent in the mobile apps debugging and testing phases, and the debugging tools available for mobile apps only offer manual debugging features [10]. Hence, MZoltar addresses that issue by providing an automated fault localization approach, offering the SFL dynamic analysis.

7. CONCLUSIONS AND FUTURE WORK

We propose MZoltar, an approach to aid in localizing defects in Android-based mobile apps relying on the Spectrum-based fault localization (SFL) technique. Moreover, the diagnostic report is shown to the user using a graphical visualization, which makes it easier to understand the report.

In an empirical study using 3 real, open-source Android applications and an example application (with single and multiple injected faults), confirms that spectrum-based fault localization is well suited for the mobile apps development context. The infrastructure to collect the information needed is lightweight (overhead of $5.75\% \pm 2.49\%$ on average), while the diagnostic accuracy is similar to the one observed on general-purpose applications [1].

Future work includes the following. First, this preliminary work does not take a very important feature into account, responsible for many run-time failures: the manifest file. As the manifest file is not *executed*, the dynamic analysis proposed in this paper is not able to include it in the reasoning process. Consequently, we plan to integrate the dynamic fault localization analysis with static analysis using Lint. Lint is a toolset integrated within the Android SDK to statically analyze potential problems in the application. This integration will have an impact in the visualizations themselves. Second, to be able to ascertain the usefulness of our approach, we plan to carry out a user study. Third, we intend to provide MZoltar in the recently announced Android Studio IDE. Finally, we plan to port MZoltar to other mobile technologies, notably to iOS and Windows Phone.

8. ACKNOWLEDGMENTS

This work is partially funded by the ERDF through the Programme COMPETE, the Portuguese Government through FCT - Foundation for Science and Technology, project reference FCOMP-01-0124-FEDER-020484.

9. REFERENCES

[1] R. Abreu. *Spectrum-based Fault Localization in Embedded Software*. PhD thesis, Delft University of Technology, November 2009.

[2] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792, Nov. 2009.

[3] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the Accuracy of Spectrum-based Fault Localization. In *Proc. of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, TAICPART-MUTATION '07, pages 89–98, 2007.

[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *Proc. of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE '12, pages 258–261, 2012.

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

[6] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, Jan. 2004.

[7] P. Bouillon, J. Krinke, N. Meyer, and F. Steimann. EZUNIT: a framework for associating failed unit tests with potential programming errors. In *Proc. of the 8th international conference on Agile processes in software engineering and extreme programming*, XP'07, pages 101–104, 2007.

[8] J. Campos, A. Riboira, A. Perez, and R. Abreu. GZoltar: An Eclipse Plug-in for Testing and Debugging. In *Proc. of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 378–381, 2012.

[9] O. Capgemini: Consulting, Technology, Sogeti, and H.-P. (HP). World Quality Report 2012. Technical report, Capgemini, Sept. 2012.

[10] M. Erfani, A. Mesbah, and P. Kruchten. Real Challenges in Mobile App Development. In *Proc. of the ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '13.

[11] S. D. Fleming, C. Scaffidi, D. Piorkowski, M. M. Burnett, R. K. E. Bellamy, J. Lawrance, and I. Kwan. An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks. *ACM Trans. on Software Engineering and Methodology*, 22(2):14, 2013.

[12] A. Gonzalez-Sanchez, E. Piel, H.-G. Gross, and A. J. C. van Gemund. Prioritizing Tests for Software Fault Localization. In *Proc. of the 2010 10th International Conference on Quality Software*, QSIC '10, pages 42–51, 2010.

[13] C. Gouveia, J. Campos, and R. Abreu. Using HTML5 Visualizations in Software Fault Localization. In *Proceedings of the 1st IEEE Working Conference on Software Visualization*, VISSOFT '13, Washington, DC, USA, 2013. IEEE Computer Society.

[14] M. Hamill and K. Goseva-Popstojanova. Common trends in software fault and failure data. *Software Engineering, IEEE Trans. on*, 35(4):484–496, 2009.

[15] D. Hao, L. Zhang, L. Zhang, J. Sun, and H. Mei. VIDA: Visual interactive debugging. In *Proc. of the 31st International Conference on Software Engineering*, ICSE '09, pages 583–586, 2009.

[16] C. Hu and I. Neamtiu. Automating GUI testing for Android applications. In *Proc. of the 6th International Workshop on Automation of Software Test*, AST '11, pages 77–83, 2011.

[17] A. Jaaskelainen, M. Katara, A. Kervinen, M. Maunumaa, T. Paakkonen, T. Takala, and H. Virtanen. Automatic GUI test generation for smartphone applications - an evaluation. In *Proc. of the 31st International Conference on Software Engineering*, ICSE '09, pages 112–122, 2009.

[18] B. Jiang, X. Long, X. Gao, Z. Liu, and W. Chan. FLOMA: Statistical fault localization for mobile embedded system. In *Advanced Computer Control (ICACC), 2011 3rd International Conference on*, pages 396–400, 2011.

[19] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proc. of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 273–282, 2005.

[20] J. Lawrance, C. Bogart, M. M. Burnett, R. K. E. Bellamy, K. Rector, and S. D. Fleming. How Programmers Debug, Revisited: An Information Foraging Theory Perspective. *IEEE Trans. on Software Engineering*, 39(2):197–215, 2013.

[21] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical Debugging: A Hypothesis Testing-Based Approach. *IEEE Trans. Softw. Eng.*, 32(10):831–848, Oct. 2006.

[22] T. A. Nguyen, C. Csallner, and N. Tillmann. GROPG: A graphical on-phone debugger. In *Proc. 35th ACM/IEEE International Conference on Software Engineering (ICSE), New Ideas and Emerging Results (NIER) track*, May 2013.

[23] G. G. Pascual, M. Pinto, and L. Fuentes. Run-time adaptation of mobile applications using genetic algorithms. In *Proc. of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '13, pages 73–82.

[24] P. Pirolli. *Information Foraging Theory: Adaptive Interaction with Information.* Human Technology Interaction Series. 1 edition, 2007.

[25] G. Shu, B. Sun, A. Podgurski, and F. Cao. Mfl: Method-level fault localization with causal inference. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST).*

[26] F. Steimann and M. Frenkel. Improving coverage-based localization of multiple faults using algorithms from integer linear programming. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 121–130.

[27] F. Steimann, M. Frenkel, and R. Abreu. Threats to the Validity and Value of Empirical Assessments of the Accuracy of Coverage-Based Fault Locators. In *Procedings of the International Symposium in Software Testing and Analysis*, ISSTA 2013, 2013.

[28] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based gui testing of an android application. In *Proc. of the IEEE Fourth International Conference on Software Testing, Verification and Validation*, ICST '11, pages 377–386, 2011.

[29] E. Wong, T. Wei, Y. Qi, and L. Zhao. A Crosstab-based Statistical Method for Effective Fault Localization. In R. Hierons and A. Mathur, editors, *Proc. of the 1st International Conference on Software Testing, Verification, and Validation (ICST'08)*, pages 42–51, Lillehammer, Norway, 2008.

[30] W. Wong, V. Debroy, Y. Li, and R. Gao. Software fault localization using DStar (D*). In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, pages 21–30, 2012.

[31] P. Zoeteweij, R. Abreu, R. Golsteijn, and A. J. C. van Gemund. Diagnosis of Embedded Software Using Program Spectra. In *Proc. of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, ECBS '07, pages 213–220, 2007.