Systematic Execution of Android Test Suites in Adverse Conditions

Christoffer Quist Adamsen
Aarhus University, Denmark
quist@cs.au.dk

Gianluca Mezzetti
Aarhus University, Denmark
mezzetti@cs.au.dk

Anders Møller
Aarhus University, Denmark
amoeller@cs.au.dk

ABSTRACT

Event-driven applications, such as, mobile apps, are difficult to test thoroughly. The application programmers often put significant effort into writing end-to-end test suites. Even though such tests often have high coverage of the source code, we find that they often focus on the expected behavior, not on occurrences of unusual events. On the other hand, automated testing tools may be capable of exploring the state space more systematically, but this is mostly without knowledge of the intended behavior of the individual applications. As a consequence, many programming errors remain unnoticed until they are encountered by the users.

We propose a new methodology for testing by leveraging existing test suites such that each test case is systematically exposed to adverse conditions where certain unexpected events may interfere with the execution. In this way, we explore the interesting execution paths and take advantage of the assertions in the manually written test suite, while ensuring that the injected events do not affect the expected outcome. The main challenge that we address is how to accomplish this systematically and efficiently.

We have evaluated the approach by implementing a tool, THOR, working on Android. The results on four real-world apps with existing test suites demonstrate that apps are often fragile with respect to certain unexpected events and that our methodology effectively increases the testing quality: Of 507 individual tests, 429 fail when exposed to adverse conditions, which reveals 66 distinct problems that are not detected by ordinary execution of the tests.

Categories and Subject Descriptors

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

Keywords

UI testing; automated testing; mobile apps; Android

ISSTA'15, July 12–17, 2015, Baltimore, MD, USA

http://dx.doi.org/10.1145/2771783.2771786

1. INTRODUCTION

As of May 2015 more than 1.5 million Android apps have been published in the Google Play Store.[1] Execution of such apps is driven by events, such as, user events caused by physical interaction with the device. One of the primary techniques developers apply for detecting programming errors is to create end-to-end test suites (also called UI tests) that explore the UI programmatically, mimicking user behavior while checking for problems. Testing frameworks, such as, Robotium,[2] Calabash,[3] and Espresso[4] are highly popular among Android app developers. As a significant amount of the software development time is often devoted to testing [19], it is not unusual that test suites have high coverage of the source code and incorporate a deep knowledge of the app UI and logic. Furthermore, the result of each single test can be of critical importance to sanction the success of the entire development process, as tests may be used for verifying scenarios in the business requirements.

Nevertheless, due to the event-driven model, only a tiny fraction of the possible inputs is typically explored by such test suites. As the test cases are written manually, they tend to concentrate on the expected event sequences, not on the unusual ones that may occur in real use environments. In other words, although the purpose of writing test suites is to detect errors, the tests are traditionally run in "good weather" conditions where no surprises occur.

Our goal is to improve testing of apps also under adverse conditions. Such conditions may arise from events that can occur at any time, comprising notifications from the operating system due to sensor status changes (e.g. GPS location change), operating system interference (e.g. low memory), or interference by another app that concurrently accesses the same resource (e.g. audio). It is well known that Android apps can be difficult to program when such events may occur at any time and change the app state [13, 14, 22, 29]. A typical example of bad behavior is that the value of a form field is lost when the screen is rotated.

As a supplement or alternative to manually written test suites, many automated testing techniques have been created aiming to find bugs with little or no help from the developer [3, 4, 6, 7, 12, 13, 14, 22, 23, 25, 27, 29]. The primary advantage of such techniques is that they can, in principle, explore the state space more extensively, including the unusual event sequences. However, these techniques generally cannot

[1] `http://www.appbrain.com/stats/number-of-android-apps`
[2] `http://code.google.com/p/robotium`
[3] `http://calaba.sh/`
[4] `http://code.google.com/p/android-test-kit`

provide as high coverage as manual techniques. Moreover, automated techniques mostly operate without any knowledge of the expected behavior of the app, so they typically test only generic correctness properties (such as, the app should not crash with a null dereference exception) and fail to notice more subtle functionality errors.

In contrast to those approaches, we wish to take advantage of existing, manually written end-to-end test suites that are already widely used by app developers. We present an algorithm that systematically injects special events in existing tests to check the robustness in adverse conditions. We thereby leverage the application-specific knowledge and amplify the tests [28, 31].

As observed by Zaeem et al. [29], certain events in mobile apps are associated with a common sense expectation of how the app should respond. For example, suspending and then resuming an app should typically be allowed anytime without affecting the behavior of the app. We use a similar idea to select which events to inject. Zaeem et al. exploit their observation in a model-based testing technique. A limitation of that approach is that it requires a UI model of the app under test and a suitable abstraction of the execution states to determine whether the events cause substantial changes. By leveraging existing test suites, we avoid both problems.

To our knowledge, no previous work has exploited existing tests of mobile apps to get assurance about the app behavior when executed in such adverse conditions. Fard et al. [9] combine existing tests and crawling for web applications, but in a way that requires heuristic regeneration of assertions, whereas we use the existing test assertions unmodified and focus on injecting events that are typically not mentioned in the tests. More fundamentally, our aim is to obtain a systematic exploration of the possible consequences of injecting such events in each test case.

Although the basic idea in our approach is simple, making it work in practice involves several design challenges. Which events are relevant to inject, and when should they be injected in each test case? Ideally, the technique should (1) increase the ability to detect bugs as much as possible, (2) run without a significant slowdown compared to ordinary test suite execution, and (3) provide error messages that precisely indicate the cause of each error being detected, to aid debugging. The first step of our approach is to establish a notion of *neutral event sequences* that may be tailored to individual apps or test cases, with sensible defaults for the Android platform. We then present an algorithm for injecting neutral event sequences, supplemented by strategies for isolating the causes of failures and reducing redundancy.

By generalizing from existing tests, the classification of the problems being detected as either bugs or false positives is naturally subjective. Indeed, in some cases the developer might think that the problem is not important enough to be fixed, for example, if a dialog window disappears when the phone is rotated. To this end, our approach can help the developer by revealing implicit assumptions of test cases that concern the special events.

In summary, our contributions are:

- A methodology for leveraging existing tests to detect bugs that involve unexpected events (Section 3). The methodology relies on the insight that existing tests can be run in adverse conditions to increase the ability to detect bugs in the apps and identify hidden assumptions of the tests.
- An implementation, Thor, designed for Android apps with Espresso or Robotium test suites (Section 4), which includes a selection of neutral event sequences.
- An experimental evaluation (Section 5). We show using 4 real Android apps with existing test suites that our methodology is able to detect bugs and identify hidden assumptions that are not exposed by ordinary test executions. In particular, our technique causes 429 otherwise succeeding tests to fail in adverse conditions out of a total of 507 tests. From those failing tests we have manually identified 66 distinct problems. We estimate that 22 of the 66 problems are critical bugs from the user perspective, where the remaining ones are likely unintended by the app developers but not harmful to the overall functionality of the apps. Among the 22 critical bugs, 18 affect the functional behavior of the app without causing it to crash, thereby demonstrating the advantage of exploiting the application-specific knowledge available in the test suites. Our experiments also show the effectiveness of the failure isolation and redundancy reduction strategies.

2. MOTIVATING EXAMPLE

This section explains our methodology using a concrete, motivating example.

An Android app is structured in various screens, each called an activity and representing a focused component for user interaction. Consider the code in Figure 1, showing a snippet of code from an activity in Pocket Code[5] for Android, an educational app for teaching visual programming.

The `ProjectsListFragment` allows users to manage their projects in a list. A `Fragment` is a piece of an app's UI that can be placed inside an `Activity`. When the activity is put into the foreground, the `onResume` method (line 2) is called on the fragment, which loads the projects from the disk and creates a `ProjectAdapter` for them (lines 6–7). This adapter holds the list of projects and provides a UI element for each entry through the `getView` method that is shown by the activity to the user (see Figure 2(*i*)).

In order to delete a project, the user should long press the entry associated with it, triggering a call to `onCreateContextMenu` (line 9). This method adds the selected project to the checked projects of the adapter (line 10) and displays a contextual menu on the screen (see Figure 2(*ii*)). On this menu, the user can press "Delete", which causes the confirmation dialog initialized on lines 13–18 to appear (see Figure 2(*iii*)). If the "Yes" button is clicked, the checked project is finally deleted (line 22).

The above use case is taken into account by the test `testDeleteCurrentProject` in Figure 3, taken from the original app repository. The test is written using the Robotium test framework. The actions are interleaved with assertions, which check that the state of the UI (lines 7–10) and the app (line 13) conforms with the expected one. A failure of the test reveals an unexpected behavior.

However, a test may succeed when executed in an ordinary manner, giving the developer a peace in mind, although the app may behave differently in a real-world scenario. As we shall see, this is the case for the test in Figure 3.

The behavior of an app depends not only on the sequence of ordinary UI actions (as simulated by the test), but also on external events that may interfere. Such events arise from

[5] `http://github.com/Catrobat/Catroid`

```
1 class ProjectsListFragment extends ... {
2  void onResume() {
3   initAdapter();
4  }
5  void initAdapter() {
6   projects = loadListFromDisk();
7   adapter = new ProjectAdapter(projects);
8  }
9  void onCreateContextMenu(MenuInfo info) {
10  adapter.addCheckedProject((...)info.pos);
11 }
12 void showConfirmDeleteDialog() {
13  list = new OnClickListener() {
14   void onClick(...) {
15    deleteCheckedProjects();
16   }
17  }
18  ....setPositiveButton(yes, list);
19 }
20 void deleteCheckedProjects() {
21  for (int pos:adapter.checkedProjects()) {
22   deleteProject((ProjectData)
23    getListView().getItemAtPos(pos));
24  }
25 }
26 }
27 class ProjectAdapter extends ... {
28  Set<Integer> checkedProjects = new ...;
29
30  View getView(final int pos, ...) {
31   findViewById(PROJECT_CHECKBOX)
32   .setOnCheckedChangeListener(
33    new OnCheckedChangeListener() {
34     void onCheckedChanged(boolean checked) {
35      if (checked) {
36       checkedProjects.add(pos);
37      } else {
38       checkedProjects.remove(pos);
39      }
40     }
41    });
42   ...
43  }
44 }
```

Figure 1: Snippet from the Pocket Code app.

incoming calls, clicks on hardware buttons, such as, the home button or the earphone media keys, device rotations, other apps trying to acquire the audio focus, the user plugging out his earphones, etc. Manually written test cases rarely take such events into account.

For this example, consider what happens during the execution of `testDeleteCurrentProject` if the app is sent to the background and resumed, as the result of, for example, the user long pressing the home button and returning to the app (see Figure 2(*iv*)). During this process, the current activity is paused, which technically means that the activity can still be partially visible, but can be left soon or, as in this case, simply resumed. The app is notified of this change by means of a series of method calls, and it is supposed to commit all the changes and release all its resources.[6] The app will be resumed when it is put back to the foreground.

This sequence of events reveals a bug in the implementation of the deletion feature. If the app is paused while the confirmation dialog is shown, it will show the same dialog when

[6]`http://developer.android.com/reference/android/app/Activity.html`

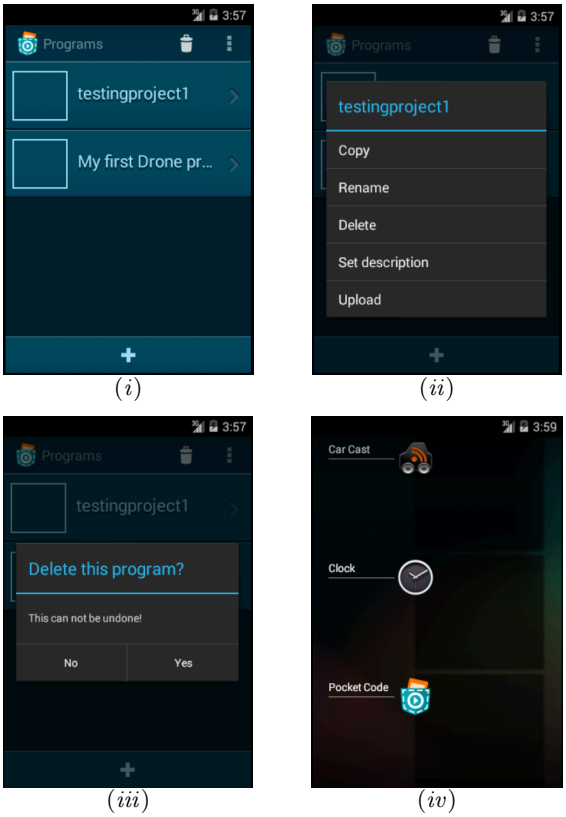

Figure 2: Snapshots from Pocket Code during the deletion of a project (*i-iii*), and a home button long-press to show the open apps (*iv*).

resumed. However, the referenced adapter gets recreated from scratch (line 3) with a fresh set of `checkedProjects` (line 28), causing the project deletion to fail silently.

This problem is not revealed by the test in an ordinary execution. However, it is caught by the assertion on line 7 in Figure 3 if the test is exposed to an adverse condition immediately after the UI action on line 5, which simulates the app being sent to the background and resumed.

3. METHODOLOGY

We assume to be given an app with a UI test suite. A UI test is a small program that uses the primitives offered by a test framework to interact with the UI of an app. The instructions usually contained in a test may trigger events on the UI (as in lines 3–6 of Figure 3), inspect the UI or directly interact with the internal state (as in line 2 where projects are created and added to the app, circumventing the UI), await timers or other conditions (typically to make sure some asynchronous task is completed before the test proceeds), and assert desired properties of the internal state or the UI (as in lines 7, 9, and 13).

Mobile apps involve many kinds of events: system events that are triggered by sensors, by changes in the environment, or by other apps, as well as actions initiated by the user (e.g. docking, tapping). Events that are triggered in typical test suites tend to concentrate on user events that directly concern the graphical UI, not on other kinds of user interactions (e.g. rotating the device) or system events. For example, although

```
1 public void testDeleteCurrentProject() {
2  createProjects();
3  clickOnButton("Programs");
4  longClickOnTextInList(DEFAULT_PROJECT));
5  clickOnText("Delete");
6  clickOnText("Yes");
7  assertFalse("project still visible",
8      searchText(DEFAULT_PROJECT));
9  assertTrue("project not visible",
10     searchText(OTHER_PROJECT));
11 String newCurrent = ProjectManager.
12     getCurrentProject().getName();
13 assertNotSame("project not deleted",
14     DEFAULT_PROJECT, newCurrent);
15 }
```

Figure 3: A test from Pocket Code that checks the deletion feature of the projects list (simplified for presentation).

device rotations are common in real use, they are rare in test suites. Among the approx. 3 500 events that appear when executing the 507 tests that we consider in Section 5, only 7 are of this kind. This means that the behavior of the apps in presence of such kinds of events largely remains untested. One likely reason is that the programmers may be less aware of those kinds of events, or that there is no obvious place to test them among the ordinary test cases. Another reason may be that some testing frameworks do not provide the necessary primitives to trigger such events; for example, Robotium does not support simulating a click on the home button.

Neutral event sequences. We say that a sequence n of events is *neutral* with respect to a given test if injecting n during the test is not expected to affect the outcome.

As an example, the event sequence *Pause–Stop–Restart*, which consists of the Android life-cycle events that occur when the user long presses the home button and then returns to the app, is neutral with respect to the test in Figure 3. Whether an event sequence is neutral or not is of course subjective – and must in the end be decided by the app developer – but a common sense expectation often exists, as observed by Zaeem et al. [29] who use a related notion called user-interaction features (see Section 6).

Notice that the property of being neutral may depend on the individual test. For example, a loss of WiFi connection followed by a 3G signal recover can be neutral for most tests but not for one that checks that data is only uploaded when WiFi is available.

The identification of neutral event sequences is pivotal for our methodology. Our idea is to systematically issue neutral sequences of events in specific moments of a test execution. If a test assertion fails, it is safe to issue a warning reporting that the app is misbehaving. The neutral event sequences of interest are those that complement the ordinary user events that concern the graphical UI of the app.

Injection points. Not all program points in a test are suitable locations for injecting events. First, there is no reason to inject the same events in consecutive instructions that merely inspect the UI or internal app state. Second, to ensure that the simulated event sequence with the injected events is realizable in practice, we should not inject events in the middle of a sequence of instructions that programmatically modify the app state. Third, to avoid introducing nondeterministic outcomes of the tests, we should not inject events right after sleep instructions. Altogether, this leaves us with the following design choice, which is also simple to implement: Injection of events takes place immediately after test instructions that trigger events, identified by the use of primitive operations from the test framework. This corresponds to the program points after each of lines 3–6 in our example in Figure 3. More precisely, we inject events after a test has triggered an event and the corresponding event handlers have completed (i.e. when the event queue becomes empty), and we delay execution of the remaining test instructions until the event handlers of the injected events have completed.

The basic algorithm. Given a set of test cases T and a list of neutral event sequences N, we execute each test in T using a modified testing framework that injects *every* event sequence from N (in the given order) at *every* injection point (as defined above). In other words, we combine all the event sequences in N into one neutral event sequence (neutral event sequences are trivially closed under concatenation) and inject it aggressively. Naturally, we only consider test cases that do not fail in ordinary executions.

In the test from Section 2, an injection point is reached after the clicks on the "Programs", "Delete", and "Yes" buttons and after the long click on the project (lines 3–6 in Figure 3). Our algorithm injects the neutral event sequences at each of these injection points, in particular, at the delete confirmation dialog (Figure 2(*iii*)), thereby triggering the error caught by the assertion in line 7 in Figure 3.

Detecting multiple errors with each test. A potential limitation of the basic algorithm is that a test stops as soon as an assertion fails or the app crashes, which may shadow other errors. For example, the basic algorithm never reaches beyond line 7 in the test in Figure 3, although the later assertions might potentially fail with another choice of injections. For this reason, we slightly extend the basic algorithm: Whenever an assertion fails or the app crashes, we rerun the test but only perform event injections at program points after the failed assertion or app crash. We keep rerunning as long as the test fails.

With the algorithm presented above, and assuming no assertion failures or app crashes, every test is subjected to a large number of additional events, but we still only execute each test once. We choose this approach to minimize the slowdown compared to ordinary test execution, as restarting tests is likely more time consuming than executing the injected events [7]. On the other hand, it is possible that we thereby miss errors that could be detected with a less aggressive strategy. Some errors may only manifest with very specific combinations of injections. We hypothesize that this is mostly a theoretical concern: *Few additional errors will be detected in practice if we inject only a subset of the neutral event sequences and use only a subset of the injection points.* We test this hypothesis experimentally in Section 5.

Isolating the causes of failures. As stated in Section 1, we aim not only to detect as many errors as possible and avoid a significant slowdown compared to ordinary test execution; we also want useful error messages to help the developer understand the causes of the errors being found. Although each failing test execution comes with a concrete trace of the events involved, the fact that we have chosen an aggressive

strategy for injecting events means that it may not be clear which of the injections are the critical ones. Our approach to address this problem is based on the following small-scope hypothesis [15]: *Most errors that can be detected by injecting neutral event sequences can be found by injecting only* one *of the neutral event sequences and at only* one *injection point.* We also test this hypothesis in Section 5.

On this basis, we choose to apply a simple variant of delta debugging [30]. Whenever a test fails (using either the basic or the extended algorithm presented above), we perform the following steps that attempt to isolate a single neutral event sequence and a single injection point to blame:

1. We first perform a binary search through N, in each step eliminating half of the neutral event sequences, and each time with the same injection points, until we find a single neutral event sequence n that leads to the same failure.
2. If such an n exists, we perform a binary search through the sequence of injection points, repeatedly eliminating half of them and each time injecting only n, until we find a single injection point triggering the failure. As relevant injection points are likely close to the failure, in each iteration we inject n in the half that is closest to the failure point.

Of course, many heuristic variations can be conceived, for example, replacing the first binary search with a linear scan if N is small, or first trying neutral event sequences that are known to be involved in many errors in prior test executions.

The isolation algorithm may fail to find a single injection to blame. The small-scope hypothesis might not apply because a combination of injections is needed to expose the error, and flakiness of tests (see [21]) may divert the search from the failure injection point. Still, our approach reduces the sets of injections to consider during debugging.

Reducing redundancy. The choice of rerunning the tests to enable detection of multiple errors with each test has two potential drawbacks. Although more assertion failures or app crashes may be encountered, and hence more error messages are emitted, some of these may have the same ultimate cause and thereby do not reveal any additional bugs in the app. Also, rerunning tests takes additional time.

As an example, consider the test in Figure 4, which is also part of the test suite for Pocket Code. The test is checking a similar use case to that of Figure 3, but deletes another project than the currently selected one, and also checks that the project is in fact deleted from the disk. This test fails for the same reason as the test from our motivating example when executed using our basic algorithm, resulting in redundant warnings. Furthermore, for the extended algorithm that aims to detect multiple errors with each test, it also leads to superfluous test executions, because the technique reruns a test for every warning being produced.

We propose the following mechanism to heuristically omit certain injections, for use in situations where the developer would like to concentrate on the error messages that are likely caused by distinct bugs and to get the error messages faster. During execution of the tests, we build a cache of abstract states. An abstract state is added each time some event e has been processed and some neutral event sequence n is about to be injected. Each abstract state consists of an abstraction of the UI state together with e and n. (Such abstractions of UI states are common in the literature on model-based testing of mobile, web, and GUI applications [3, 6, 7, 8, 9, 11, 24, 29].) Now, we simply skip the injection if the abstract state already appears in the cache. For the example, this means that the injection, that cause the redundant warning from the test in Figure 4 to be triggered, is omitted. The intuition behind this choice is that any bug that may be found by that injection would likely have been found already when the abstract state was added to the cache.

```
1 public void testDeleteProject() {
2  createProjects();
3  clickOnButton("Programs");
4  longClickOnTextInList(PROJECT_1);
5  clickOnText("Delete");
6  assertTrue("Dialog title is wrong!",
7      searchText("Delete this program?"));
8  clickOnText("Yes");
9  assertFalse("project still visible",
10     searchText(PROJECT_1));
11 ArrayList<String> projectList =
12     UtilFile.getProjectNames(...);
13 boolean deleted = true;
14 for (String project : projectList) {
15  if (project.equalsIgnoreCase(PROJECT_1)) {
16   deleted = false;
17  }
18 }
19 assertTrue("project not deleted", deleted);
20 }
```

Figure 4: Another test from Pocket Code, illustrating potential redundancy (cf. Figure 3) when injecting neutral event sequences.

This mechanism may obviously cause some bugs to be missed. Injecting a single neutral event sequence in some abstract state may falsify multiple assertions, so a bug can be missed if an injection is skipped due to the cache mechanism. In Section 5 we quantify the trade-off between reducing redundancy and maintaining the ability to detect bugs.

4. IMPLEMENTATION

We have implemented the testing technique in a tool THOR[7] designed for Android apps and Robotium test suites. The tool lets the app developer select the set of tests to run, the set of neutral event sequences to take into account, and whether or not to enable the different variations of the algorithm presented in Section 3. During the execution, the tool provides an interactive visualization of the issues found, including all the necessary information to reproduce them, allowing further investigation.

THOR executes the tests on an emulator running Android KitKat 4.4.3 (from the Android Open Source Project). A controller component on the host machine guides the executions of the tests. The test execution is parallelized by using multiple Android emulator instances. The controller is implemented in Node.js[8] and is managed via a web interface. We have manually instrumented the Android framework by adding hooks that allow the tester component to control the execution of a test, detect injection points, and perform the injection of neutral event sequences.

The redundancy reduction strategy requires a way to abstract runtime states. Much like previous work on model-

[7] `http://brics.dk/thor/`, named after the storm god.
[8] `http://nodejs.org/`

based testing, our implementation abstracts away from data stored in activities, on disk, and by content providers, preserving only information about the structure of the UI as represented by the Android *view hierarchy*. More precisely, our abstraction consists of the tree of objects, which are all instances of the `View` class, represented by their class names and associated event handlers, while ignoring all concrete string values, screen coordinates, timestamps, etc. A convenient property of this choice of abstraction is that it can be computed quickly, which is important for its use in reducing redundancy.

Selecting neutral event sequences. The Android framework provides ample opportunities for selecting neutral event sequences. It contains well over 60 different services to manage the different resources available on the system (e.g. `IActivityManager`, `IAudioService`, `IAlarmManager`, `IBackupManager`, `ICameraService`, `IDropBoxManagerService`, `ILocationManager`, `IMountService`, `ITelephony`, `IUsbManager`, `IVibratorService`, `IWifiManager`, and `IWindowManager`). Most importantly, the activity manager is responsible for the life-cycle of the activities running on the system, which includes creating, pausing, resuming, and destroying activities, depending on external factors (e.g. low memory) and user interactions (e.g. home button click, screen rotation). As another example, the audio service manages the speakers and earphones, including granting permission to emit sounds (also called *audio focus*). Apps are allowed to use the functionalities offered by some of the services using remote procedure calls with a thread-migration programming model. In this way, apps can invoke service methods as if they were running on a thread of the service process. Moreover, services can invoke methods of apps. This opens for many different ways in which apps can be influenced by events from services, or even of other apps. Any event that causes the internal state of a service to change may affect the app as well: the service may call a method on the app process depending on its state, or the app may call a method on the service process that depends on the service state. Hence, internal service state changes can potentially alter the behavior of an app. Any sequence of events that should not influence the outcome of a test case is of interest.

For our experiments we focus on events concerning the *activity manager* and the *audio service* as these are widely used. The activity manager is particularly relevant, as all apps must interact with it and cannot ignore the life-cycle events. Many common high-level events (e.g. incoming call, device rotation, docking) cause life-cycle events. According to the Android documentation, the activity manager always issues events to an app as a consequence of the changes of the activity status. The possible events and the associated state changes are shown in Figure 5.

Example. For our example in Section 2 the activity manager triggers a pause event by calling the method `schedulePause` on `ApplicationThreadProxy`, which eventually dispatches the event to the current activity, whose default implementation dispatches it further to its contained fragment by calling `onPause` on the `Fragment` class.

Other services, including the audio service, are only relevant for apps that use particular hardware components. Some of the events that are concerned with the audio service are *audio focus request*, which causes a loss event to be sent to an app whenever another app is requesting to use the audio, and *abandon audio focus*, which causes an app to be notified that the audio is available.

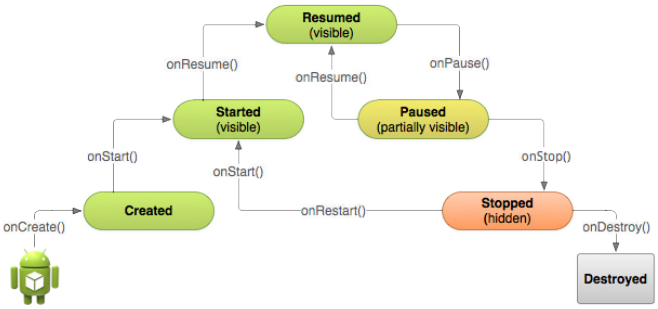

Figure 5: The Android activity lifecycle (from `http://developer.android.com/`).

We choose an initial collection of neutral event sequences for Thor. Concerning the events related to the activity manager we select the sequences of events in Figure 5 that cycle back to the Resumed state (assuming an implicit edge from Destroyed to Created due to the user restarting the app).

Pause–Resume (PR) – issued in some devices when the screen is turned off and on.

Pause–Stop–Restart (PSR) – issued when the user long-presses the home button to show the open apps and then returns to the app.

Pause–Stop–Destroy–Create (PSDC) – issued when the phone is rotated or docked.

Technically, PSR also contains a Resume event, and PSDC also contains Start and Resume, but we omit these in the event sequence names for presentation.

We have excluded other cycles because they are not neutral, e.g. Pause-Stop-Create. Indeed they cause the app to be killed by Android, therefore invalidating any expectations on the outcome of the test.

Concerning the audio service, the following examples illustrate neutral event sequences:

Request-Abandon audio focus – an app requesting the audio focusing and abandoning it.

Request(May duck)-Abandon audio focus – the same scenario as before, but with the app still being allowed to play at reduced volume.

Audio becoming noisy–Media key play – pausing the media due to high volume and resuming.

Notice that all of these event sequences satisfy the definition of being neutral (Section 3). It is easy to extend Thor with neutral event sequences that concern other services. However, since the apps used for the evaluation do not depend on, e.g., location services, carrier identification, or the connection type, we have not implemented such, although they may be of interest for other apps. For example, the following events could be taken into account:

GPS precision loss–Recall – issued in case the phone reaches a zone where the GPS is less precise and then returns.

Carrier change – issued when the phone connects to another GSM cell during roaming.

3G–WiFi–3G – issued when the network connection switches from 3G to WiFi and back (the converse scenario may naturally also be included).

5. EXPERIMENTAL EVALUATION

We evaluate THOR by conducting an empirical study on real Android apps to investigate whether THOR is capable of exposing bugs that are undetected by ordinary executions of existing test suites, and at what extra cost in testing time. We divide this into five research questions:

Q1 (error detection) Several aspects of the error detection capability deserve attention. To what extent is it possible to trigger failures in existing test suites by injecting neutral event sequences? Different failures may have the same ultimate cause. How many distinct problems in the apps do the failures identify? How many of the problems are likely perceived as critical bugs from the user's perspective, and how many of these affect the functional behavior of the app but without causing it to crash? How many failures are missed if disabling the rerun of a test at each failure?

Q2 (execution time) What is the slowdown for the different modes of THOR compared to ordinary test execution? Specifically, how much extra time is spent when enabling the rerun of a test at each failure, and how much time is saved when enabling the redundancy reduction mechanism?

Q3 (redundancy reduction) When enabling the redundancy reduction mechanism, what is the effect on the number of test failures and critical bugs being detected?

Q4 (failure isolation) Is the failure isolation mechanism effective in finding a neutral event sequence and injection point for each failure?

Q5 (hypotheses) Our design in Section 3 was based on two hypotheses. Is it correct that few additional errors will be detected if we inject only a subset of the neutral event sequences and use only a subset of the injection points? Also, is it correct that most of the errors can be found by injecting a single neutral event sequence in a single injection point?

5.1 Experiments

UI testing frameworks for Android are popular: Robotium[9] and Calabash[10] each count more than 850 stars and 400 forks on GitHub, and Espresso[11] has recently been added to the Android Support Library. In addition, several companies (e.g. Appurify, Xamarin, TestDroid) are offering Android cloud testing facilities. This gives evidence that UI testing is widely used in practice. Nevertheless, we regrettably only have access to a few nontrivial apps with UI test suites, since most are closed source as also noted by Fard et al. [9]. Our experiments are therefore based on a case study of 4 open-source apps. The experiments are performed on a 2.4 GHz Intel Core i5 laptop with 8 GB RAM using the x86 Android emulator and a pool of 3 emulator instances.

Table 1 shows some characteristics of our benchmark apps. AnyMemo[12] (AM) is an app designed for learning different languages, computer related topics, etc. through flashcards. Car Cast[13] (CC) is a simple podcast player that uses the media player facilities and reacts to several audio related external events. Pocket Code (PC) is the app we used in Section 2 as motivating example. Finally, Pocket Paint[14] (PP) is a paint program with various tools for editing images.

As evident from the size of the UI test suites, the app developers have put a considerable effort into writing UI tests. However, since running all tests can take a considerable amount of time (e.g. more than 2 hours in the case of PC), it is not uncommon that parts of a test suite are out of sync with the app code as the app is being developed. Moreover, the developers may run the tests in specific environments, which are unknown to us, for example, with a particular version and configuration of the Android emulator ecosystem. For these reasons, it is not surprising that some of the tests fail when we execute the test suites even without injecting any new events. We mark those tests as *unstable* and exclude them from the evaluation of THOR. When executing the resulting stable tests with THOR, we encounter no failures that can be attributed to environment settings.

In order to answer Q1–Q5 we conduct three experiments.

Experiment 1. We run THOR on each test suite, using all the variations presented in Section 3 (i.e. enabling or disabling the rerunning of a test when a failure occurs, the failure isolation technique, and the redundancy reduction mechanism). The configuration uses the neutral event sequences concerning the activity manager and the audio service mentioned in Section 4. We manually classify each failing test to determine the root cause, in order to group failures that are caused by the same bug and identify which bugs are likely critical from the user's perspective. As such manual classification is time consuming, we settle for a large representative subset of the failures. This experiment allows us to answer Q1–Q3.

Experiment 2. To answer Q4 we measure the success rate of the failure isolation strategy when applied to all tests that fail in adverse conditions.

Experiment 3. To test the hypotheses in Q5 we develop an algorithm that injects a random subset of the neutral event sequences and use a random subset of the injection points (using a uniform distribution), and execute it 50 times per test case. For each test, we compare the failures found by these random runs with the ones found by the ordinary execution of THOR in Experiment 1. Next, for each failure detected by THOR where the failure isolation mechanism was unable to isolate a single neutral event sequence and a single injection point, we try every possible single neutral event sequence and single injection point in turn to check whether one actually exists. To limit the time for conducting this experiment, we use a randomly selected subset of the tests.

5.2 Results

Q1. The results from running THOR on the four apps in Experiment 1 show that as many as 429 tests fail out of a total of 507 when run in adverse conditions. This amounts to 1 770 failures counted as distinct failing test assertions or app crashes, none of which appear in ordinary execution of the tests. These numbers witness the good weather assumption about the environment in which tests are traditionally executed, and clearly demonstrate that our technique is able to trigger failures in existing test suites.

The majority of the failures are due to the PSDC event sequence (AM: 91%, CC: 100%, PC: 61%, PP: 98%), whereas

[9] http://code.google.com/p/robotium

[10] http://calaba.sh

[11] http://developer.android.com/tools/testing-support-library/index.html#Espresso

[12] http://code.google.com/p/anymemo

[13] http://github.com/bherrmann7/Car-Cast

[14] http://github.com/Catrobat/Paintroid

Table 1: The apps used in the evaluation of Thor.

App	Version	Rating	Downloads	LOC Source	LOC UI Tests	Tests Stable	Tests Unstable	Coverage Stable	Test time Stable
Pocket Code (PC)	Aug 07, 2014	3.6/5	50K–100K	34K	19.4K	340	142	65%	2h 21m 17s
Pocket Paint (PP)	Sep 19, 2014	3.7/5	10K–50K	6.6K	3.8K	121	11	76%	45m 25s
Car Cast (CC)	Jul 11, 2014	4.1/5	100K–500K	6K	0.5K	17	1	48%	12m 47s
AnyMemo (AM)	Apr 03, 2014	4.5/5	100K–500K	20.1K	1.2K	29	19	31%	12m 17s

For each app, we show its version, the rating on the Android Play Store, and the number of downloads. The LOC column shows the number lines of code for the app source and the UI tests. The Tests column shows the number of tests that were stable vs. unstable (i.e. consistently or randomly failing) in our test environment. The last two columns show the code coverage and the time it takes to execute the stable tests.

Table 2: Failures and bug categories.

		Logical			Critical UI		UI			Other	
App	Crash	Silent fail	Not persisted	User setting lost	Operation ignored	Unexpected screen	Not persisted	Element disappears	Spec. fail	Brittle test	Emulator issue
Pocket Code	1 (9)	7 (42)	-	1 (6)	-	4 (51)	2 (54)	14 (104)	3 (5)	-	-
Pocket Paint	2 (45)	-	1 (4)	4 (42)	1 (25)	-	-	9 (131)	-	1 (103)	2 (2)
Car Cast	1 (7)	-	-	-	-	-	-	5 (18)	-	-	-
AnyMemo	-	-	-	-	-	-	4 (24)	4 (15)	-	-	-

The number of bugs in each category (with the corresponding number of failures shown in parentheses). **Logical**. *Crash*: e.g. `NullPointerException` or bad use of Android Support Library. *Silent fail*: e.g. the example from Section 2. *Not persisted*: e.g. internal static field `savedPictureUri` is reset in PP. *User setting lost*: e.g. the selected brush size is lost in PP. **Critical UI**. *Operation ignored*: e.g. cannot draw in PP. *Unexpected screen*: e.g. navigation to unexpected screen, activity, or fragment. **UI**. *Not persisted*: e.g. button disabled, text cleared, or checkboxes unchecked. *Element disappears*: e.g. menu or dialog disappears, or dropdown closes. *Spec. fail*: less important assertions, e.g. a wrong text such as "Delete these programs" instead of "Delete this program". **Other.** *Brittle test*: invalid reference to a `View` in the test due to app relaunch. *Emulator issue*: e.g. emulator unable to perform an action.

PR accounts for the lowest amount of warnings (AM: 5%, CC: 0%, PC: 16%, PP: 1%). This is expected, since the three neutral event sequences that consist of life-cycle events in turn invoke more event handlers in the app. For example, PSDC is the only one that causes the activity to be destroyed and recreated (see Figure 5), meaning that, for example, inadequate persistence is more likely to expose a problem.

In the experiment, no failures originate from audio related event sequences. Thor is able to dynamically detect the specific tests that uses the audio so that it only injects audio related events when relevant (similar to the relevant events detection by Machiry et al. [22]). Hence, we only inject audio related events in few tests of CC, which is the only app that uses audio.

Our manual classification of a random selection of 682 of the 1 770 failures gives the categorization depicted in Table 2. As discussed, a single bug may cause multiple test failures. The table presents the number of distinct bugs and the corresponding number of failures (in parentheses) in each category that we have identified. The categories *Logical*, *Critical UI*, and *UI* correspond to real problems, whereas *Other* corresponds to technical issues originating from a specific coding style used in some tests that is brittle towards app relaunches (this can be avoided with minor rewriting efforts) or from the flaky nature of UI tests. The table further divides the different categories and also includes examples of typical kinds of errors being found.

From the table it follows that we have identified 66 distinct problems in the apps from the 1 770 generated failures. On average, each problem is spotted approx. 10 times. This is not surprising; different tests often visit the same UI components and will therefore encounter the same failures, if, for example, one of these components disappears when the device is rotated. Later in this section, we separately evaluate how the redundancy reduction mechanism affects these numbers (cf. Q4).

Table 2 also shows that 22 out of 66 distinct problems detected by Thor fall into the categories *Logical* and *Critical UI*, which we conservatively classify as being critical bugs from the user's perspective.

The bug in PC that we described in Section 2 is among the ones detected by Thor in the category *Silent fail*. Another bug in PC causes the app to navigate to a wrong screen, because the activity does not persist the active fragment. In PP, user settings, such as, the currently selected tool and brush size, are not properly persisted, causing them to be reset under certain life-cycle events. A list of some other bugs that Thor has revealed is shown in Table 3.

Overall, the number of errors is dominated by the *UI* category, comprising, for example, simple UI widgets that disappear from the screen. Many of these problems are likely to be ignored by app developers, since the improvement of the app does not match the implementation effort needed to solve the problems. As an example, it is typically not considered a serious problem that a menu or dialog closes upon e.g. rotation. On the other hand, some of the bugs in the *UI* category involve text fields being reset, which can result in a negative user experience. In a few cases, the choice of the neutral event sequences to inject may depend on the individual test cases. For example, PP has a full-screen mode, which is closed when the device is rotated. This situation is detected by Thor. If this behavior is intended, the PSDC event sequence is not neutral, meaning that the test failure concerned has revealed a hidden assumption of the test. We emphasize that all the issues in the *UI* category are conservatively counted as non-critical.

The fact that Thor is able to detect functional bugs, such as, incorrect persistence of user settings, proves that our approach is capable of exposing bugs that are undetected by ordinary execution of the tests. Also notice that among the 22 distinct bugs that damage the user experience, only 4 are crashes. This shows the value of leveraging existing

Table 3: Examples of errors detected by Thor.

Pocket Code
- Null pointer crash in the copy program dialog. - The OK button gets disabled on the new program dialog. - A fragment (Scripts/Looks/Sounds) is randomly opened. - The user setting Show Details is lost. - The selected project name disappears from the action bar title. - The app randomly navigates to the program list screen and opens the copy program dialog. - Clicking "OK" on the copy dialog of a script has no effect. - Clicking on "+" in the bottom bar navigates to a wrong screen. - Title changes from "Delete this program?" to "Delete these programs?". - The bottom bar appears. - Created program element disappears when dragged. - Deletion of various elements fails silently. - Various dialogs close inadvertently.
Pocket Paint
- The dialogs tool info and tool settings crash due to missing empty constructors. - The position of the canvas on the screen is reset. - When drawing on the canvas, the strokes are not retained. - The selected tool, color, and brush is not persisted. - The full-screen mode gets disabled. - The "Undo" button gets enabled although no drawing actions have been performed. - Various dialogs close inadvertently.
Car-Cast
- Crashes when an open dialog (loader) is dismissed. - Various dialogs close inadvertently.
AnyMemo
- Content of a text field disappears. - Review dialog closes. - Buttons ("Forgot", "Easy", etc.) disappear.

tests, compared to automated testing techniques that focus entirely on app-agnostic error conditions.

The use of existing tests also has advantages compared to the approach by Zaeem et al. [29] (see Section 6). For example, the bug presented in Section 2 does not cause the UI to change and is hence unnoticed by their tool. As a small additional experiment, we run Thor on CC in a mode where it also raises warnings if the UI changes due to an injection of a neutral event sequence (based on a comparison of screenshots). This roughly doubles the number of warnings but reveals no new bugs.

In order to determine the number of failures that are missed by disabling the rerun of a test at each failure, we perform a case study on PP where we manually classify each failing test (with rerunning disabled) to determine the root cause. Our results show that the basic algorithm without the rerunning extension detects only 8 of the 17 distinct bugs.

Q2. Table 4 presents the slowdown for all the variations of our algorithm compared to ordinary test executions (cf. Table 1). The slowdown of our basic strategy is 0.99–1.38x, which is competitive to an ordinary test execution. We note that there is a small overhead for injecting events during the test execution, but nonetheless, running test suites in adverse conditions can be faster (as in the case for PP), because failures cause test cases to exit early.

When Thor is run with the rerun extension, the execution time increases significantly (2.11x–4.70x). This increase is expected, since test suites with many failures will have many reruns. Such reruns are expensive, especially when the test cases are long (some tests from our study trigger more than 100 UI events). However, this mode is suited for situations where the developer is interested in learning all the failures of

Table 4: The slowdown when running in adverse conditions.

Strategy	AM	CC	PC	PP
Basic	1.05x	1.21x	1.38x	0.99x
Enable rerun failing tests	2.11x	3.09x	4.70x	3.70x
Enable redundancy reduction	1.02x	1.30x	1.57x	1.17x
Enable both	1.73x	1.93x	3.46x	2.04x

the test suite in one execution. In many cases the developer will be satisfied with learning one failure in each test case, as provided by the basic algorithm. Eventually, when the initial bugs have been fixed and only few failures remain, the rerun mode can be used with little overhead.

The redundancy reduction mechanism aims to eliminate redundant failures. In doing so, it trades part of the bug detection capability for performance. When few duplicates are present, the overhead of building the cache of abstract states outweighs the benefits gained by reduction. From Table 4 it follows that the mechanism is successful in improving the performance when there are many duplicates, thereby alleviating much of the overhead of the rerun extension.

Q3. According to Experiment 1, the redundancy reduction mechanism works best when combined with the rerun extension, which tends to find more failures that have the same cause. For example, PP suffers from duplication of PSDC-related failures when executed using this algorithm (recall that 98% percent of the failures in PP are due to PSDC). However, when executed using the redundancy reduction mechanism, the number of PSDC-related failures reduces from 350 to 75, resulting in significantly fewer reruns. Importantly, a 79% reduction of the number of failures does not cause a similar degradation in the number of distinct bugs being detected: 14 of the 17 distinct problems in PP are detected when redundancy reduction is enabled. In combination with the results showing the speedup obtained by this mechanism (as discussed in relation to Q2), we conclude that the redundancy reduction mechanism provides an effective compromise between speedup and bug detection capability.

Q4. Our results from conducting Experiment 2 show that the failure isolation mechanism is capable of successfully identifying one single neutral event sequence and injection point for all except 5 failures. Three of the unsuccessful isolations succeeded in identifying the neutral event sequence, but was unable to reduce the set of injection points. This situation occurs when the binary search fails to spot the error in any iteration. The remaining two attempts also successfully identified the relevant neutral event sequence, and was able to reduce the number of injection points from 49 and 27 to 3 and 14, respectively.

Q5. Experiment 3 provides us with the necessary data to test our two hypotheses. From a total of 13 784 random runs coming from 144 different test cases and running for a total of 94 hours (compared to 4 hours and 24 minutes for our ordinary execution of Thor on all the 507 test cases), we observe a total of 7 810 failures, 2 783 of which are duplicates. We here consider two failures to be the same if they have the same exception and they are raised in the same test case. Only 56 of the 5 027 distinct failures obtained in random runs are not already detected by our basic algorithm. This number reduces to 26 if we consider the total number of

distinct failures without distinguishing between the test cases that raise them, but only look at the actual exception. We point out that this can only give a rough estimate of the actual number of bugs being detected by the two techniques, because the same failure may concern distinct problems. Still, these results support our first hypothesis, namely that only few more errors would be detected if we inject all the possible subsets of neutral sequences in all the possible subsets of injection points.

Concerning the second hypothesis, the results of Experiment 3 show that a high percentage of the failures raised by the basic algorithm can be eventually reduced to a single event and a single injection point to blame: only 26 out of 324 failures used in the experiment need a more complex combination of injected events.

6. RELATED WORK

Numerous techniques and tools have been developed to support software testing, specifically for mobile apps, and we here discuss the most closely related work.

The use of existing test suites is a key property of our technique for guiding exploration and providing test assertions that specify the intended behavior of the app being tested. This idea of leveraging existing tests has been applied before. Xie and Notkin [26] infer operational abstractions from existing unit tests and then generate new tests that attempt to violate these abstractions. Fraser and Zeller [10] similarly infer parameterized tests from ordinary unit tests. These techniques are not immediately applicable to our setting with event-driven applications and errors that involve events that are typically not mentioned in the original tests.

The Testilizer tool for web application testing by Fard et al. [9] uses a technique more closely related to ours. By extending human-written test suites with automated crawling and heuristics for assertion regeneration, they achieve an improvement in fault detection and code coverage. In comparison, our use of neutral event sequences does not require random crawling and assertion regeneration. Zhang and Elbaum [31] use test amplification for validating exception handling, but do not benefit from the existing assertions in the tests.

Many automated testing techniques aim to explore apps by variations of random testing, or crawling, without the use of existing test suites [3, 4, 6, 7, 12, 13, 14, 22, 23, 25, 28, 29]. This has the practical advantage that it is easier to perform large-scale experiments on 1000s of apps, as these are immediately available, unlike their UI tests. More fundamentally, since these testing techniques cannot take advantage of the test-case specific assertions, they often use code coverage as a proxy for error detection capability and are restricted to detecting application-agnostic error conditions, such as, app crashes. Assuming that our preliminary experimental results generalize – recall that 18 of the 22 critical bugs found by Thor are *not* crashes – this means that random testing techniques can only see the tip of the iceberg regarding app errors that arise in adverse conditions.

Other error detection techniques use notions of neutrality similar to ours, either implicitly or explicitly. LeakDroid [27] repeatedly executes event cycles that should have a neutral effect on the resource usage, while monitoring the execution to identify potential leaks. Orion [18] performs semantics preserving mutations of a program to test the correctness of optimizing compilers. The Quantum tool by Zaaem et al. [29] is based on a notion of user-interaction features, which are actions that are associated with a common sense expectation of how the app should respond. Neutral event sequences are related to user-interaction features where the common sense expectation is that the app behavior should be unaffected by the actions. As discussed in Section 1, Quantum requires a UI model of the app and a suitable abstraction of the execution states, instead of leveraging UI tests.

The fact that life-cycle events are a major source of programming errors is also noted by Hu et al. [14]. Their tool, AppDoctor, performs fast random testing by triggering low-level event handlers directly and then attempts to eliminate false positives by faithfully simulating the high-level user events (e.g. emitting a touch event Down, waiting 3 seconds, and then emitting a touch event Up, instead of directly invoking the Long-press event handler). Thor injects events via the Android framework, not by invoking event handlers directly. This may in principle still lead to false positives, however, we have found no false positives among the 1 770 warnings in our experiments.

Another kind of "adverse conditions" is environment failures, which are the focus of stress testing tools, such as, VanarSena [25], Puma [11], and Caiipa [20]. These tools aim to expose bugs by inducing faults in network response (e.g. replacing an actual response by HTTP 404), and fuzzing device configuration, wireless network conditions, etc. Some of the device specific events triggered by Thor are related to this approach, for example, the neutral event sequence 3G–WiFi–3G.

Errors in event-driven apps may also be caused by unexpected nondeterministic ordering of events, as studied in recent work on race detection [12, 23]. Such techniques require manual investigation to identify the harmful races and tend to produce many false positives. None of the bugs discovered by Thor involve races.

Other techniques use symbolic execution [4, 16], which can potentially reach the challenging targets in the app, but are difficult to scale and do not exploit the information present in UI tests. Finally, existing tools that are based on static analysis (e.g. [5, 17, 2, 1]) focus on security vulnerabilities, not on the kinds of errors being detected by Thor.

7. CONCLUSION

We have presented a light-weight methodology for leveraging existing test suites by executing them in adverse conditions, systematically injecting event sequences that should not affect the outcome of the tests. Our evaluation on a small collection of Android apps demonstrates that the approach is effective in finding critical bugs, many of which are functionality errors that are difficult to detect with other automated testing techniques.

In addition, our results show that the cost in additional testing time is low relative to the number of bugs found, and that the technique is capable of isolating the causes of failures to support debugging.

8. ACKNOWLEDGMENTS

This work was supported by the Danish Research Council for Technology and Production.

9. REFERENCES

[1] Fortify Source Code Analyzer, 2015. http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/.

[2] IBM Security AppScan Source, 2015. http://www.ibm.com/software/products/en/appscan-source.

[3] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. MobiGUITAR – a tool for automated model-based testing of mobile apps. *IEEE Software*, 2015. To appear.

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

[5] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 259–269, 2014.

[6] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proc. ACM SIGPLAN Int. Conf. on Object Oriented Programming Systems Languages & Applications*, pages 641–660, 2013.

[7] W. Choi, G. C. Necula, and K. Sen. Guided GUI testing of android apps with minimal restart and approximate learning. In *Proc. ACM SIGPLAN Int. Conf. on Object Oriented Programming Systems Languages & Applications*, pages 623–640, 2013.

[8] A. M. Fard and A. Mesbah. Feedback-directed exploration of web applications to derive test models. In *Proc. IEEE 24th Int. Symp. on Software Reliability Engineering*, pages 278–287, 2013.

[9] A. M. Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proc. ACM/IEEE Int. Conf. on Automated Software Engineering*, pages 67–78, 2014.

[10] G. Fraser and A. Zeller. Generating parameterized unit tests. In *Proc. 20th Int. Symp. on Software Testing and Analysis*, pages 364–374, 2011.

[11] S. Hao, B. Liu, S. Nath, W. G. J. Halfond, and R. Govindan. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proc. 12th Annual Int. Conf. on Mobile Systems, Applications, and Services*, pages 204–217, 2014.

[12] C. Hsiao, C. Pereira, J. Yu, G. Pokam, S. Narayanasamy, P. M. Chen, Z. Kong, and J. Flinn. Race detection for event-driven mobile applications. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, page 35, 2014.

[13] C. Hu and I. Neamtiu. Automating GUI testing for Android applications. In *Proc. 6th Int. Workshop on Automation of Software Test*, pages 77–83, 2011.

[14] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, effectively detecting mobile app bugs with AppDoctor. In *Proc. 9th EuroSys Conf.*, page 18, 2014.

[15] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Proc. Int. Symp. on Software Testing and Analysis*, pages 14–25, 2000.

[16] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proc. Int. Symp. on Software Testing and Analysis*, pages 67–77, 2013.

[17] J. Kim, Y. Yoon, K. Yi, and J. Shin. ScanDal: Static analyzer for detecting privacy leaks in Android applications. In *Mobile Security Technologies*, 2012.

[18] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 216–226, 2014.

[19] Y. Li, P. K. Das, and D. L. Dowe. Two decades of web application testing – a survey of recent advances. *Inf. Syst.*, 43:20–54, 2014.

[20] C. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. 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 *Proc. 20th Annual Int. Conf. on Mobile Computing and Networking*, pages 519–530, 2014.

[21] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proc. 22nd ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, pages 643–653, 2014.

[22] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: an input generation system for Android apps. In *Proc. European Software Engineering Conf. / ACM SIGSOFT Symp. on the Foundations of Software Engineering*, pages 224–234, 2013.

[23] P. Maiya, A. Kanade, and R. Majumdar. Race detection for Android applications. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, page 34, 2014.

[24] A. Mesbah, A. van Deursen, and D. Roest. Invariant-based automatic testing of modern web applications. *IEEE Trans. Software Engineering*, 38(1):35–53, 2012.

[25] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *Proc. 12th Annual Int. Conf. on Mobile Systems, Applications, and Services*, pages 190–203, 2014.

[26] T. Xie and D. Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engineering*, 13(3):345–371, 2006.

[27] D. Yan, S. Yang, and A. Rountev. Systematic testing for resource leaks in Android applications. In *Proc. IEEE 24th Int. Symp. on Software Reliability Engineering*, pages 411–420, 2013.

[28] S. Yang, D. Yan, and A. Rountev. Testing for poor responsiveness in Android applications. In *Int. Workshop on Engineering Mobile-Enabled Systems*, volume 10, pages 1–6, 2013.

[29] R. N. Zaeem, M. R. Prasad, and S. Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *Proc. IEEE Seventh Int. Conf. on Software Testing, Verification and Validation*, pages 183–192, 2014.

[30] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. on Software Engineering*, 28(2):183–200, 2002.

[31] P. Zhang and S. G. Elbaum. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. *ACM Trans. on Software Engineering and Methodology*, 23(4):32:1–32:28, 2014.