Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor

Gang Hu Xinhao Yuan Yang Tang Junfeng Yang
Columbia University
{ganghu,xinhaoyuan,ty,junfeng}@cs.columbia.edu

Abstract

Mobile apps bring unprecedented levels of convenience, yet they are often buggy, and their bugs offset the convenience the apps bring. A key reason for buggy apps is that they must handle a vast variety of system and user actions such as being randomly killed by the OS to save resources, but app developers, facing tough competitions, lack time to thoroughly test these actions. AppDoctor is a system for efficiently and effectively testing apps against many system and user actions, and helping developers diagnose the resultant bug reports. It quickly screens for potential bugs using *approximate execution*, which runs much faster than real execution and exposes bugs but may cause false positives. From the reports, AppDoctor automatically verifies most bugs and prunes most false positives, greatly saving manual inspection effort. It uses *action slicing* to further speed up bug diagnosis. We implement AppDoctor in Android. It operates as a cloud of physical devices or emulators to scale up testing. Evaluation on 53 out of 100 most popular apps in Google Play and 11 of the most popular open-source apps shows that, AppDoctor effectively detects 72 bugs—including two bugs in the Android framework that affect all apps—with quick checking sessions, speeds up testing by 13.3 times, and vastly reduces diagnosis effort.

1. Introduction

Mobile apps are a crucial part of the widely booming mobile ecosystems. These apps absorb many innovations and greatly improve our lives. They help users check emails, search the web, social-network, process documents, edit pictures, access (sometimes classified [16]) data, etc. Given the unprecedented levels of convenience and rich functionality of apps, it is unsurprising that Google Play [24], the app store of Android, alone has over 1 M apps with tens of billions of downloads [5].

Unfortunately, as evident by the large number of negative comments in Google Play, apps are frequently buggy, and the bugs offset the convenience the apps bring. A key reason for buggy apps is that they must correctly handle a vast variety of system and user actions. For instance, an app may be switched to background and killed by a mobile OS such as Android at any moment, regardless of what state the app is in. Yet, when the user reruns the app, it must still restore its state and proceed as if no interruption ever occurred. Unlike most traditional OS which support generic swapping of processes, a mobile OS can kill apps running in background to save battery and memory, while letting them backup and restore their own states. App developers must now correctly handle all possible system actions that may pause, stop, and kill their apps—the so-called *lifecycle events* in Android—at all possible moments, a very challenging problem. On top of these system actions, users can also trigger arbitrary UI actions available on the current UI screen. Unexpected user actions also cause various problems, such as security exploits that bypass screen locks [11, 50].

Testing these actions takes much time. Testing them over many device configurations (e.g., screen sizes), OS versions, and vendor customizations takes even more time. Yet, many apps are written by indie developers or small studios with limited time and resource budget. Facing tough competitions, developers often release apps under intense time-to-market pressure. Unsurprisingly, apps are often under-tested and react oddly to unexpected actions, seriously degrading user experience [31].

We present AppDoctor, a system for efficiently and effectively testing apps against many system and user actions, and helping developers diagnose the resultant bug reports. It gains efficiency using two ideas. First, it uses *approximate execution* to greatly speed up testing and reduce diagnosis effort. Specifically, it quickly screens for potential bugs by performing actions in approximate mode—which run much faster than actions in faithful mode and can expose bugs but allow false positives (FPs). For example, in-

EuroSys 2014, April 13–16, 2014, Amsterdam, Netherlands.

http://dx.doi.org/10.1145/2592798.2592813

stead of waiting for more than two seconds to inject a long-click action on a GUI widget, AppDoctor simply invokes the widget's long-click event handler. Invoking the handler is much faster but allows FPs because the handler may not be invoked at all even if a user long-clicks on the widget—the app's GUI event dispatch logic may ignore the event or send the event to other widgets. Given a set of bug reports detected through approximate executions, AppDoctor reduces the FPs caused by approximation as follows. Based on the traces of actions in bug reports, AppDoctor automatically validates most bugs by generating testcases of low-level events such as key presses and screen touches (e.g., a real long click). These testcases can be used by developers to reproduce the bugs independently without AppDoctor. Moreover, AppDoctor automatically prunes most FPs with a new algorithm that selectively switches between approximate and faithful executions. By coupling these two modes of executions, AppDoctor solves a number of problems of prior approaches which either require much manual effort to inspect bug reports or are too slow (§3).

Approximate execution is essentially a "bloom filter" approach to bug detection: it leverages approximation for speed, and validates results with real executions. Like prior work [14, 15], it generates testcases to help developers independently reproduce bugs. Unlike prior work, it generates *event testcases* and aggressively embraces approximation.

Second, AppDoctor uses *action slicing* to speed up reproducing and diagnosing app bugs. The trace leading to a bug often contains many actions. A long testcase makes it slow to reproduce a bug and isolate the cause. Fortunately, many actions in the trace are not relevant to the bug, and can be sliced out from the trace. However, doing so either requires precise action dependencies or is slow. To solve this problem, AppDoctor again embraces approximation, and employs a novel algorithm and action dependency definition to effectively slice out many unnecessary actions with high speed.

We explicitly designed AppDoctor as a dynamic tool (i.e., it runs code) so that it can find many bugs while emitting few or no FPs. We did not design AppDoctor to catch all bugs (i.e., it has false negatives). An alternative is static analysis, but a static tool is likely to have difficulties understanding the asynchronous, implicit control flow due to GUI event dispatch. Moreover, a static tool cannot easily generate low-level event testcases for validating bugs. AppDoctor does not use symbolic execution because symbolic execution is typically neither scalable nor designed to catch bugs triggered by GUI event sequences. As a result, the bugs AppDoctor finds are often different from those found by static analysis or symbolic execution.

We implement AppDoctor in Android, the most popular mobile platform today. It operates as a cloud of mobile devices or emulators to further scale up testing, and supports many device configurations and Android OS versions. To inject actions, it leverages Android's instrumentation framework [3], avoiding modifications to the OS and simplifying deployment.

Evaluation on 53 of the top 100 apps in Google Play and 11 of the most popular open-source apps shows that AppDoctor effectively detected 72 bugs in apps with tens of millions of users, built by reputable companies such as Google and Facebook; it even found two bugs in the Android framework that affect all apps; its approximate execution speeds up testing by 13.3 times; out of the 64 reports generated from one quick checking session, it verified 43 bugs and pruned 16 FPs automatically, and left only 5 reports for developer inspection; and its action slicing technique reduced the average length of traces by 4 times, further simplifying diagnosis.

Next section gives some background. §3 describes two examples to illustrate AppDoctor's advantages over prior approaches. §4 presents an overview of AppDoctor, §5 approximate execution, §6 action slicing, and §7 implementation. §8 shows the results. §9 discusses limitations, and §10 related work. §11 concludes.

2. Background

In Android, an app organizes its logic into *activities*, each representing a single screen of UI. For instance, an email app may have an activity for user login, another for listing emails, another for reading an email, and yet another for composing an email. The number of activities varies greatly between apps, from a few to more than two hundred, depending on an app's functionality. All activities run within the main thread of the apps.

An activity contains *widgets* users interact with. Android provides a set of standard widgets [4], such as buttons, text boxes, seek bars (a slider for users to select a value from a range of values), switches (for users to select options), and number pickers (for users to select a value from a set of values by touching buttons or swiping a touch screen). Widgets handle a standard set of UI *actions*, such as clicks (press and release a widget), long-clicks (press, hold, and release a widget), typing text into text boxes, sliding seek bars, and toggling switches.

Users interact with widgets by triggering low-level *events*, including touch events (users touching the device's screen) and key events (users pressing or releasing keys). Android OS and apps work together to compose the low-level events into actions and dispatch the actions to the correct widgets. This dispatch can get quite complex because developers can customize widgets in many different ways. For instance, they can override the low-level event handlers to compose the events into non-standard actions or forward events to other widgets for handling. Moreover, they can create GUI layout with one widget covering another at runtime, so the widget on top receives the actions.

Users also interact with an activity through three special keys of Android. The Back key typically causes Android to

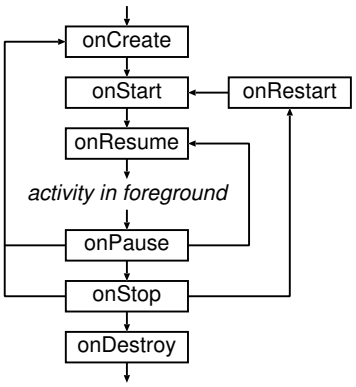

Figure 1: *Android activity lifecycles [1].* Boxes represent lifecycle event handlers. Arrows are possible status transition paths.

go back to the previous activity or undo a previous action. The Menu key typically pops up a menu widget listing actions that can be done within the current activity, The Search key typically starts a search in the current app. These keys present a standard, familiar interface for Android users.

Besides user actions, an activity handles a set of systems actions called the *lifecycle events* [1]; Figure 1 shows these events and the names of their event handlers. Android uses these events to inform an activity about status changes including when (1) the activity is created (`onCreate`), (2) it becomes visible to the user but may be partially covered by another activity (`onStart` and `onRestart`), (3) it becomes the app running in foreground and receives user actions (`onResume`), (4) it is covered by another activity but may still be visible (`onPause`), (5) it is switched to the background (`onStop`), and (6) it is destroyed (`onDestroy`).

Android dispatches lifecycle events to an activity for many purposes. For instance, an activity may want to read data from a file and fill the content to widgets when it is first created. More crucially, these events give an activity a chance to save its state before Android kills it.

User actions, lifecycle events, and their interplay can be arbitrary and complex. According to our bug results, many popular apps and even the Android framework sometimes failed to handle them correctly.

3. Examples

This section describes two examples to illustrate the advantages of AppDoctor over prior approaches. The first example is an Android GUI framework bug AppDoctor automatically found and verified and the second example is a FP AppDoctor automatically pruned.

Bug example. This bug is in Android's code for handling an app's request of a service. For instance, when an app attempts to send a text message and asks the user to choose a text message app, the app calls Android's `createChooser` method. Android then displays a dialog containing a list of apps. When there is no app for sending text messages, the dialog is empty. If at this moment, the user switches the app to the background, waits until Android saves the app's state and stops the app, and then switches the app back to the foreground, the app would crash trying to dereference `null`.

One approach to finding app bugs is to inject low-level events such as touch and key events using tools such as Monkey [47] and MonkeyRunner [40]. This approach typically has no FPs because the injected events are as real as what users may trigger. It is also simple, requiring little infrastructure to reproduce bugs, and the infrastructure is often already installed as part of Android. Thus, diagnosing bugs detected with this approach is easy.

However, this approach is quite slow because some low-level events take a long time to inject. Specifically for this bug, this approach needs time at three places. First, to detect this bug, it must wait for a sufficiently long period of time for Android to save the app state and stop the app. In our experiments, this wait is at least 5 seconds. Second, this approach does not know when the app has finished processing an event, so it has to conservatively wait for some time after each event until the app is ready for the next event. This wait is at least 6s. Third, without knowing what actions it can perform, it typically blindly injects many redundant events (e.g., clicking at points within the same button which causes the same action), while missing critical ones (e.g., stopping the app while the dialog is displayed in this bug).

AppDoctor solves all three problems. It approximates the effects of the app stop and start by directly calling the app's lifecycle event handlers, running much faster and avoiding the long wait. It detects when an action is done, and immediately performs the next action. It also understands what actions are available to avoid doing much redundant work. It detected this bug when checking the popular app Craigslist and a number of other apps. (To avoid inflating our bug count, we counted all these reports as one bug in our evaluation in §8.1.) This bug was previously unknown to us. It was was recently fixed in the Android repository. AppDoctor not only found this bug fast, but also generated an event testcase that can reliably reproduce this problem on "clean" devices that do not have AppDoctor installed, providing the same level of diagnosis help to developers as Monkey.

FP example. Another approach to test apps is to drive app executions by directly calling the app's event handlers (e.g., by calling the handler of long-click without doing a real long-click) or mutating an app's data (e.g., by setting the contents of a text box directly). A closed-source tool AppCrawler [7] appears to do so. However, this approach suffers from FPs because the actions it injects are approximate and may never occur in real executions. A significant possibility of FP means that developers must inspect the bug reports, requiring much manual effort. To illustrate why this approach has FPs, we describe an FP AppDoctor encountered and automatically pruned.

This FP is in the MyCalendar app. It has a text box for users to input their birth month. It customizes this

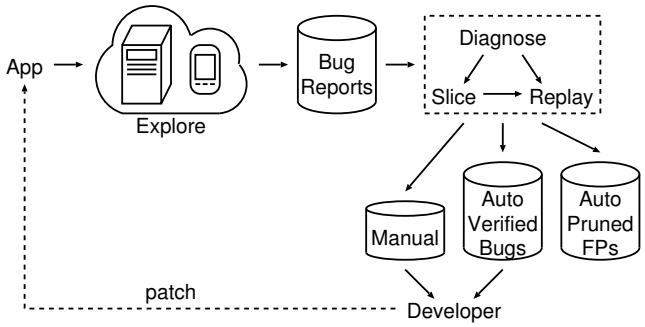

Figure 2: *AppDoctor workflow.*

text box by allowing users to select the name of a month only using a number picker it displays, ensuring that the text box's content can only be the name of one of the 12 months. When AppDoctor checked this app with approximate actions, it found an execution that led to an `IndexOutOfBoundException`. Specifically, it found that this text box was marked as editable, so it set the text to "test," a value users can never possibly set, causing the crash. Tools that directly call event handlers or set app data will suffer from this FP. Because of the significant possibility of FPs (25% in our experiments; see §8), users must inspect these tools' reports, a labor-intensive, error-prone process.

Fortunately, by coupling approximate and faithful executions, AppDoctor automatically prunes this FP. Specifically, for each bug report detected by performing actions in approximate mode, AppDoctor validates it by performing the actions again in faithful mode. For this example, AppDoctor attempted to set the text by issuing low-level touch and key events. It could not trigger the crash again because the app correctly validated the input, so it automatically classified the report as an FP.

App input validation is just one of the reasons for FPs. Another is the complex event dispatch logic in Android and apps. A widget may claim that it is visible and its event handlers are invokable, but in real execution a user may never trigger the handlers. For instance, one GUI widget W_1 may be covered by another W_2, so the OS does not invoke W_1's handlers to process user clicks. However, AppDoctor cannot rule out W_1 because visibility is only part of the story, and W_2 may actually forward the events to W_1. Precisely determining whether an event handler can be triggered by users may require manually deciphering the complex OS and app's event dispatch logic. AppDoctor's coupling of approximate and faithful executions solves all these problems with one stone.

4. Overview

This section gives an overview of AppDoctor. Figure 2 shows its workflow. Given an app, AppDoctor explores possible executions of the app on a cloud of physical devices and emulator instances by repeatedly injecting actions. This exploration can use a variety of search algorithms and heuristics to select the actions to inject (§7.4). To quickly screen for potential bugs, AppDoctor performs actions in approximate mode during exploration (§5.2). For each potential bug detected, it emits a report containing the failure caused by the bug and the trace of actions leading to the failure.

Once AppDoctor collects a set of bug reports, it runs automated diagnosis to classify reports into bugs and FPs by replaying each trace several times in approximate, faithful, and mixed mode (§5.3). It affords to replay several times because the number of bug reports is much smaller than the number of checking executions. It also applies action slicing to reduce trace lengths, further simplifying diagnosis (§6). It outputs (1) a set of auto-verified bugs accompanied with testcases that can reproduce the bugs on clean devices independent of AppDoctor, (2) a set of auto-pruned FPs so developers need not inspect them, and (3) a small number of reports marked as likely bugs or FPs with detailed traces for developer inspection.

AppDoctor focuses at bugs that may cause crashes. It targets apps that use standard widgets and support standard actions. We leave it for future work to support custom checkers (e.g., a checker that verifies the consistency of app data), widgets, and actions. AppDoctor automatically generates typical inputs for the actions it supports (e.g., the text in a text box; see §7.6), but it may not find bugs which requires a specific input. These as well as other limitations may lead to false positives (see §9).

5. Approximate Execution

This section presents AppDoctor's approximate execution technique. We start by introducing the actions AppDoctor supports (§5.1), and then discuss the explore (§5.2) and the diagnosis stages (§5.3).

5.1 Actions

AppDoctor supports 20 actions, split into three classes. The 7 actions in the first class run much faster in approximate mode than in faithful mode. The 5 actions in the second class run identically in approximate and faithful modes. The 8 actions in the last class have only approximate modes.

We start with the first class. The first 4 actions in this class are GUI events on an app's GUI widgets, and the other 3 are lifecycle events. For each action, we provide a general description, how AppDoctor performs it in approximate mode, how AppDoctor performs it in faithful mode, and the main reason for FPs.

LongClick. A user presses a GUI widget for a time longer than 2 seconds. In approximate mode, AppDoctor calls the widget's event handler by calling `widget.performLongClick`. In faithful mode, AppDoctor sends a touch event Down to the widget, waits for 3 seconds, and then sends a touch event Up. The main reason for FPs is that, depending on the event dispatch logic in Android OS and the app (§2), the touch events may not be sent to the widget, so the LongClick handler of the widget is not

invoked in a real execution. A frequent scenario is that the widget is covered by another widget, so the widget on top intercepts all events.

SetEditText. A user sets the text of a TextBox. In approximate mode, AppDoctor directly sets the text by calling the widget's method `setText`. In faithful mode, AppDoctor sends a series of low-level events to the text box to set text. Specifically, it sends a touch event to set the focus to the text box, Backspace and Delete keys to erase the old text, and other keys to type the text. The main reason for FPs is that developers can customize a text box to allow only certain text to be set. For instance, they can validate the text or override the widget's touch event handler to display a list of texts for a user to select.

SetNumberPicker. A user sets the value of a number picker. In approximate mode, AppDoctor directly sets the value by calling the widget's method `setValue`. In faithful mode, AppDoctor sends a series of touch events to press the buttons inside the number picker to gradually adjust its value. The main reason for FPs is similar to that of SetEditText where developers may allow only certain values to be set.

ListSelect. A user scrolls a list widget and selects an item in the list. In approximate mode, AppDoctor calls the widget's `setSelection` to make the item show up on the screen and select it. In faithful mode, AppDoctor sends a series of touch events to scroll the list until the given item shows up. The main reason for FPs is that developers can customize the list widget and limit the visible range of the list to a user.

PauseResume. A user switches an app to the background (e.g., by running another app) for a short period of time, and then switches back the app (see lifecycle events in §2). Android OS pauses the app when the switch happens, and resumes it after the app is switched back. In approximate mode, AppDoctor calls the foreground activity's event handlers `onPause` and `onResume` to emulate this action. In faithful mode, AppDoctor starts another app (currently Android's Settings app for configuring system-wide parameters), waits for 1s, and switches back. The main reason for FPs is that developers can alter the event handlers called to handle lifecycle events.

StopStart. This action is more involved than PauseResume. It occurs when a user switches an app to the background for a longer period of time, and then switches back. Since the time the app is in background is long, Android OS saves the app's state and destroys the app to save memory. Android later restores the app's state when the app is switched back. In approximate mode, AppDoctor calls the following event handlers of the current activity: `onPause`, `onSaveInstanceState`, `onStop`, `onRestart`, `onStart`, and `onResume`. In faithful mode, AppDoctor starts another app, waits for 10s, and switches back. The main reason for FPs is that developers can alter the event handlers called to handle lifecycle events.

Relaunch. This action is even more involved than StopStart. It occurs when a user introduces some configuration changes that cause the current activity to be destroyed and recreated. For instance, a user may rotate her device (causing the activity to be destroyed) and rotate it back (causing the activity to be recreated). In approximate mode, AppDoctor calls Android OS's `recreate` to destroy and recreate the activity. In faithful mode, AppDoctor injects low-level events to rotate the device's orientation twice. The main reason for FPs is that apps may register their custom event handlers to handle relaunch-related events, so the activities are not really destroyed and recreated.

All these 7 actions in the first class run much faster in approximate mode than in faithful mode, so AppDoctor runs them in approximate mode during exploration. AppDoctor supports a second class of 5 actions for which invoking their handlers is as fast as sending low-level events. Thus, AppDoctor injects low-level events for these actions in both approximate and faithful modes.

Click. A user quickly taps a GUI widget. In both modes, AppDoctor sends a pair of touch events, Down and Up, to the center of a widget.

KeyPress. A user presses a key on the phone, like the Back key or the Search key. AppDoctor sends a pair of key events Down and Up with the corresponding key code to the app. This action sends only special keys because standard text input is handled by SetEditText.

MoveSeekBar. A user changes the value of a seek bar widget. In both modes, AppDoctor calculates the physical position on the widget that corresponds to the value the user is setting, and send a pair of touch event Down and Up on that position to the widget.

Slide. A user slides her finger on the screen. AppDoctor first sends a touch event Down on the point where the slide starts. Then AppDoctor sends a series of touch event Move on the points along the slide. Finally, AppDoctor sends a touch event Up on the point where the slide stops. AppDoctor supports two types of slides: horizontal and vertical.

Rotate. A user changes the orientation of the device. AppDoctor injects a low-level event to rotate the device's orientation.

AppDoctor supports a third class of 8 actions caused by external events in the execution environment of an app, such as the disconnection of the wireless network. AppDoctor injects them by sending emulated low-level events to an app, instead of for example disconnecting the network for real. We discuss three example actions below.

Intent. An app may run an activity in response to a request from another app. These requests are called *intents* in Android. Currently AppDoctor injects all intents that an app declares to handle, such as viewing data, searching for media files, and getting data from a database.

```
appdoctor.explore_once() { // returns a bug trace
  trace = {};
  appdoctor.reset_init_state();
  while (app not exit and action limit not reached) {
    action_list = appdoctor.collect();
    action = appdoctor.choose(action_list);
    appdoctor.perform(action, APPROX);
    trace.append(action);
    if (failure found)
      return trace;
  }
}
```

Figure 3: *Algorithm to explore one execution for bugs.*

Network. AppDoctor injects network connectivity change events, such as the change from wireless to 3G and from connected to disconnected status.

Storage. AppDoctor injects storage related events such as the insertion or removal of an SD card.

5.2 Explore

When AppDoctor explores app executions for bugs, it runs the actions described in the previous subsection in approximate mode for speed. Figure 3 shows AppDoctor's algorithm to explore one execution of an app for bugs. It sets the initial state of the app, then repeatedly collects the actions that can be done, chooses one action, performs the action in approximate mode, and checks for bugs. If a failure such as an app crash occurs, it returns a trace of actions leading to the failure.

To explore more executions, AppDoctor runs this algorithm repeatedly. It collects available actions by traversing the GUI hierarchy of the current activity leveraging the Android instrumentation framework (§7.1). AppDoctor then chooses one of the actions to inject. By configuring how to choose actions, AppDoctor can implement different search heuristics such as depth-first search, breadth-first search, priority search, and random walk (§7.4). AppDoctor performs the action as soon as the previous action is done, further improving speed (§7.5).

5.3 Diagnosis

The bug reports detected by AppDoctor's exploration are not always true bugs because the effects of actions in approximate mode are not always reproduced by the same actions in faithful mode. Manually inspecting all bug reports would be labor-intensive and error-prone, raising challenges for time and resource-constrained app developers. Fortunately, AppDoctor automatically classifies bug reports for the developers using the algorithm shown in Figure 4, which reduced the number of reports developers need to inspect by 13.6× in our evaluation (§8.4).

This algorithm takes an action trace from a bug report, and classifies the report into four types: (1) verified bugs

```
appdoctor.diagnose(trace) { // returns type of bug report
  // step 1: tolerate environment problems
  if (not appdoctor.reproduce(trace, APPROX))
    return PRUNED_FP;
  // step 2: auto-verify bugs
  trace = appdoctor.slice(trace); // described in Section 6
  if (appdoctor.reproduce(trace, FAITHFUL)) {
    testcase = appdoctor.to_monkeyrunner(trace);
    if (MonkeyRunner reproduces the failure with testcase)
      return VERIFIED_BUG;
    else return LIKELY_BUG;
  }
  // step 3: auto-prune FPs
  for (action1 in trace) {
    appdoctor.reset_init_state();
    // replay actions in approximate mode, except action1
    for (action2 in trace) {
      if (action2 != action1)
        appdoctor.perform(action2, APPROX);
      else
        appdoctor.perform(action2, FAITHFUL);
      if (replay diverges) break;
    }
    if (failure disappears)
      return PRUNED_FP; // action1 is the culprit
  }
  return LIKELY_FP;
}
```

Figure 4: *Algorithm to diagnose one trace.*

(real bugs reproducible on clean devices), (2) pruned false positives, (3) likely bugs, and (4) likely false positives. Type 1 and 2 need no further manual inspection to classify (for verified bugs, developers still have to pinpoint the code responsible for the bugs and patch it). The more reports AppDoctor places in these two types, the more effective AppDoctor is. Type 3 and 4 need some manual inspection, and AppDoctor's detailed action trace and suggested types of the reports help reduce inspection effort.

As shown in the algorithm, AppDoctor automatically diagnoses a bug report in three steps. First, it does a quick filtering to prune false positives caused by Android emulator/OS/environment problems. Specifically, it replays the trace in approximate mode, checking whether the same failure occurs. If the failure disappears, then the report is most likely caused by problems in the environment, such as bugs in the Android emulator (which we did encounter in our experiments) or temporary problems in remote servers. AppDoctor prunes these reports as FPs.

Second, it automatically verifies bugs. Specifically, it simplifies the trace using the action slicing technique described in the next section, and replays the trace in faithful mode. If the same failure appears, then the trace almost

always corresponds to a real bug. AppDoctor generates a MonkeyRunner testcase, and verifies the bug using clean devices independent of AppDoctor. If it can reproduce the failure, it classifies the report as a verified bug. The testcase can be sent directly to developers for reproducing and diagnosing the bug. If MonkeyRunner cannot reproduce the failure, then it is most likely caused by the difference in how AppDoctor and MonkeyRunner wait for an action to finish. Thus, AppDoctor classifies the report as a likely bug, so developers can inspect the trace and modify the timing of the events in the MonkeyRunner testcase to verify the bug.

Third, AppDoctor automatically prunes FPs. At this point, the trace can be replayed in approximate mode, but not in faithful mode. If AppDoctor can pinpoint the action that causes this divergence, it can confirm that the report is an FP. Specifically, for each action in the trace (`action1` in Figure 4), AppDoctor replays all other actions in the trace in approximate mode except this action. If the failure disappears, AppDoctor finds the culprit of the divergence, and classifies the report as a pruned FP. If AppDoctor cannot find such an action, it classifies the report as a likely FP for further inspection.

6. Action Slicing

AppDoctor uses action slicing to remove unnecessary actions from a trace before determining whether the trace is a bug or FP (`slice` in Figure 4). This technique brings two benefits. First, by shortening the trace, it also shortens the final testcase (if the report is a bug), reducing developer diagnosis effort. Second, a shorter trace also speeds up replay.

Slicing techniques [30, 54] have been shown to effectively shorten an instruction trace by removing instructions irrelevant to reaching a target instruction. However, these techniques all hinge on a clear specification of the dependencies between instructions, which AppDoctor does not have for the actions in its traces. Thus, it appears that AppDoctor can only use slow approaches such as attempting to remove actions one subset by one subset to shorten the trace.

Our insight is that, because AppDoctor already provides an effective way to validate traces, it can embrace approximation in slicing as well. Specifically, given a trace, AppDoctor applies a fast slicing algorithm that computes a minimal slice assuming minimal, approximate dependencies between actions. It validates whether this slice can reproduce the failure. If so, it returns this slice immediately. Otherwise, it applies a slow algorithm to compute a more accurate slice.

Figure 5 shows the fast slicing algorithm. It takes a trace and returns a slice of the trace containing actions necessary to reproduce the failure. It starts by putting the last action of the trace into `slice` because the last action is usually necessary to cause the failure. It then iterates through the trace in reverse order, adding any action that the actions in the slice approximately depend on.

```
appdoctor.fast_slice(trace) {
  slice = {last action of trace};
  for (action in reverse(trace))
    if (action in slice)
      slice.add(get_approx_depend(action, trace));
  return slice;
}
get_approx_depend(action, trace) {
  for (action2 in trace) {
    if (action is enabled by action2)
      return action2;
    if (action is always available
      && action2.state == action.state)
      return action2;
  }
}
```

Figure 5: *Fast slicing algorithm to remove actions from trace.*

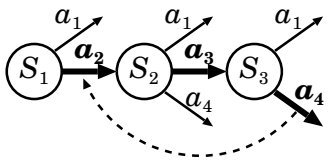

Figure 6: *Type 1 action dependency.* S_i represents app states, and a_i represents actions. Bold solid lines are the actions in the trace, thin solid lines the other actions available at a given state, and dotted lines the action dependency. a_4 depends on a_2 because a_2 enables a_4.

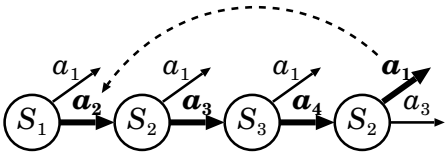

Figure 7: *Type 2 action dependency.* a_1 depends on a_2 because a_1 is performed in S_2, and a_2 is the action that first leads to S_2.

The key of this algorithm is `get_approx_depend` for computing approximate action dependencies. It leverages an approximate notion of an activity's *state*. Specifically, this state includes each widget's type, position, and content and the parent-child relationship between the widgets. It also includes the data the activity saves when it is switched to background. To obtain this data, AppDoctor calls the activity's `onPause`, `onSaveInstanceState` and `onResume` handler. This state is approximate because the activity may hold additional data in other places such as files.

Function `get_approx_depend` considers only two types of dependencies. First, if an action becomes available at some point, AppDoctor considers this action depending on the action that "enables" this action. For instance, suppose a

Click action is performed on a button and the app displays a new activity. We say that the Click enables all actions of the new activity and is depended upon by these actions. Another example is shown In Figure 6. Action a_4 becomes available after action a_2 is performed, so AppDoctor considers a_4 dependent on a_2.

Second, if an action is always available (e.g., a user can always press the Menu key regardless of which activity is in foreground) and is performed in some state S_2, then it depends on the action that first creates the state S_2 (Figure 7). For instance, suppose a user performs a sequence of actions ending with action a_2, causing the app to enter state S_2 for the first time. She then performs more actions, causing the app to return to state S_2, and performs action a_1 "press the Menu key." `get_approx_depend` considers that action a_1 depends on action a_2. The intuition is that the effect of an always available action usually depends on the current app state, and this state depends on the action that leads the app to this state.

When the slice computed by fast slicing cannot reproduce the failure, AppDoctor tries a slower slicing algorithm by removing cycles from the trace, where a cycle is a sequences of actions starts and ends at the same state. For instance, Figure 7 contains a cycle ($S_2 \rightarrow S_3 \rightarrow S_2$). If a sequence of actions do not change the app state, discarding them should not affect the reproducibility of the bug. If the slower algorithm also fails, it falls back to the slowest approach. It iterates through all actions in the trace, trying to remove them one subset by one subset.

Our results show that fast slicing works very well. It worked for 43 out of 61 traces. The slower version worked for 10 more. Only 8 needed the slowest version. Moreover, it reduced the mean trace length from 38.71 to 10.03, making diagnosis much easier.

7. Implementation

AppDoctor runs on a cluster of Android devices or emulators. Figure 8 shows the architecture. A controller monitors multiple agents and, when some agents become idle, commands these agents to start checking sessions based on developer configurations. The agents can run on the same machine as the controller or across a cluster of machines, enabling AppDoctor to scale. Each agent connects to a device or an emulator via the Android Debug Bridge [2]. The agent installs to the devices or emulators the *target app* to check and an *instrumentation app* for collecting and performing actions. It then starts and connects to the instrumentation app, which in turn starts the target app. The agent then explores possible executions of the target app by receiving the list of available actions from the instrumentation app and sending commands to the instrumentation app to perform actions on the target app.

The agent runs in a separate process outside of the emulator or the device for robustness. It tolerates many types

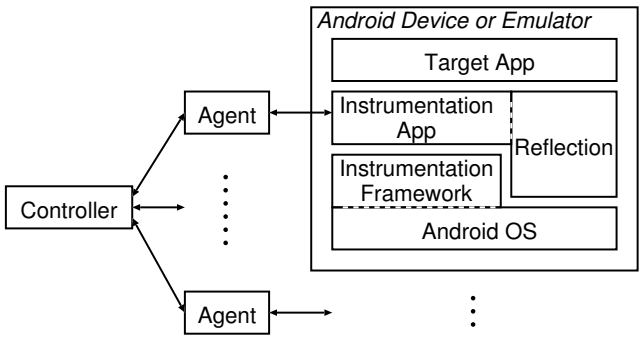

Figure 8: *AppDoctor architecture.* Dotted lines in the "Anrdoid Device or Emulator" box indicates tight coupling of components.

of failures including Android system failures and emulator crashes. Furthermore, it enables the system to keep information between checking executions, so AppDoctor can explore a different execution than previously explored (§7.4).

The controller contains 188 lines of Python code. The agent contains 3701 lines of Python code. The instrumentation app contains 7259 lines of Java code. The remainder of this section discusses AppDoctor's implementation details.

7.1 Instrumentation App

To test an app, AppDoctor needs to monitor the app's state, collect available actions from the app, and perform actions on the app. The Android instrumentation framework [3] provides interfaces for monitoring events delivered to an app and injecting events into the app. We built the instrumentation app using this framework. It runs in the same process as the target app for collecting and performing actions. It also leverages Java's reflection mechanism to collect other information from the target app that the Android instrumentation framework cannot collect. Specifically, it uses reflection to get the list of widgets of an activity and directly invoke an app's events handlers even if they are private or protected Java methods. The instrumentation app enables developers to write app-specific checkers, which we leave for future work.

7.2 App Repacking and Signing

For security purposes, Android requires that the instrumentation app and the target app be signed by the same key. To work around this restriction, AppDoctor unpacks the target app and then repacks and signs the app using its own key. Furthermore, since AppDoctor needs to communicate with the instrumentation app through socket connections, it uses ApkTool [12] to add network permission to the target app.

7.3 Optimizations

We implemented two optimizations in AppDoctor to further speedup the testing process. First, AppDoctor pre-generates a repository of cleanly booted emulator snapshots, one per configuration (e.g., screen size and density). When checking an app, AppDoctor simply starts from the specific snap-

shot instead of booting an emulator from scratch, which can take 5 minutes. Second, to check multiple executions of an app, AppDoctor reuses the same emulator instance without starting a new one. To reset the app's initial state (§5.2), it simply kills the app process and wipes its data. These two optimizations minimize the preparation overhead and ensure that AppDoctor spends most of the time checking apps.

7.4 Exploration Methods

Recall that when AppDoctor explores possible executions of an app, it can choose the next action to explore using different methods (Figure 3). It currently supports four methods: interactive, scripted, random, and systematic. With the interactive method, AppDoctor shows the list of available actions to the developer and lets her decide which one to perform, so she has total control of the exploration process. This method is most suitable for diagnosing bugs. With the scripted method, developers write scripts to select actions, and AppDoctor runs these test scripts. This method is most suitable for regression and functional testing. With the random method, AppDoctor randomly selects an action to perform. This method is most suitable for automatic testing. With the systematic method, AppDoctor systematically enumerates through the actions for bugs using several search heuristics, including breadth-first search, depth-first search, and developers written heuristics. This method is most suitable for model checking [21, 26, 32, 41, 44, 51, 52].

7.5 Waiting for Actions to Finish

Recall that AppDoctor performs actions on the target app as soon as the previous action is done. It detects when the app is done with the action using the Android instrumentation framework's `waitForIdle` function, which returns when the main thread—the thread for processing all GUI events—is idle. Two apps, Twitter and ESPN, sometimes keep the main thread busy (e.g., during the login activity of Twitter), so AppDoctor falls back to waiting for a certain length of time (3 seconds). Apps may also run asynchronous tasks in background using Android's `AsyncTask` Java class, so even if their main threads are idle, the overall event processing may still be running. AppDoctor solves this problem by intercepting asynchronous tasks and waiting for them to finish. Specifically, AppDoctor uses reflection to replace `AsyncTask` with its own to monitor all background tasks and wait for them to finish.

7.6 Input Generation

Apps often require inputs to move from one activity to another. For instance, an app may ask for an email address or user name. AppDoctor has a component to generate proper inputs to improve coverage. It focuses on text boxes because they are the most common ways for apps to get texts from users. Android has a nice feature that simplifies AppDoctor's input generation. Specifically, Android allows developers to specify the type of a text box (e.g., email addresses and integers), so that when a user starts typing, Android can display the keyboard customized for the type of text. Leveraging this feature, AppDoctor automatically fills many text boxes with texts from a database we pre-generated, which includes email addresses, numbers, etc. To further help developers test apps, AppDoctor allows developers to specify input generation rules in the form of "`widget-name:pattern-of-text-to-fill`." In our experiments, the most common use of this mechanism is to specify login credentials. Other than text boxes, developers may also specify rules to generate inputs for other actions, including the value set by `SetNumberPicker`, the item selected by `ListSelect` and the position set by `MoveSeekBar`. By default, AppDoctor generates random inputs for these three actions. Note that AppDoctor can leverage symbolic execution [14, 22] to generate inputs that exercise tricky code paths within apps, which we intend to explore in future work. However, our current mechanisms suffice to detect many bugs because, based on our experience, apps treat many input texts as "black boxes," and simply store and display the texts without actually using them in any fancy way.

7.7 Replay and Nondeterminism

Recall that, at various stages, AppDoctor replays a trace to verify if the trace can reproduce the corresponding failure. This replay is subject to the nondeterminism in the target app and environment. A plethora of work has been done in deterministic record-replay [20, 27, 34, 43]. Although AppDoctor can readily leverage any of these techniques, we currently have not ported them to AppDoctor. For simplicity, we implemented a best-effort replay technique and replayed every trace 20 times.

7.8 Removing Redundant Reports

One bug may manifest multiple times during exploration, causing many redundant bug reports. After collecting reports from all servers, AppDoctor filters redundant reports based mainly on the type of the failure and the stack trace and keeps five reports per bug.

7.9 Extracting App Information

AppDoctor uses ApkTool [12] to unpack the target app for analysis. It processes `AndroidManifest.xml` to find necessary information, including target app's identifier, startup activity, and library dependencies. It then uses this information to start the target app on configurations with the required libraries. AppDoctor analyzes resource files to get the symbolic names corresponding to each widget, enabling developers to refer to widgets by symbolic names in their testing scripts (§7.4) and input generation rules (§7.6).

8. Evaluation

We evaluated AppDoctor on a total of 64 popular apps from Google Play, including 53 closed-source apps and 11

open-source ones, listed in §8.1. We selected the closed-source apps as follows. We started from the top 100 popular apps in Nov 2012, then excluded 31 games that use custom GUI widgets written from scratch which AppDoctor does not currently handle, 3 apps that require bank accounts or paid memberships, 2 libraries that do not run alone, 8 apps with miscellaneous dependencies such as requiring text message authentication, and 3 apps that do not work with the Android instrumentation framework. We selected 11 open-source apps also based on popularity. Their source code simplifies inspecting the cause of the bugs detected. We picked the most popular apps because they are well tested, presenting a solid benchmark suite of AppDoctor's bug detection capability.

We ran several quick checking sessions on these apps. Each session ran for roughly one day and had 165,000 executions, 2,500 per app. These executions were run on a cluster of 14 Intel Xeon servers. Each execution ran until 100 actions were reached or the app exited. Each session detected a slightly different set of bugs because checking executions used heuristics. Except §8.1 which reports cumulative bug results over all sessions, all other subsections report results from the latest session.

The rest of this section focuses on four questions:
§8.1: Can AppDoctor effectively detect bugs?
§8.2: Can AppDoctor achieve reasonable coverage?
§8.3: Can AppDoctor greatly speed up testing?
§8.4: Can AppDoctor reduce diagnosis effort?

8.1 Bugs Detected

AppDoctor found a total of 72 bugs in 64 apps. Of these bugs, 67 are new and the other 5 bugs were unknown to us but known to the developers. We have reported 9 new bugs in the open-source apps to the developers because these bugs are easier to diagnose with source code and the apps have public websites for reporting bugs. The developers have fixed 2 bugs and confirmed two more. Of the 5 bugs unknown to us but known to the developers, 4 were fixed in the latest version, and the other 1 was reported by users but without event traces to reproduce the bugs.

Table 1 shows the bug count for each app we checked. We also show the number of users obtained from Google Play to show the popularity of the apps. The results show that AppDoctor can find bugs even in apps that have tens of millions of users and built by reputable companies such as Google and Facebook. AppDoctor even found two bugs in the Android framework that affect all apps. AppDoctor found no bugs in 24 apps: WordSearch, Flixster, Adobe Reader, BrightestFlaghlight, Ebay, Skype, Pinterest, Spotify, OI Shopping List, Daily Money, Dropbox, Midomi, Groupon, Speedtest, ColorNote, Voxer, RedBox, Lookout, Facebook Messenger, Devuni Flashlight, Go SMS, Wikipedia, Ultimate StopWatch and Wells Fargo.

We inspected all of the bugs found in the open-source apps to pinpoint their causes in the source code. Table 2 shows the details of these bugs. Most of the bugs are caused by accessing `null` references. The common reasons are that the developers forget to initialize references, access references that have been cleaned up, miss checks of `null` references, and fail to check certain assumptions about the environments. Most of these bugs can be triggered only under rare event sequences.

We describe two interesting bugs. The first is Bug 11 in Table 2, a bug in the Android framework. AppDoctor was

App	Bugs	Users (M)	Open?	Hints
Android	2	n/a		
Google Maps	3	$500 \sim 1000$		
Facebook	2	$500 \sim 1000$		L, D
Pandora	1	$100 \sim 500$		L
Twitter	1	$100 \sim 500$		L
Google Translate	3	$100 \sim 500$		
Shazam	3	$100 \sim 500$		
Sgiggle	2	$100 \sim 500$		
Advanced Task Killer	1	$50 \sim 100$		
Barcode Scanner	1	$50 \sim 100$		
Zedge	1	$50 \sim 100$		
Amazon Kindle	1	$50 \sim 100$		L
Yahoo Mail	3	$50 \sim 100$		L
TuneIn Player	1	$50 \sim 100$		
Walk Band	2	$50 \sim 100$		D
PhotoGrid	1	$50 \sim 100$		
Kik Messenger	3	$50 \sim 100$		L
Logo Quiz	2	$10 \sim 50$		
Zynga Words	2	$10 \sim 50$		D
Amazon	2	$10 \sim 50$		L
Mobile Bible	3	$10 \sim 50$		
MyCalendar	1	$10 \sim 50$		L
Dictionary	1	$10 \sim 50$		
GasBuddy	1	$10 \sim 50$		
ooVoo	1	$10 \sim 50$		L
iHeartRadio	1	$10 \sim 50$		
IMDB Mobile	1	$10 \sim 50$		
ESPN Sports	2	$10 \sim 50$		
Craigslist	2	$10 \sim 50$		
TextGram	1	$10 \sim 50$		
Google MyTracks	2	$10 \sim 50$	Yes	
Terminal Emulator	1	$10 \sim 50$	Yes	
Fandango	2	$10 \sim 50$		L, D
DoubleDown	1	$5 \sim 10$		
OI FileManager	2	$5 \sim 10$	Yes	
MP3 Ringtone Maker	2	$1 \sim 5$		
BlackFriday	6	$1 \sim 5$		
ACV Comic Viewer	2	$1 \sim 5$	Yes	
OpenSudoku	1	$1 \sim 5$	Yes	
OI Notepad	1	$0.1 \sim 0.5$	Yes	
OI Safe	1	$0.1 \sim 0.5$	Yes	

Table 1: *Each app's bug count.* First row lists the bugs AppDoctor found in the Android framework, which affect almost all apps. Number of users is in millions. The "Open?" column indicates whether the app is open source. "Hints" lists additional information we added: "L" for login credentials (§7.6) and "D" for delays (§7.7).

	App	Bug Description	Status
1	Google MyTracks	Pressing 'Search' button bypassed License dialog and environment check, causing a crash	Fixed
2	OI File Manager	Checked for NullPointerException in doInBackground() but missed in onPostExecute()	Fixed
3	Terminal Emulator	Rare event sequence led to access of discarded variable	Confirmed
4	OI File Manager	Rare event order led to use of uninitialized variable	Confirmed
5	ACV Comic Viewer	Incorrect assumption of the presence of Google Services caused a crash	Reported
6	ACV Comic Viewer	Failed to check for the failure of opening a file due to lack of permission, causing a crash	Reported
7	OI Notepad	Failed to check for the availability of another software after rotation, while checked before rotation	Reported
8	OpenSudoku	Failed to check for the failure of loading a game, which was caused by the deletion of the game	Reported
9	OI Safe	Rare event sequence led to access of discarded variable	Reported
10	Google MyTracks	Dismissing a dialog which had been removed from the screen due to lifecycle events	Known
11	Android	Rare event order led to a failed check in Android code	Known

Table 2: *All bugs found in open-source app.* We list one Android framework bug (Bug 11) because AppDoctor found this bug when testing OpenSudoku and Wikipedia. The bug was reported by others, but the report contained no event traces causing the bug.

able to generate a testcase that reliably reproduces the bug on OpenSudoku, Wikipedia, Yahoo Mail, Shazam, Facebook, and Pinterest. We counted this bug as one bug to avoid inflating our bug count. To trigger this bug in OpenSudoku, a user selects a difficulty level of the game, and presses the Back key of the phone quickly, which sometimes crashes the app. The cause is that when an app switches from one activity to another, many event handlers are called. In the common case, these event handlers are called one by one in order, which tends to be what developers test. However, in rare cases, another event handler, such as the handler of the Back key in OpenSudoku, may jump into the middle of this sequence while the app is in an intermediate state. If this handler refers to some part of the app state, the state may be inconsistent or already destroyed, causing a crash. This bug was reported to the Android bug site, but no event sequences were provided on how the bug might be triggered, and the bug is still open. We recently reported the event sequence to trigger this bug, and are waiting for developers to reply.

The second is a bug in Google Translate, the most popular language translation app in Android, closed source, and built by Google. This bug causes Google Translate to crash after a user presses the Search key on the phone at the wrong moment. When a user first installs and runs Google Translate, it pops up a license agreement dialog with an Accept and a Decline button. If she presses the Accept button, she enters the main screen of Google Transate. If she presses the Decline button, Google Translate exits and she cannot use the app. However, on Android 2.3, if the user presses the Search button, the dialog is dismissed, but Google Translate is left in a corrupted state, and almost always crashes after a few events regardless of what the user does. We inspected the crash logs and found that the crashes were caused by accessing uninitialized references, indicating a logic bug inside Google Translate. This bug is specific to Android 2.3, and does not occur in Android 4.0 and 4.2.

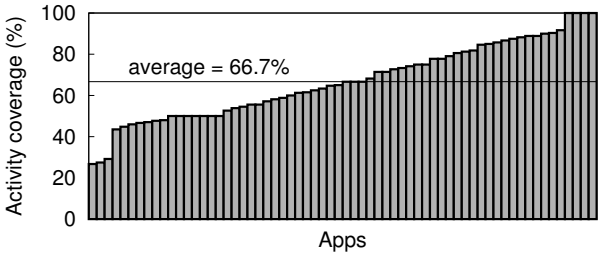

Figure 9: *Activity coverage on apps.* Each bar represents an app.

AppDoctor found a similar bug in Google MyTracks (Bug 1 in Table 2). Unlike the bug in Google Translate, this bug can be triggered in Android 2.3, 4.0, and 4.2. We reported it and, based on our report, developers have fixed it and released a new version of the app.

8.2 Coverage

We measured AppDoctor's coverage from the latest 1-day checking session. We used two metrics. First, we measured AppDoctor's coverage of activities by recording the activities AppDoctor visited, and comparing them with all the activities in the apps. We chose this metric because once AppDoctor reaches an activity, it can explore most actions of the activity. Figure 9 shows the results. With only 2,500 executions per app, AppDoctor covered 66% of the activities averaged over all apps tested, and 100% for four apps.

To understand what caused AppDoctor to miss the other activities, we randomly sampled 12 apps and inspected their disassembled code. The four main reasons are: (1) dead code (Lookout, Facebook, Flixster, OI Shopping List); (2) hardware requirement (e.g., PandoraLink for linking mobile devices with cars) not present (Sgiggle, Pandora, Brightest Flashlight); (3) activities available only to developers or pre-

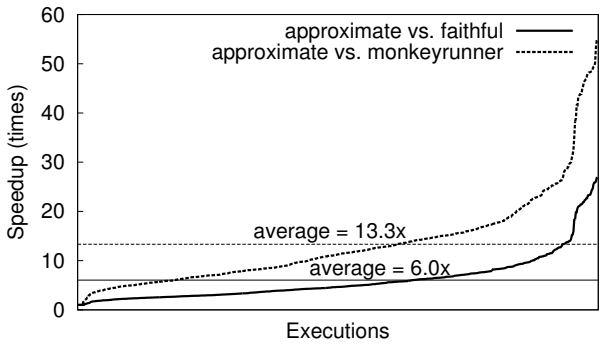

Figure 10: *Speedup of approximate execution.* The x-axis represents all executions in one checking session. Solid line shows speedup between approximate and faithful modes, and dotted line between approximate mode and MonkeyRunner.

mium accounts (Lookout, Groupon, Flixster, Facebook Messenger, Photogrid, Fandango), and (4) activities available after a nondeterministic delay (Groupon, Midomi, Pandora).

Second, we also evaluated AppDoctor's false negatives by running it on 6 bugs in five open-source apps, including two bugs in KeePassDroid (a password management app) and one bug each in OI Shopping List, OI Notepad, OI Safe (another password management app), and OI File Manager. We picked these bugs because they are event-triggered bugs, which AppDoctor is designed to catch. AppDoctor found 5 out of the 6 bugs. It missed one bug in KeePassDroid because it treated 2 different states as the same and pruned the executions that trigger the bug.

8.3 Speedup

AppDoctor's speedup comes from (1) actions run faster in approximate mode and (2) it performs the next action as soon as the previous one finishes. Figure 10 shows the speedup caused by the two factors. For each of the 165,000 executions from the latest AppDoctor checking session, we measured the time it took to complete this execution in (a) approximate mode, (a) faithful mode, and (c) MonkeyRunner. The difference between (a) and (b) demonstrates the speedup from approximate execution. The difference between (a) and (c) demonstrates the speedup from both approximate execution and AppDoctor's more efficient wait method. As shown in Figure 10, approximate execution yields 6.0× speedup, and the efficient wait brings the speedup to 13.3×. Since most executions do not trigger bugs, AppDoctor spends majority of time in approximate mode, so this result translates at least 13.3× speedup over MonkeyRunner, "at least" because MonkeyRunner blindly injects events whereas AppDoctor does so systematically.

8.4 Automatic Diagnosis

We evaluated how AppDoctor helps diagnosis using the reports from the last checking session. We focus on: (1) how many reports AppDoctor can automatically verify; (2) for

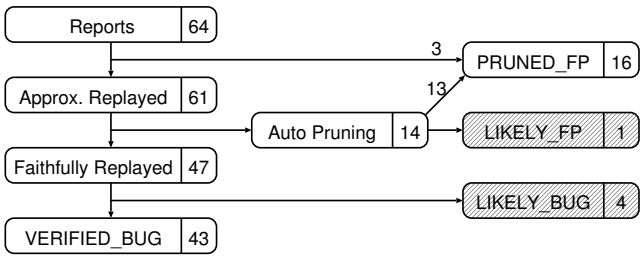

Figure 11: *Auto-diagnosis results.* Number of bug reports are shown at each step. White boxes are automatically classified. Shaded boxes need manual inspection.

the reports AppDoctor cannot verify, whether they are FPs or bugs; (3) what causes FPs; and (4) how effective action slicing is at pruning irrelevant events.

Figure 11 shows AppDoctor's automatic diagnosis results based on the latest checking session, which reduced the number of bug reports to inspect from 64 to only 5, 12.8× reduction. AppDoctor initially emitted 64 bug reports on all 64 apps in the latest session. Of these reports, it could replay 61 in approximate mode, and discarded the other 3 reports as false positives. It then simplified the 61 reports and managed to replay 47 in faithful mode. Based on the 47 faithfully replayed reports, it generated MonkeyRunner testcases and automatically reproduced 43 bugs on clean devices, verifying that these 43 reports are real bugs. 4 MonkeyRunner testcases did not reproduce the bugs, so AppDoctor flagged them as needing manual inspection. Out of the 14 reports that could be replayed in approximate mode but not in faithful mode, AppDoctor automatically pruned 13 false positives. The remaining one it could not prune was due to a limitation in our current implementation of the faithful mode of ListSelect (§5.1). Selecting an item by injecting low-level events involves two steps: scrolling the list to make the item show up, and moving the focus to the selected item. We implemented the first step by injecting mouse events to scroll the list, but not the second step because it requires external keyboard or trackball support, which we have not added. AppDoctor flagged this report as needing manual inspection. Thus, out of 64 reports, AppDoctor automatically classified 59, leaving only 5 for manual inspection.

The 5 reports that need manual inspection contain 4 MonkeyRunner testcases and 1 report caused by ListSelect. We manually inspected the MonkeyRunner testcases and modified one line each to change the timing of an event. The modified testcases verified the bugs on real phones. For the one report caused by ListSelect, we manually reproduced it on real phones. Thus, all of these 5 reports are real bugs.

The total number of bugs AppDoctor found in this session is (43 + 5) = 48, lower than 72, the total number of bugs over all sessions, because each session may find slightly different set of bugs due to our search heuristics.

AppDoctor automatically pruned 13 FPs in this session, demonstrating the benefit of faithful replay. 6 are caused

by approximate long click, 5 approximate configuration change, and 2 approximate text input.

9. Limitations

Currently, AppDoctor supports system actions such as Stop-Start and Relaunch, and common GUI actions such as Click and LongClick. Adding new standard actions is easy. AppDoctor does not support custom widgets developed from scratch because these widgets receive generic events such as Click at (x, y) and then use complex internal logic to determine the corresponding action. AppDoctor also does not support custom action handlers on custom widgets created from standard widgets. Its input generation is incomplete. Symbolic execution [29] will help solve this problem. If an app talks to a remote server, AppDoctor does not control the server. AppDoctor's replay is not fully deterministic, which may cause AppDoctor to consider a real bug as a false positive and prune it out, but this problem can be solved by previous work [20, 23, 49]. AppDoctor leverages Android instrumentation which instruments only Jave-like bytecode, so AppDoctor has no control over the native part of the apps. These limitations may cause AppDoctor to miss bugs.

10. Related Work

To our knowledge, no prior systems combined approximate and faithful executions, systematically tested against lifecycle events, identified the problem of FPs caused by approximate executions, generated event testcases, or provided solutions to automatically classify most reports into bugs and FPs and to slice unnecessary actions from bug traces.

Unlike static tools, AppDoctor executes the app code and generates inputs. As a result, AppDoctor can provide an automated script to reproduce the bug on real devices, which static tools cannot do. Moreover, AppDoctor automatically verifies the bugs it finds, so all the verified bugs are not false positives and do not need manual inspection, unlike reports from static tools. Android apps are event-driven, and their control flow is hidden behind complex callbacks and inter process calls. Static tools often have a hard time analyzing event-driven programs, generate exceedingly many FPs that bury real bugs.

Fuzz testing [28, 47] feeds random inputs to programs. Without knowledge of the programs, this approach has difficulties getting deep into the program logic. Model-based testing has been applied to mobile systems [9, 35, 37, 45]. They automatically inject GUI actions based on a model of the GUI. To extract this model, various techniques are used. Robotium [6] uses reflection to collect widgets on the GUI. Dynamic crawlers [9, 35, 37] collects available events from GUI widgets, an approach AppDoctor also takes. Some use static analysis to infer possible user actions for each widget [53]. Regardless of how they compute models, they do actions either in faithful mode (i.e., inject low-level events) or in approximate mode (i.e., directly calling handlers), but not both. As illustrated in §3, they suffer from either low speed or high manual inspection effort. Moreover, none of them systematically tests for life cycle events. Interestingly, despite the significant FP rate (25% in our experiments), no prior systems that inject actions in approximate mode noted this problem, likely due to poor checking coverage. For example, DynoDroid [37] caught only 15 bugs in 1050 apps.

Several systems [10, 19, 38] leverage symbolic execution to check apps or GUI event handlers. The common approach is to mark the input to event handlers as symbolic, and explore possible paths within the handlers. These systems tend to be heavyweight and are subject to the undecidable problem of constraint solving. The event-driven nature of apps also raises challenges for these systems, as tracing the control flow through many event handlers may require analyzing complex logic in both the GUI framework and the apps. Thus, these systems often use approximate methods to generate event sequences, which may not be feasible, causing FPs. Authors of a workshop paper [39] describe test drivers that call event handlers, including lifecycle event handlers, to drive symbolic execution. However, they call lifecycle event handlers only to set up an app to receive user actions. They do not systematically test how the app reacts to these events. Nor did they present an implemented system. Symbolic execution is orthogonal to AppDoctor: it can help AppDoctor handle custom widgets and actions (§9), and AppDoctor can help it avoid FPs by generating only feasible event sequences.

Mobile devices are prone to security and privacy issues. TaintDroid [18], PiOS [17] and CleanOS [46] leverage taint tracking to detect privacy leakages. Malware detectors, RiskRanker [25] and Crowdroid [13], use both static and dynamic analysis to identify malicious code. Mobile devices are prone to abnormal battery drain caused by apps or configurations. Prior work [36, 42] detects or diagnoses abnormal battery problems.

Action slicing shares the high-level concept with program slicing [8, 30, 33, 48], which removes unnecessary instructions from programs, paths, or traces. Different from program slicing, action slicing prunes actions, rather than instructions. It embraces approximation to aggressively slice out actions and replay to validate the slicing results.

11. Conclusion

We presented AppDoctor, a system for efficiently and effectively testing Android apps and helping developers diagnose bug reports. AppDoctor uses approximate execution to speed up testing and automatically classify most reports into bugs or false positives. It uses action slicing to remove unnecessary actions from bug traces, further reducing diagnosis effort. AppDoctor works on Android, and operates as a device or emulator cloud. Results show that AppDoctor effectively detects 72 bugs in 64 of the most popular apps, speeds up testing by 13.3 times, and vastly reduces diagnosis effort.

Acknowledgments

We thank Charlie Hu (our shepherd), Xu Zhao, and the anonymous reviewers for their many helpful comments. This work was supported in part by AFRL FA8650-11-C-7190, FA8650-10-C-7024, and FA8750-10-2-0253; ONR N00014-12-1-0166; NSF CCF-1162021, CNS-1117805, CNS-1054906, and CNS-0905246; NSF CAREER; AFOSR YIP; Sloan Research Fellowship; and Google.

References

[1] Activity Class in Android Developers Site. http://developer.android.com/reference/android/app/Activity.html.

[2] Android Debug Bridge in Android Developers Site. http://developer.android.com/tools/help/adb.html.

[3] Android instrumentation framework. http://developer.android.com/reference/android/app/Instrumentation.html.

[4] Building Blocks in Android Developers Site. http://developer.android.com/design/building-blocks/.

[5] Google Play Hits 1 Million Apps. http://mashable.com/2013/07/24/google-play-1-million.

[6] Robotium framework for test automation. http://www.robotium.org.

[7] Testdroid: Automated Testing Tool for Android. http://testdroid.com.

[8] AGRAWAL, H., AND HORGAN, J. R. Dynamic program slicing. In *ACM SIGPLAN Notices* (1990), vol. 25, ACM, pp. 246–256.

[9] AMALFITANO, D., FASOLINO, A. R., TRAMONTANA, P., DE CARMINE, S., AND MEMON, A. M. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (2012), pp. 258–261.

[10] ANAND, S., NAIK, M., HARROLD, M. J., AND YANG, H. Automated concolic testing of smartphone apps. In *Proceedings of the 20th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '12/FSE-20)*.

[11] Lock Screen Security Hole Found On Some Android-Powered Samsung Galaxy Phones. http://techcrunch.com/2013/03/20/tell-me-if-youve-heard-this-one-before-lock-screen-security-flaw-found-on-samsungs-android-phones.

[12] android-apktool. http://code.google.com/p/android-apktool/.

[13] BURGUERA, I., ZURUTUZA, U., AND NADJM-TEHRANI, S. Crowdroid: behavior-based malware detection system for Android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices* (2011), pp. 15–26.

[14] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)* (Dec. 2008), pp. 209–224.

[15] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS '06)* (Oct.–Nov. 2006), pp. 322–335.

[16] U.S. government, military to get secure Android phones. http://www.cnn.com/2012/02/03/tech/mobile/government-android-phones.

[17] EGELE, M., KRUEGEL, C., KIRDA, E., AND VIGNA, G. PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '11)* (2011).

[18] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)* (2010), pp. 1–6.

[19] GANOV, S., KILLMAR, C., KHURSHID, S., AND PERRY, D. E. Event listener analysis and symbolic execution for testing GUI applications. In *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering* (2009), ICFEM '09.

[20] GEORGES, A., CHRISTIAENS, M., RONSSE, M., AND DE BOSSCHERE, K. JaRec: a portable record/replay environment for multi-threaded Java applications. *Softw. Pract. Exper. 34*, 6 (2004), 523–547.

[21] GODEFROID, P. Model checking for programming languages using verisoft. In *Proceedings of the 24th Annual Symposium on Principles of Programming Languages (POPL '97)* (Jan. 1997), pp. 174–186.

[22] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)* (June 2005), pp. 213–223.

[23] GOMEZ, L., NEAMTIU, I., AZIM, T., AND MILLSTEIN, T. RERAN: timing- and touch-sensitive record and replay for Android. In *Proceedings of the 35rd International Conference on Software Engineering (ICSE '13)* (2013), pp. 72–81.

[24] Google Play. https://play.google.com/store.

[25] GRACE, M., ZHOU, Y., ZHANG, Q., ZOU, S., AND JIANG, X. RiskRanker: scalable and accurate zero-day Android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (2012), pp. 281–294.

[26] GUO, H., WU, M., ZHOU, L., HU, G., YANG, J., AND ZHANG, L. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)* (Oct. 2011), pp. 265–278.

[27] GUO, Z., WANG, X., TANG, J., LIU, X., XU, Z., WU, M., KAASHOEK, M. F., AND ZHANG, Z. R2: An application-

level kernel for record and replay. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)* (Dec. 2008), pp. 193–208.

[28] HU, C., AND NEAMTIU, I. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test* (2011), pp. 77–83.

[29] JEON, J., MICINSKI, K. K., AND FOSTER, J. S. SymDroid: Symbolic execution for Dalvik bytecode.

[30] JHALA, R., AND MAJUMDAR, R. Path slicing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)* (June 2005), pp. 38–47.

[31] KHALID, H. On identifying user complaints of iOS apps. In *Proceedings of the 35rd International Conference on Software Engineering (ICSE '13)* (2013), pp. 1474–1476.

[32] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI '07)* (Apr. 2007), pp. 243–256.

[33] KOREL, B., AND LASKI, J. Dynamic program slicing. *Inf. Process. Lett. 29*, 3 (1988), 155–163.

[34] LAADAN, O., VIENNOT, N., AND NIEH, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '10)* (June 2010), pp. 155–166.

[35] LEE, K., FLINN, J., GIULI, T., NOBLE, B., AND PEPLIN, C. AMC: Verifying User Interface Properties for Vehicular Applications. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services* (2013), MobiSys '13.

[36] MA, X., HUANG, P., JIN, X., WANG, P., PARK, S., SHEN, D., ZHOU, Y., SAUL, L. K., AND VOELKER, G. M. eDoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones. In *NSDI'13* (2013).

[37] MACHIRY, A., TAHILIANI, R., AND NAIK, M. Dynodroid: an input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013), ESEC/FSE 2013, pp. 224–234.

[38] MIRZAEI, N., MALEK, S., PĂSĂREANU, C. S., ESFAHANI, N., AND MAHMOOD, R. Testing Android apps through symbolic execution. In *The Java Pathfinder Workshop 2012* (2012), JPF 2012.

[39] MIRZAEI, N., MALEK, S., PĂSĂREANU, C. S., ESFAHANI, N., AND MAHMOOD, R. Testing Android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes 37*, 6 (Nov. 2012).

[40] monkeyrunner. `http://developer.android.com/tools/help/monkeyrunner_concepts.html`.

[41] MUSUVATHI, M., PARK, D. Y., CHOU, A., ENGLER, D. R., AND DILL, D. L. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)* (Dec. 2002), pp. 75–88.

[42] PATHAK, A., JINDAL, A., HU, Y. C., AND MIDKIFF, S. P. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (2012), pp. 267–280.

[43] RONSSE, M., AND DE BOSSCHERE, K. RecPlay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst. 17*, 2 (1999), 133–152.

[44] SIMSA, J., GIBSON, G., AND BRYANT, R. dBug: Systematic Testing of Unmodified Distributed and Multi-Threaded Systems. In *The 18th International SPIN Workshop on Model Checking of Software (SPIN'11)* (2011), pp. 188–193.

[45] TAKALA, T., KATARA, M., AND HARTY, J. Experiences of system-level model-based GUI testing of an Android application. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation* (2011), ICST '11.

[46] TANG, Y., AMES, P., BHAMIDIPATI, S., BIJLANI, A., GEAMBASU, R., AND SARDA, N. CleanOS: limiting mobile data exposure with idle eviction. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)* (2012), pp. 77–91.

[47] UI/Application Exerciser Monkey. `http://developer.android.com/tools/help/monkey.html`.

[48] WEISER, M. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)* (Mar. 1981), pp. 439–449.

[49] WU, M., LONG, F., WANG, X., XU, Z., LIN, H., LIU, X., GUO, Z., GUO, H., ZHOU, L., AND ZHANG, Z. Language-based replay via data flow cut. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '10/FSE-18)*, pp. 197–206.

[50] YANG, J., CUI, A., STOLFO, S., AND SETHUMADHAVAN, S. Concurrency attacks. In *the Fourth USENIX Workshop on Hot Topics in Parallelism (HOTPAR '12)* (June 2012).

[51] YANG, J., SAR, C., AND ENGLER, D. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)* (Nov. 2006), pp. 131–146.

[52] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using model checking to find serious file system errors. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)* (Dec. 2004), pp. 273–288.

[53] YANG, W., PRASAD, M. R., AND XIE, T. A grey-box approach for automated GUI-model generation of mobile applications. In *Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering* (2013), FASE'13.

[54] ZHANG, X., AND GUPTA, R. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)* (2004), pp. 94–106.