Automatic and Scalable Fault Detection for Mobile Applications

Lenin Ravindranath, *M.I.T. & Microsoft Research*
Suman Nath, *Microsoft Research*
Jitendra Padhye, *Microsoft Research*
Hari Balakrishnan, *M.I.T.*

ABSTRACT

This paper describes the design, implementation, and evaluation of VanarSena, an automated fault finder for mobile applications ("apps"). The techniques in VanarSena are driven by a study of 25 million real-world crash reports of Windows Phone apps reported in 2012. Our analysis indicates that a modest number of root causes are responsible for many observed failures, but that they occur in a wide range of places in an app, requiring a wide coverage of possible execution paths. VanarSena adopts a "greybox" testing method, instrumenting the app binary to achieve both coverage and speed. VanarSena runs on cloud servers: the developer uploads the app binary; VanarSena then runs several app "monkeys" in parallel to emulate user, network, and sensor data behavior, returning a detailed report of crashes and failures. We have tested VanarSena with 3000 apps from the Windows Phone store, finding that 1138 of them had failures; VanarSena uncovered 2969 distinct bugs in existing apps, including 1227 that were not previously reported. Because we anticipate VanarSena being used in regular regression tests, testing speed is important. VanarSena uses two techniques to improve speed. First, it uses a "hit testing" method to quickly emulate an app by identifying which user interface controls map to the same execution handlers in the code. Second, it generates a `ProcessingCompleted` event to accurately determine when to start the next interaction. These features are key benefits of VanarSena's greybox philosophy.

1. INTRODUCTION

No one doubts the importance of tools to improve software reliability. For mobile apps, improving reliability is less about making sure that "mission critical" software is bug-free, but more about survival in a brutally competitive marketplace. Because the success of an app hinges on good user reviews, even a handful of poor reviews can doom an app to obscurity. A scan of reviews on mobile app stores shows that an app that crashes is likely to garner poor reviews.

Mobile app testing poses different challenges than traditional "enterprise" software. Mobile apps are often used in more uncontrolled conditions, in a variety of different locations, over different wireless networks, with a wide range of input data from user interactions and sensors, and on a variety of hardware platforms. Coping with these issues is particularly acute for individual developers or small teams.

There are various approaches for mobile app testing. Static analysis of app binaries [30, 16], although scalable, can fail to uncover app faults due to the runtime issues such as poor network condition and corrupted or unexpected responses from cloud services. Symbolic execution [10] and its hybrid variant, concolic execution, require constructing symbolic model for program execution environment. Although such a model is shown feasible for simple android libraries [33] and UI events [7, 17] of simple apps, applicability of the techniques have been limited for two key reasons. First, it is not easy to model real-world execution environment for mobile apps, consisting of sensors, networks, and cloud services. Second, they do not scale well to real-world apps due to notorious path explosion problem. Recent efforts, therefore, focus on dynamic analysis, where runtime behavior of an app is examined by executing it [36, 15, 8, 29, 31]. We take a similar approach.

Our goal is to develop a system that uses dynamic analysis to find common faults, that is easy to use, and yet thorough and scalable. The developer should be able to submit an app binary to the system, and then within a short amount of time obtain a report. This report should provide a correct stack trace and a trace of interactions or inputs for each failure. We anticipate the system being used by developers interactively while debugging, as well as a part of regular nightly and weekly regression tests, so speed is important. An ideal way to deploy the system is as a service in the cloud, so the ability to balance resource consumption and discovering faults is also important.

We describe VanarSena, a system that meets these goals. The starting point in the design is to identify what types of faults have the highest "bang for the buck" in terms of causing real-world failures. To this end, we studied 25 million crash reports from 116,000 Windows Phone apps reported in 2012. Three key findings inform our design: first, over 90% of the crashes were attributable to only 10% of all the root causes we observed. Second, although the "90-10" rule holds, the root causes affect a wide variety of execution paths in an app. Third, a significant fraction of these crashes can be mapped to externally induced events, such as unhandled HTTP error codes (see §2).

The first finding indicates that focusing on a small number of root causes will improve reliability significantly. The second suggests that the fault finder needs to cover as many execution paths as possible. The third indicates that software emulation of user inputs, network behavior, and sensor data is likely to be effective, even without deploying on phone hardware.

Using these insights, we have developed VanarSena,[1] a system that finds faults in mobile applications. The developer uploads the app binary to the service, along with any supporting information such as a login and password. VanarSena instruments the app, and launches several *monkeys* to run the instrumented version on phone emulators. As the app is running, VanarSena emulates a variety of user, network and sensor behaviors to uncover and report observed failures.

A noteworthy principle in VanarSena is its "greybox" approach, which instruments the app binary before emulating its execution. Greybox testing combines the benefits of "whitebox" testing, which requires detailed knowledge of an app's semantics to model interactions and inputs, but isn't generalizable, and "blackbox" testing, which is general but not as efficient in covering execution paths.

The use of binary instrumentation enables a form of execution-path exploration we call *hit testing* (§5.1), which identifies how each user interaction maps to an event handler. Because many different interactions map to the same handler, hit testing is able to cover many more paths that are likely to produce new faults per unit time than blackbox methods. Moreover, app instrumentation makes VanarSena extensible because we or the developer can write handlers to process events of interest, such as network calls, inducing faults by emulating slow or faulty networks. We have written several fault inducers. Binary instrumentation also allows VanarSena to determine when to emulate the next user interaction in the app. This task is tricky because emulating a typical user requires knowing when the previous page has been processed and rendered, a task made easier with our instrumentation approach. We call this generation of `ProcessingCompleted` event (§5.2).

We have implemented VanarSena for Windows Phone apps, running it as an experimental service. We evaluated VanarSena empirically by testing 3,000 apps from the Windows Phone store for commonly-occurring faults. VanarSena discovered failures in 1,108 of these apps, which have presumably undergone some testing and real-world use[2]. Overall, VanarSena detected 2,969 crashes, including 1,227 that were not previously reported. The testing took 4500 machine hours on 12 medium Azure-class machines (average of 1.5 hours per app). At current Azure prices, the cost of testing is roughly 25 cents per app. These favorable cost and time estimates result from VanarSena's use of *hit testing* and `ProcessingCompleted` event.

While many prior systems [36, 15, 8, 29, 31] have analyzed mobile apps in various ways by exercising them with automated "monkeys" and by inducing various faults, this paper makes three new research contributions, that have general applicability.

Our first contribution is the study of 25 million crash reports from windows phone apps (§2). We believe that this is the first study of its kind. The insights from this study anchor the design of VanarSena. Other mobile app testing systems such as Dynodroid [31] can also benefit from these insights. For example, Dynodroid currently does not inject faults due to external factors such as bad networks or event timing related to unexpected or abnormal user behavior. Our study shows that these are among the most common root causes of real-world crashes.

Our second contribution is the technique of *hit testing* (§5.1). This technique allows VanarSena to speed up testing significantly. Our third contribution is the idea of generation and use of `ProcessingCompleted` event (§5.2). We show that this is necessary to both speed up the testing, and to correctly simulate both *patient* and *impatient* users. We are not aware of any other app testing framework that incorporates comparable techniques.

While the implementation details of hit testing and generation of processing completed event are specific to VanarSena, the core ideas behind both of them are quite general, and can be used by other testing frameworks as well.

2. APP CRASHES IN-THE-WILD

To understand why apps crash in the wild, we analyze a large data set of crash reports. We describe our data set, our method for determining the causes of crashes, and the results of the analysis.

2.1 Data Set

Our data set was collected by Windows Phone Error Reporting (WPER) system, a repository of error reports from all deployed Windows Phone apps. When an app crashes due to an unhandled exception, the phone sends a crash report to WPER with a small sampling probability[3]. The crash report includes the app ID, the exception type, the stack trace, and device state information such as the amount of free memory, radio signal strength, etc.

We study over 25 million crash reports from 116,000 apps collected in 2012. Figure 1 shows the number of crash reports per app. Observe that the data set is not skewed by crashes from handful of bad apps. A similar analysis shows that the data is not skewed by a small number of device types, ISPs, or countries of origin.

2.2 Root Causes of Observed Crashes

[1]VanarSena in Hindi means an "army of monkeys".

[2]Thus, VanarSena would be even more effective during earlier stages of development

[3]The developer has no control over the probability.

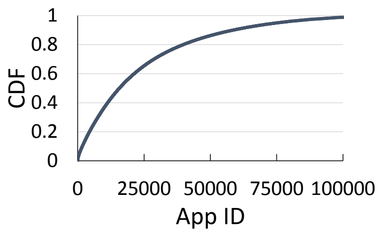

Figure 1: CDF of crash reports per app.

```
0: TransitTracker.BusPredictionManager.ReadCompleted
1: System.Net.WebClient.OnOpenReadCompleted
2: System.Net.WebClient.OpenReadOperationCompleted
...
```

Figure 2: Stack trace fragment for Chicago Transit Tracker crash. The exception was WebException.

To determine the root cause of a crash, we start with the stack trace and the exception type. An exception type gives a general idea about what went wrong, while the stack trace indicates where things went wrong. An example stack fragment is shown in Figure 2. Here, a WebException was thrown, indicating that something went wrong with a web transfer, causing the `OnOpenReadCompleted` function of the `WebClient` class to throw an exception. The exception surfaced in the `ReadCompleted` event handler of the app, which did not handle it, causing the app to crash.

We partition crash reports that we believe originate due to the same root cause into a collection called a *crash bucket*: each crash bucket has a specific exception type and system function name where the exception was thrown. For example, the crash shown in Figure 2 will be placed in the bucket labeled `WebException, System.Net.WebClient.OnOpenReadCompleted`.

Given a bucket, we use two techniques to determine the likely root cause of its crashes. First, we use data mining techniques [5] to discover possible patterns of unusual device states (such as low memory or poor signal strength) that hold for all crashes in the bucket. For example, we found that all buckets with label (OutOfMemoryException, *) have the pattern `AvailableMemory = 0`.

Second, given a bucket, we manually search various Windows Phone developer forums such as `social.msdn.microsoft.com` and `stackoverflow.com` for issues related to the exception and the stack traces in the bucket. We limit such analysis to only the 100 largest buckets, as it is not practical to investigate all buckets and developer forums do not contain enough information about many rare crashes. We learned enough to determine the root cause of 40 of the top 100 buckets.

2.3 Findings

A small number of large buckets cover most of the crashes. Figure 3 shows the cumulative distribution of various bucket sizes. The top 10% buckets cover more than 90% crashes (note the log-scale on the x-axis). This suggests that we can analyze a small number of top buckets and still cover a large fraction of crashes. Table 1 shows several large buckets of crashes.

A significant fraction of crashes can be mapped to well-defined externally-inducible root causes. We use the following taxonomy to classify various root causes. A root cause is *deterministically inducible* if it can be reproduced by deterministically modifying the external factors on which the app depends. For example, crashes of a networked app caused by improperly handling an HTTP Error 404 (Not Found) can be induced by an HTTP proxy that returns Error 404 on a Get request. Some crashes such as those due to memory faults or unstable OS states are not deterministically inducible. We further classify inducible causes into two categories: device and input. Device-related causes can be induced by systematically manipulating device states such as available memory, available storage, network signal, etc. Input-related causes can be induced by manipulating various external inputs to apps such as user inputs, data from network, sensor inputs, etc.

Table 1 shows several top crash buckets, along with their externally-inducible root causes and their categories. For example, the root causes behind the bucket with label (`WebException, WebClient.OnDownloadStringCompleted`) are various HTTP Get errors such as 401 (Unauthorized), 404 (Not Found), and 405 (Method Not Allowed), and can be induced with a web proxy intercepting all network communication to and from the app.

We were able to determine externally-inducible root causes of 40 of the top 100 buckets; for the remaining buckets, we either could not determine their root causes from information in developer forums or identify any obvious way to induce the root causes. Together, these buckets represent around 48% of crashes in the top 100 buckets (and 35% of all crashes); the number of unique root causes for these buckets is 8.

These results imply that a significant number of crashes can be induced with a relatively small number of root causes.

Although a small number, the dominant root causes affect many different execution paths in an app. For example, the same root cause of HTTP Error 404 can affect an app at many distinct execution points where the app downloads data from a server. To illustrate how often it happens, we consider all crashes from one particular app in Figure 4 and count the number of distinct stack traces in various crash buckets of the app. The higher the number of distinct stack traces in a bucket, the more the distinct execution points where the app crashed due to the same root causes responsible for the bucket. As shown in Figure 4, for 25 buckets, the number of distinct stack traces is more than 5. The trend holds in general, as shown in Figure 5, which plots the distribution of distinct stack traces in all (app, bucket) partitions. We find that it is common for the same root cause to affect many tens of execution paths of an app.

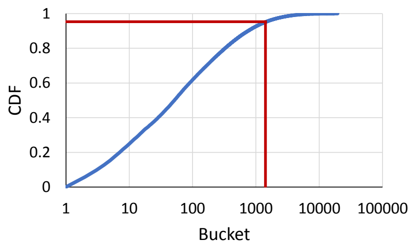

Figure 3: Cumulative distribution of bucket sizes

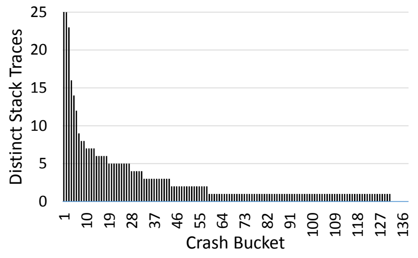

Figure 4: Distinct stack traces in various buckets for one particular app

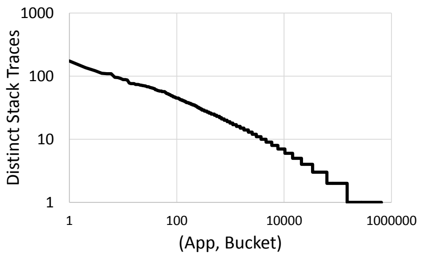

Figure 5: Distinct stack traces in various buckets for all apps

Rank (Fraction)	Bucket		Root Cause	Category	HowToInduce
	Exception	Crash Function			
1 (7.51%)	OutOfMemory Exception	*	WritablePages = 0	Device/Memory	Memory pressure
2 (6.09%)	InvalidOperation Exception	ShellPageManager.CheckHResult	User clicks buttons or links in quick succession, and thus tries to navigate to a new page when navigation is already in progress	Input/User	Impatient user
3 (5.24%)	InvalidOperation Exception	NavigationService.Navigate			
8 (2.66%)	InvalidOperation Exception	NavigationService.GoForwardBackCore			
12 (1.16%)	WebException	Browser.AsyncHelper.BeginOnUI	Unable to connect to remote server	Input/Network	Proxy
15 (0.83%)	WebException	WebClient.OnDownloadStringCompleted	HTTP errors 401, 404, 405		
5 (2.30%)	XmlException	*	XML Parsing Error	Input/Data	Proxy
11 (1.14%)	NotSupportedException	XmlTextReaderImpl.ParseDoctypeDecl			
37 (0.42%)	FormatException	Double.Parse	Input Parsing Error	Input/User, Input/Data	Invalid text entry, Proxy
50 (0.35%)	FormatException	Int32.Parse			

Table 1: Examples of crash buckets and corresponding root causes, categories, and ways to induce the crashes

3. GOALS AND NON-GOALS

Our goal is to build a scalable, easy to use system that tests mobile apps for common, externally-inducible faults as thoroughly as possible. We want to return the results of testing to the developer as quickly as possible, and for the system to be deployable as a cloud service in a scalable way.

VanarSena does not detect *all* app failures. For example, VanarSena cannot detect crashes that result from hardware idiosyncrasies, or failures caused by specific inputs, or even failures caused by the confluence of multiple simultaneous faults that we do test for. VanarSena also cannot find crashes that result from erroneous state maintenance; for example, an app may crash only after it has been run hundreds of times because some log file has grown too large.

Before we describe the architecture of VanarSena, we need to discuss how we measure *thoroughness*, or coverage. Coverage of testing tools is traditionally measured by counting the fraction of basic blocks [9] of code they cover. However, this metric is not appropriate for our purpose. Mobile apps often include third party libraries of UI controls (e.g., fancy UI buttons) Most of the code in these libraries is inaccessible at run time, because the app typically uses only one or two of these controls. Thus, coverage, as measured by basic blocks covered would look unnecessarily poor.

Instead, we focus on the user-centric nature of mobile apps. A mobile app is typically built as a collection of pages. An example app called *AroundMe* is shown in Figure 6. The user navigates between pages by interacting with controls on the page. For example, each category listing on page 1 is a control. By clicking on any of the business categories on page 1, the user would navigate to page 2. Page 1 also has a swipe control. By swiping on the page, the user ends up on the search page (page 4). From a given page, the user can navigate to the parent page by pressing the back button. The navigation graph of the app is shown in Figure 7. The nodes of the graph represent pages, while the edges represent unique *user transactions* [37] that cause the user to move between pages. Thus, we measure coverage in terms of unique pages visited [8], and unique user transactions mimicked by the tool. In §7.2, we will show that we cover typical apps as thoroughly as a human user.

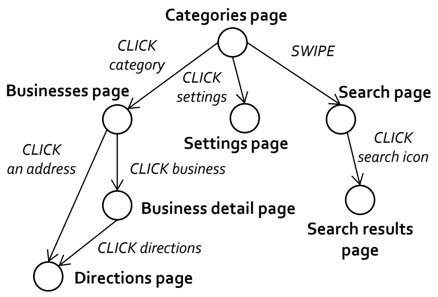

Figure 7: App structure for the example in Figure 6.

4. ARCHITECTURE

Figure 8 shows the architecture of VanarSena. VanarSena instruments the submitted app binary. The *Monkey manager* then spawns a number of *monkeys* to test the app. A *monkey* is a UI automation tool built around the Windows Phone

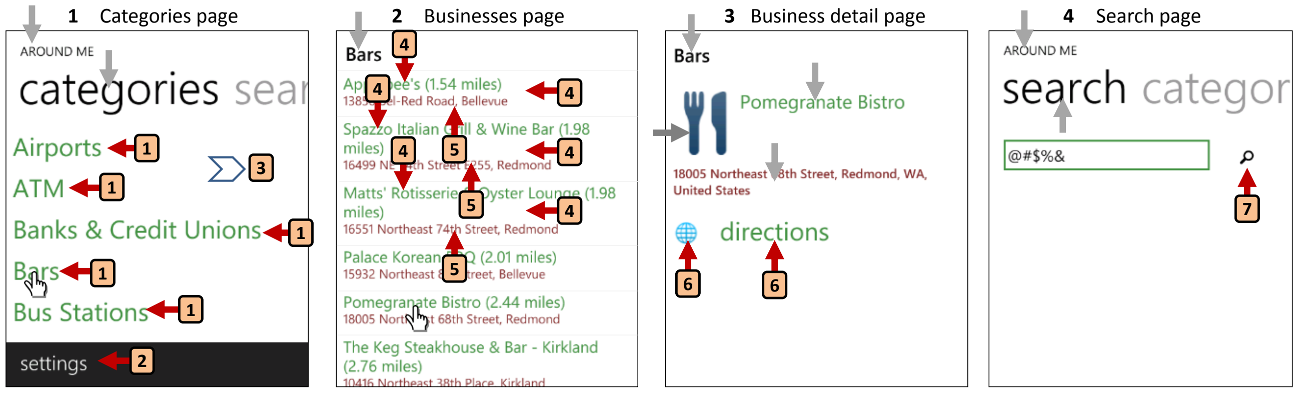
1 Categories page
AROUND ME
categories sear
Airports
ATM
Banks & Credit Unions
Bars
Bus Stations
settings
2 Businesses page
Bars
Spazzo Italian Grill & Wine Bar (1.98 miles)
Matts' Rotisserie Oyster Lounge (1.98 miles)
16551 Northeast 74th Street, Redmond
Palace Korean BBQ (2.01 miles)
Pomegranate Bistro (2.44 miles)
18005 Northeast 68th Street, Redmond
The Keg Steakhouse & Bar - Kirkland (2.76 miles)
3 Business detail page
Bars
Pomegranate Bistro
18005 Northeast 68th Street, Redmond, WA, United States
directions
4 Search page
AROUND ME
search categor
@#$%&
1
2
3
4
5
6
7

Figure 6: Example app pages. UI elements pointed by red arrows can be interacted with. Arrows and numbers will be explained in Section 5.1.

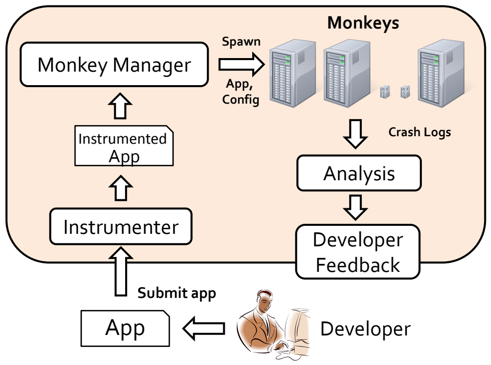

Figure 8: VanarSena Architecture. Components in the shaded box run in the cloud.

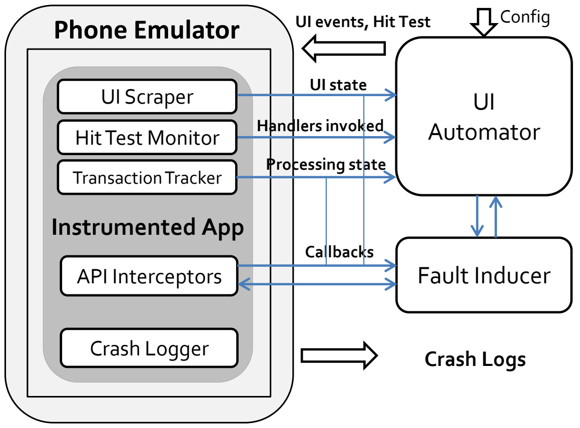

Figure 9: Monkey design.

Emulator. The monkey can automatically launch the app in the emulator and interact with the UI like a user. When the app is *monkeyed*, we systematically feed different inputs and emulate various faults. If the app crashes, the monkey generates a detailed crash report for the developer. Figure 9 shows the key components of the monkey.

Emulator: We use an off-the-shelf Windows Phone emulator in our implementation. We intentionally do not modify the emulator in any way. The key benefit of using an emulator instead of device hardware is scalability: VanarSena can easily spin up multiple concurrent instances in a cloud infrastructure to accelerate fault-finding.

Instrumentation: The instrumenter runs over the app binary; it adds five modules to the app as shown in Figure 9. At run-time, these modules generate information needed for UI Automator and the Fault Inducer.

UI Automator: The UI Automator (UIA) launches and navigates the instrumented app in the emulator. It emulates user interactions such as clicking buttons, filling textboxes, and swiping. It incorporates techniques to ensure both coverage and speed (§5).

Fault Inducer: During emulated execution, the Fault Inducer (FI) systematically induces different faults at appropriate points during execution (§6).

5. UI AUTOMATOR

As the UIA navigates through the app, it needs to make two key decisions: what UI control to interact with next, and how long to wait before picking the next control. In addition, because of the design of each monkey instance, VanarSena adopts a "many randomized concurrent monkeys" approach, which we discuss below.

To pick the next control to interact with, the UIA asks the *UI Scraper* module (Figure 9) for a list of visible controls on the current page (controls may be overlaid atop each other).

In one design, the UIA can systematically explore the app by picking a control that it has not interacted with so far, and emulating pressing the back button to go back to the previous page if all controls on a page have been interacted with. If the app crashes, VanarSena generates a crash report, and the monkey terminates.

Such a simple but systematic exploration has three problems that make it unattractive. First, multiple controls often lead to the same next page. For example, clicking on any of the business categories on page 1 in Figure 6 leads to the Business page (page 2), a situation represented by the single edge between the pages in Figure 7. We can accelerate testing in this case by invoking only one of these "equiva-

```
void btnFetch_Click(object sender, EventArgs e) {
  if (HitTestFlag == true) {
    HitTest.MethodInvoked(12, sender, e);
    return;
  }

  // Original Code
}
```

Figure 10: Event Handlers are instrumented to enable Hit Testing. Handler's unique id is 12.

lent" controls, although it is possible that some of these may lead to failures and not others (a situation mitigated by using multiple independent monkeys).

Second, some controls do not have any event handlers attached to them. For example, the title of the page may be a text-box control that has no event handlers attached to it. UIA should not waste time interacting with such controls, because it will run no app code.

Last but not least, a systematic exploration can lead to dead ends. Imagine an app with two buttons on a page. Suppose that the app always crashes when the first button is pressed. If we use systematic exploration, the app would crash after the first button is pressed. To explore the rest of the app, the monkey manager would have to restart the app, and ensure that the UIA does not click the first button again. Maintaining such state across app invocations is complicated and makes the system more complex for many reasons, prominent among which is the reality that the app may not even display the same set of controls on every run!

We address the first two issues using a novel technique we call *hit testing* (§5.1), and the third by running multiple independent random monkeys concurrently (§5.3).

5.1 Hit Testing

Because static analysis cannot accurately determine which controls on a page are invokable and lead to distinct next pages, we develop a run-time technique called hit testing. The idea is to test whether (and which) event handler in the app is activated when a control is interacted with.

Hit testing works as follows. The instrumentation framework instruments all UI event handlers in an app with a *hit test monitor*. It also assigns each event handler a unique ID. Figure 10 shows an example. When hit testing is enabled, interacting with a control will invoke the associated event handler, but the handler will simply return after informing the UIA about the invocation, without executing the event handler code.

On each new page, UIA sets the HitTestFlag and interacts with all controls on the page, one after the other. At the end of the test, the UIA can determine which controls lead to distinct event handlers. UIA can test a typical page within a few hundred milliseconds.

The arrows and the associated numbers in Figure 6 shows the result of hit tests on pages. For example, clicking any item on the categories page leads to the same event handler, while clicking on the word "categories" on that page does not invoke any event handler (gray arrow). In fact, the con-

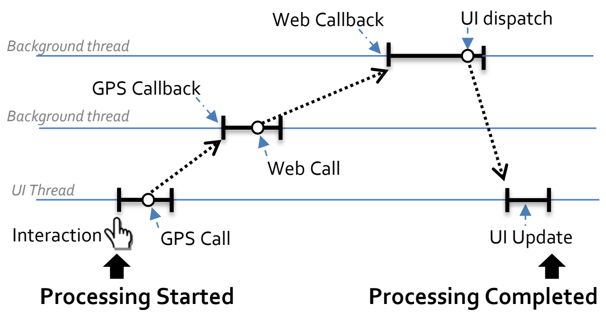

Figure 11: App Busy and Idle events.

trols on the page lead to just three unique event handlers: clicking on one of the categories leads to event handler 1, clicking on settings leads to handler 2 and swiping on the page leads to handler 3. Note also that several controls on page 1 have no event handlers attached them (gray arrows). By using hit testing, the monkey can focus only on controls that have event handlers associated with them. And from different controls associated with the same event handler, it needs to pick only one[4], thereby significantly reducing the testing time. In §7.2, we will evaluate the impact of hit testing.

5.2 When to interact next?

Emulating an "open loop" or impatient user is straightforward because the monkey simply needs to invoke event handlers independent of whether the current page has properly been processed and rendered, but emulating a real, patient user who looks at the rendered page and then interacts with it is trickier. Both types of interactions are important to test. The problem with emulating a patient user is that it is not obvious when a page has been completely processed and rendered on screen. Mobile applications exhibit significant variability in the time they take to complete rendering: we show in §7 (Figure 21) that this time could vary between a few hundred milliseconds to several seconds. Waiting for the longest possible timeout using empirical data would slow the monkey down to unacceptable levels.

Fortunately, VanarSena's greybox binary instrumentation provides a natural solution to the problem, unlike blackbox techniques. The instrumentation includes a way to generate a signal that indicates that processing of the user interaction is complete. (Unlike web pages, app pages do not have a well-defined page-loaded event [40] because app execution can be highly asynchronous. So binary instrumentation is particularly effective here.)

We instrument the app to generate a signal that indicates that processing of the user interaction is complete. We use techniques developed in AppInsight [37] to generate the signal, as follows.

The core ideas in AppInsight are the concept of *user transaction*, and techniques to track their progress. For example,

[4]In other words, we assume that two controls that lead to same event handler are equivalent. See §5.3 for a caveat.

Figure 11 shows the user transaction [37] for the interaction with "Bars" in Figure 6. The thick horizontal lines represent thread executions, while the dotted lines link asynchronous calls to their corresponding callbacks [37]. When a category (e.g. "Bars") is clicked on page 1, it calls the associated event handler, which in turn makes an asynchronous call to get GPS location. After obtaining the location, the callback thread makes another asynchronous call to a web server to fetch information about bars near that location. The web callback thread parses the results and initiates a dispatcher call to update the UI with the list of bars. To track this transaction, we instrument the app to add transaction tracker (Figure 9). It monitors the transaction at runtime and generates a `ProcessingCompleted` event when all the processing (synchronous and asynchronous) associated with an interaction is complete. Two key problems in tracking the transaction are (a) monitoring thread start and ends with minimal overhead, and (b) matching asynchronous calls with their callbacks, across thread boundaries. We address these problems using techniques from AppInsight [37]. The key difference between AppInsight and transaction tracker is that our tracker is capable of tracking the transaction in an online manner (i.e. during execution). In contrast, AppInsight generated logs that were analyzed by an offline analyzer.

5.3 Randomized Concurrent Monkeys

VanarSena uses many simple monkeys operating independently and at random, rather than build a single more complicated and stateful monkey.

Each monkey picks a control at random that would activate an event handler that it has not interacted with in past. For example, suppose the monkey is on page 1 of Figure 6, and it has already clicked on settings previously, then it would choose to either swipe (handler 3), or click one of the businesses at random (handler 1).

If no such control is found, the monkey clicks on the back button to travel to the parent page. For example, when on page 3 of Figure 6, the monkey has only one choice (handler 6). If it finds itself back on this page after having interacted with one of the controls, it will click the back button to navigate back to page 2. Pressing the back button in page 1 will quit the app.

Because an app can have loops in its UI structure (e.g. a "Home" button deep inside the app to navigate back to the first page), running the monkey once may not fully explore the app. To mitigate this, we run several monkeys concurrently. These monkeys do not share state, and make independent choices.

Running multiple, randomized monkeys in parallel has two advantages over a single complicated monkey. First, it overcomes the problem of deterministic crashes. Second, it can improve coverage. Note that we assumed that when two controls lead to the same event handler, they are equivalent. While this assumption generally holds, it is not a fact. One can design an app where all button clicks are handled by a single event handler, which takes different actions depending on the button's name. Random selection of controls ensures that different monkeys would pick different controls tied to the same event handler, increasing coverage for apps that use this coding pattern.

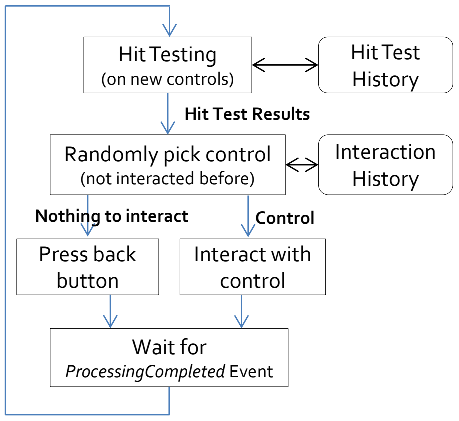

Figure 12: UI automator flow.

```
Original code
void fetch(string url) {
  WebRequest.GetResponse(url, callback);
}
Rewritten code
void fetch(string url) {
  WebRequestIntercept.GetResponse(url, callback);
}
class WebRequestIntercept {
  void GetResponse(string url, delegate callback) {
    if (MonkeyConfig.InducingResponseFaults)
       ResponseFaultInducer.Proxy(url, callback);
     if (MonkeyConfig.InducingNetworkFaults)
       NetworkFaultInducer.RaiseNetworkEvent();
  }
}
```

Figure 13: Intercepting web API to proxy through web response FIM and informing network FIM about the impending network transfer.

Putting it all together: Figure 12 shows the overall flow of the UI automator.

6. INDUCING FAULTS

The Fault Inducer (FI) is built as an extensible module in which various fault inducing modules (FIM) can be plugged in. The monkey manager configures each monkey to turn on one or more FIMs.

The FIMs are triggered by the instrumentation added to the app. The binary instrumentation rewrites the app code to intercept calls to specific APIs to proxy them through the appropriate FIM. Figure 13 shows an example. When the call to the HTTP API is made at run-time, it can be proxied through the FIM that mimics web errors. The FIM may return an HTTP failure, garble the response, and so forth.

We built five FIMs that help uncover some of the prominent crash buckets in Table 1. The first three intercept API calls and return values that apps may overlook, while the others model unexpected user behavior.

(1) Web errors: When an app makes a HTTP call, the FIM intercepts the calls and returns HTTP error codes such as 404 (Not Found) or 502 (Bad Gateway, or unable to connect). These can trigger WebExceptions. The module can also intercept the reply and garble it to trigger parsing errors. Parsing errors are particularly important for apps that obtain data from third-party sites. We use Fiddler[3] to intercept and manipulate web requests.

(2) Poor Network conditions: Brief disconnections and poor network conditions can trigger a variety of network errors, leading to WebExceptions. To emulate these network conditions, we instrument the app to raise an event to the FI just before an impending network transfer. The FIM can then emulate different network conditions such as brief disconnection, slow network rate, or long latency. We use a DummyNet-like tool [38] to simulate these conditions.

(3) Sensor errors: We introduce sensor faults by returning null values and extreme values for sensors such as GPS and accelerometers.

(4) Invalid text entry: A number of apps do not validate user inputs before parsing them. For example, MyStocks, a prominent stock tracking app, crashes if a number is entered in the box meant for stock symbols. To induce these faults, the UIA and the FI work together. The UI Scraper generates an event to the FI when it encounters a textbox. The FIM then informs the UIA to either leave the textbox empty, or fill it with text, numbers, or special symbols.

(5) Impatient user: In §5.2, we described how the UIA emulates a patient user by waiting for the `ProcessingCompleted` event. However, real users are often impatient, and may interact with the app again before processing of the previous interaction is complete. For example, in Figure 6, an impatient user may click on "Bars" on page 1, decide that the processing is taking too long, and click on the back button to try and exit the app. Such behavior may trigger race conditions in the app code. Table 1 shows that it is the root cause of many crashes. To emulate an impatient user, the transaction tracker in the app raises an event to the FI when a transaction starts, i.e., just after the UIA interacted with a control. To emulate an impatient user, the FIM then instructs the UIA to immediately interact with another specific UI control, without waiting for `ProcessingCompleted` event. We emulate three distinct impatient user behaviors—clicking on the same control again, clicking on another control on the page, and clicking on the back button.

It is important to be careful about when faults are induced. When a FIM is first turned on, it does not induce a fault on every intercept or event, because it can result in poor coverage. For example, consider testing the AroundMe app (Figure 6) for web errors. If the FIM returns 404 for every request, the app will never populate the list of businesses on page 2, and the monkey will never reach page 3 and 4 of the app. Hence, a FIM usually attempts to induce each fault with some small probability. Because VanarSena uses multiple concurrent monkeys, this approach works in practice.

During app testing, VanarSena induces only one fault at a time: each one instance of the monkey runs with just one FIM turned on. This approach helps us pinpoint the fault that is responsible for the crash. The monkey manager runs multiple monkeys concurrently with different FIMs turned on.

7. EVALUATION

We evaluate VanarSena along two broad themes. First, we demonstrate the usefulness of the system by describing the crashes VanarSena found on 3,000 apps from the Windows Phone Store. Then, we evaluate the optimizations and heuristics described in §5.

To test the system, we selected apps as follows. We bucketized all apps that were in the Windows Phone app store in the first week of April 2013 into 6 groups, according to their rating (no rating, rating $\leq 1, \cdots$, rating ≤ 5). We randomly selected 500 apps from each bucket. This process gives us a representative set of 3,000 apps to test VanarSena with.

We found that 15% of these apps had a textbox on the first page. These might have required user login information, but we did not create such accounts for the apps we evaluated. So it is possible (indeed, expected) that for some apps, we didn't test much more than whether there were bugs on the sign-in screen. Despite this restriction, we report many bugs, suggesting that most (but not all) apps were tested reasonably thoroughly. In practice, we expect the developer to supply app-specific inputs such as sign-in information.

7.1 Crashes

We ran 10 concurrent monkeys per run, where each run tests one of the eight fault induction modules from Table 3, as well as one run with no fault induction. Thus, there were 9 different runs for each app, 90 monkeys in all. In these tests, the UIA emulated a patient user, except when the "impatient user" FIM was turned on.

We ran the tests on 12 Azure machines, set up to both emulate Windows Phone 7 and Windows Phone 8 in different tests. Overall, testing 3,000 apps with 270,000 distinct monkey runs took 4,500 machine hours, with an estimated modest cost of about \$800 for the entire corpus, or ≈ 25 cents per app on average for one complete round of tests, a cost small enough for nightly app tests to be done. The process emulated over 2.5 million interactions, covering over 400,000 pages.

7.1.1 Key Results

Overall, VanarSena flagged 2969 unique crashes[5] in 1108 apps. Figure 14 shows that it found one or two crashes in

[5]The uniqueness of the crash is determined by the exception type and stack trace. If the app crashes twice in exactly the same place, we count it only once.

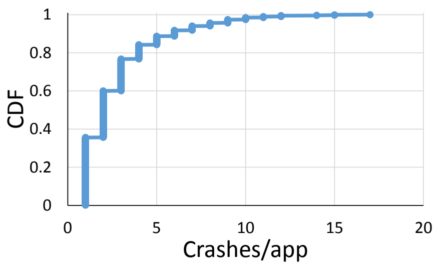

Figure 14: Crashes per app

Rating value	VanarSena	WPER
None	350 (32%)	21%
1	127 (11%)	13%
2	146 (13%)	16%
3	194 (18%)	15%
4	185 (17%)	22%
5	106 (10%)	13%

Table 2: Number of crashed apps for various ratings

60% of the apps. Some apps had many more crashes—one had 17!

Note that these crashes were found in apps that are already in the marketplace; these are not "pre-release" apps. VanarSena found crashes in apps that have already (presumably!) undergone some degree of testing by the developer.

Table 2 bucketizes crashed apps according to their ratings rounded to nearest integer values. Note that we have 500 total apps in each rating bucket. We see that VanarSena discovered crashes in all rating buckets. For example, 350 of the no-rating 500 apps crashed during our testing. This represents 31% of total (1108) apps that crashed. We see that the crash data in WPER for these 3000 apps has a similar rating distribution except for the 'no-rating' bucket. For this bucket, WPER sees fewer crashes than VanarSena most likely because these apps do not have enough users (hence no ratings).

7.1.2 *Comparison Against the WPER Database*

It is tempting to directly compare the crashes we found with the crash reports for the same apps in the WPER database discussed in §2. Direct comparison, however, is not possible because both the apps and the phone OS have undergone revisions since the WPER data was collected. But we can compare some broader metrics.

VanarSena found 1,227 crashes not in the WPER database. We speculate that this is due to two reasons. First, the database covers a period of one year. Apps that were added to the marketplace towards the end of the period may not have been run sufficiently often by users. Also, apps that are unpopular (usually poorly rated), do not get run very often in the wild, and hence do not encounter all conditions that may cause them to crash.

The crashes found by VanarSena cover 16 out of 20 top crash buckets (exception name plus crash method) in WPER, and 19 of the top 20 exceptions. VanarSena does not report any `OutOfMemoryException` fault because we have not written a FIM for it; we tried a few approaches, but have not yet developed a satisfactory test. Moreover, most apps that have this fault are games, which VanarSena does not test adequately at this time (§8).

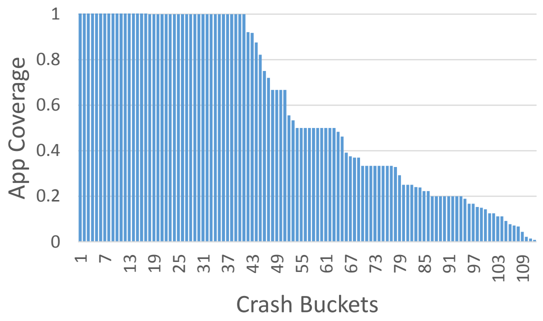

Figure 15: Coverage of crash buckets in WPER data

Figure 15 shows another way to compare VanarSena crash data and WPER. For this graph, we consider the subset of WPER crashes that belong to the crash buckets and the apps for which VanarSena found at least one crash. For each bucket, we take the apps that appear in WPER, and compute what fraction of these apps are also crashed by VanarSena. We call this fraction *bucket coverage*. Figure 15 shows that for 40% of the buckets, VanarSena crashed *all* the apps reported in WPER, which is a significant result suggesting good coverage.

7.1.3 *Analysis*

Even "no FIM" detects failures. Table 3 shows the breakdown of crashes found by VanarSena. The first row shows that even without turning any FIM on, VanarSena discovered 506 unique crashes in 429 apps (some apps crashed multiple times with distinct stack traces; also, the number of apps in this table exceeds 1108 for this reason). The table also gives the name of an example app in this category. The main conclusion from this row is that merely exploring the app thoroughly can uncover faults. A typical exception observed for crashes in this category is the `NullReferenceException`. The table also shows that 239 of these 506 crashes (205 apps) were not in the WPER database.

We now consider the crashes induced by individual FIMs. To isolate the crashes caused by a FIM, we take a conservative approach. If the signature of the crash (stack trace) is also found in the crashes included in the first row (i.e., no FIM), we do not count the crash. We also manually verified a large sample of crashes to ensure that they were actually being caused by the FIM used.

Most failures are found by one or two FIMs, but some apps benefit from more FIMs. Figure 16 shows the number of apps that crashed as a function of the number of FIMs that induced the crashes. For example, 235 apps required no FIM to crash them at all[6]. Most app crashes are found with less

[6]This number is less than 429 (row 1 of Table 3), because some of those 429 apps crashed with other FIMs as well. Unlike Table 3, apps in Figure 16 add up to 1108.

FIM	Crashes (Apps)	Example crashes		Not in WPER
		App Example	CrashBucket	
No FIM	506 (429)	GameStop	NullReferenceException, InvokeEventHandler	239 (205)
Text Input	215 (191)	91	FormatException, Int32.Parse	78 (68)
Impatient User	384 (323)	DishOnIt	InvalidOperationException, Navigation.GoBack	102 (89)
HTTP 404	637 (516)	Ceposta	WebException, Browser.BeginOnUI	320 (294)
HTTP 502	339 (253)	Bath Local School	EndpointNotFoundException, Browser.BeginOnUI	164 (142)
HTTP Bad Data	768 (398)	JobSearchr	XmlException, ParseElement	274 (216)
Network Poor	93 (76)	Anime Video	NotSupportedException, WebClient.ClearWebClientState	40 (34)
GPS	21 (19)	Geo Hush	ArgumentOutOfRangeException, GeoCoordinate..ctor	9 (9)
Accelerometer	6 (6)	Accelero Movement	FormatException, Double.Parse	1 (1)

Table 3: Crashes found by VanarSena.

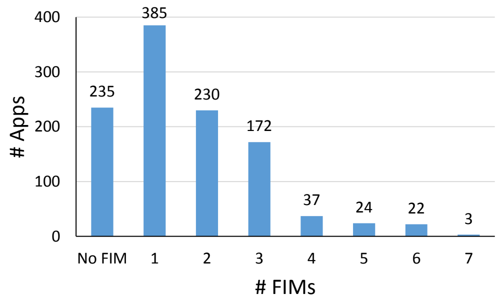

Figure 16: FIMs causing crashes

than three FIMs, but complex apps fail for multiple reasons (FIMs). Several apps don't use text boxes, networking, or sensors, making those FIMs irrelevant, but for apps that use these facilities, the diversity of FIMs is useful. The tail of this chart is as noteworthy as the rest of the distribution.

Many apps do not check the validity of the strings entered in textboxes. We found that 191 apps crashed in 215 places due to this error. The most common exception was `FormatException`. We also found web exceptions that resulted when invalid input was passed to the cloud service backing the app.

Emulating an impatient user uncovers several interesting crashes. Analysis of stack traces and binaries of these apps showed that the crashes fall in three broad categories. First, a number of apps violate the guidelines imposed by the Windows Phone framework regarding handling of simultaneous page navigation commands. These crashes should be fixed by following suggested programming practices [1]. Second, a number of apps fail to use proper locking in event handlers to avoid multiple simultaneous accesses to resources such as the phone camera and certain storage APIs. Finally, several apps had app-specific race conditions that were triggered by the impatient behavior.

Several apps incorrectly assume a reliable server or network. Some developers evidently assume that cloud servers and networks are reliable, and thus do not handle HTTP errors correctly. VanarSena crashed 516 apps in 637 unique places by intercepting web calls, and returning the common "404" error code. The error code representing Bad Gateway ("502") crashed 253 apps.

Some apps are too trusting of data returned from servers. They do not account for the possibility of receiving corrupted or malformed data. Most of the crashes in this category were due to XML and JSON parsing errors. These issues are worth addressing also because of potential security concerns.

Some apps do not correctly handle poor network connectivity. In many cases, the request times out and generates a web exception which apps do not handle. We also found a few interesting cases of other exceptions, including a `NullReferenceException`, where an app waited for a fixed amount of time to receive data from a server. When network conditions were poor, the data did not arrive during the specified time. Instead of handling this possibility, the app tried to read the non-existent data.

A handful of apps do not handle sensor failures or errors. When we returned a `NaN` for the GPS coordinates, which indicates that the GPS is not switched on, some apps crashed with `ArgumentOutOfRangeException`. We also found a timing-related failure in an app where it expected to get a GPS lock within a certain amount of time, failing when that did not happen.

API compatibility across OS versions caused crashes. For example, in the latest Windows Phone OS (WP8), the behavior of several APIs has changed [2]. WP8 no longer supports the FM radio feature and developers were advised to check the OS version before using this feature. Similar changes have been made to camera and GPS APIs. To test whether the apps we selected are susceptible to API changes, we ran them with the emulator emulating WP8. The UIA emulated patient user, and no FIMs were turned on. We found that 8 apps crashed with an `RadioDisabledException`, while the camera APIs crashed two apps. In total, we found about 221 crashes from 212 apps due to API compatibility issues[7].

7.2 Monkey Techniques

We now evaluate the heuristics and optimizations discussed in §5. Unless specified otherwise, the results in this section use the same 3000 apps as before. The apps were run 10 times, with no FIM, and the UIA emulated a patient user.

7.2.1 *Coverage*

We measure coverage in terms of pages and user transactions. We desire that the monkey should cover as much of the app as possible. However, there is no easy way to

[7]Note that this data is not included in any earlier discussion (e.g. Table 3) since we used Windows 7 emulator for all other data.

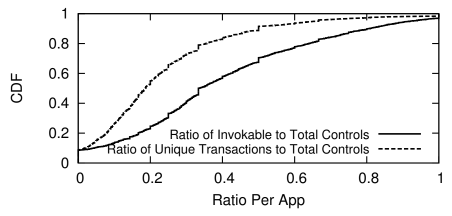

Figure 17: Fraction of invokable to total controls and unique event handlers to total controls in an app.

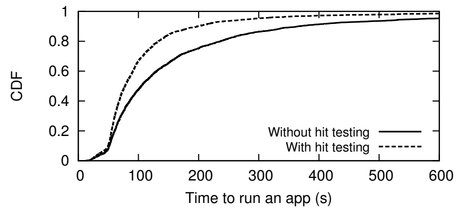

Figure 18: Time to run apps with and without hit testing

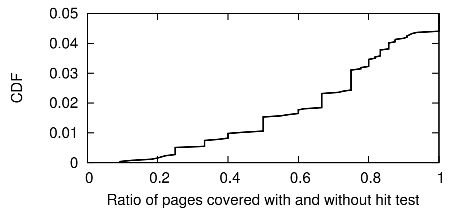

Figure 19: Fraction of pages covered with and without hit testing.

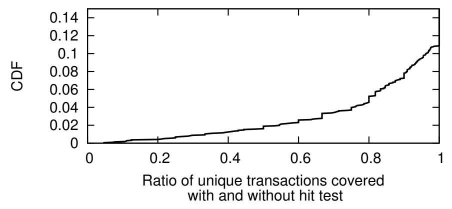

Figure 20: Fraction of transactions covered with and without hit testing.

determine how many unique pages or user transactions the app contains. Any static analysis may undercount the pages and controls, since some apps generate content dynamically. Static analysis may also overestimate their numbers, since apps often include 3rd party libraries that include a lot of pages and controls, only a few of which are accessible to the user at run-time.

Thus, we rely on human calibration to thoroughly explore a small number of apps and compare it to monkey's coverage. We randomly picked 35 apps and recruited 3 users to manually explore the app. They were specifically asked to click on possible controls and trigger as many unique transactions as possible. We instrumented the apps to log the pages visited and the transactions invoked. Then, we ran the app through our system, with the configuration described earlier.

In 26 out of 35 apps, the monkey covered 100% of pages and more than 90% of all transactions. In five of the remaining nine apps, the monkey covered 75% of the pages. In four apps, the monkey was hampered by the need for app-specific input such as login/passwords and did not progress far. Although this study is small, it gives us confidence that the monkey is able to explore the vast majority of apps thoroughly.

7.2.2 *Benefits of Hit Testing*

Hit testing accelerates testing by avoiding interacting with non-invokable controls. Among invokable controls, hit testing allows the monkey to interact with only those that lead to unique event handlers.

To evaluate the usefulness of hit testing, we turned off randomization in the UIA, and ran the monkey with and without hit testing. When running without hit testing, we assume that every control leads to a unique event handler, so the monkey interacts with every control on the page.

Figure 17 shows the ratio of invokable controls and unique event handlers to the total controls in each app. We found that in over half the apps, less than 33% of the total controls in the app were invokable, and only 18% lead to unique event handlers.

Figure 18 shows the time to run apps with and without hit testing. The 90th percentile of the time to run the app once with no fault induction was 365 seconds without hit testing, and only 197 seconds with hit testing. The tail was even worse: for one particular app, a single run took 782 seconds without hit testing, while hit testing reduced the time to just 38 seconds, a 95% reduction!

At the same time, we found that hit testing had minimal impact on app coverage (Figure 19 and Figure 20). In 95.7% of the apps, there was no difference in page coverage with and without hit testing, and for 90% of the apps, there was no difference in transaction coverage either. For the apps with less than 100% coverage, the median page and transaction coverage was over 80%. This matches the observation made in [37]: usually, only distinct event handlers lead to distinct user transactions.

7.2.3 *Importance of the* ProcessingCompleted *Event*

When emulating a patient user, the UIA waits for the `ProcessingCompleted` event to fire before interacting with the next control. Without such an event, we would need to use a fixed timeout. We now show that using such a fixed timeout is not feasible.

Figure 21 shows distribution of the processing time for transactions in the 3000 apps. Recall (Figure 11) that this includes the time taken to complete all processing associated with a current interaction [37]. For this figure, we separate the transactions that involved network calls and those that did not. We also ran the apps while the FIM emulated typical 3G network speeds. This FIM affects only the duration of

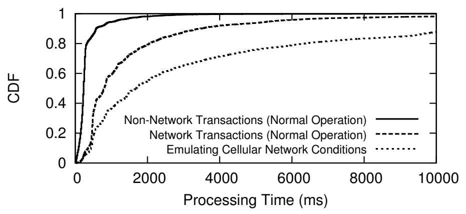

Figure 21: Processing times for transaction.

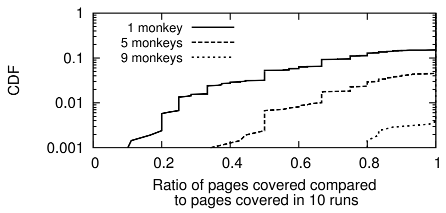

Figure 22: Fraction of pages covered by runs compared to pages covered by 10 runs.

transactions that involve networking, and the graph shows this duration as well.

The graph shows that processing times of the transactions vary widely, from a few milliseconds to over 10 seconds. Thus, with a small static timeout, we may end up unwittingly emulating an impatient user for many transactions. Worse yet, we may miss many UI controls that are populated only after the transaction is complete. On the other hand, with a large timeout, for many transactions, the UIA would find itself waiting unnecessarily. For example, a static timeout of 4 seconds covers 90% of the normal networking transactions, but is unnecessarily long for non-networking transactions. On the other hand, this value covers only 60% of the transactions when emulating a 3G network.

This result demonstrates that using the *ProcessingCompleted* event allows VanarSena to maximize coverage while minimizing processing time.

7.2.4 Multiple Concurrent Monkeys are Useful

Figure 22 shows the CDF of the fraction of pages covered with 1, 5, and 9 monkeys compared to the pages covered with 10 monkeys. The y-axis is on a log scale. Although 85% of apps need only one monkey for 100% coverage, the tail is large. For about 1% of the apps, new pages are discovered even by the 9th monkey. Similarly, Figure 23 shows that for 5% of the apps, VanarSena continues to discover new transactions even in the 9th monkey.

We did an additional experiment to demonstrate the value of multiple concurrent runs. Recall that we ran each app through each FIM 10 times. To demonstrate that it is possible to uncover more bugs if we run longer, we selected 12 apps from our set of 3000 apps that had the most crashes in WPER system. We ran these apps 100 times through each FIM. By doing so, we uncovered 86 new unique crashes among these apps (4 to 18 in each) in addition to the 60 crashes that we had discovered with the original 10 runs.

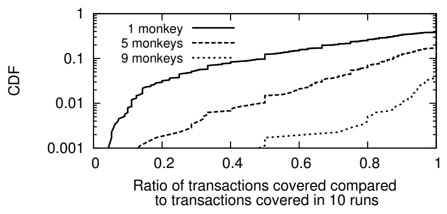

Figure 23: Fraction of transactions covered by runs compared to transactions covered by 10 runs.

8. DISCUSSION AND LIMITATIONS

Why not instrument the emulator? VanarSena could have been implemented by modifying the emulator to induce faults. As a significant practical matter, however, modifying the large and complex emulator code would have required substantially more development effort than our architecture. Moreover, it would require the fault detection software to be adapted to the emulator evolving.

Why cloud deployment? We envision VanarSena as a cloud service for a couple of reasons. First, the cloud offers elastic resources – i.e. a large number of emulators can be deployed on demand. Second, a cloud-based deployment also makes for easier updates. We can update the monkey in a variety of ways – e.g. by adding more FIMs based on crash reports from the field, or by using (as yet undiscovered) techniques for improving testing speed or resource consumption. That said, it is easy to envision non-cloud deployment models as well. For example, VanarSena can work by spawning multiple VMs on developer's desktop (resources permitting), or on a network of local machines. These other deployment scenarios have their own advantages and disadvantages. The fact that monkeys run independently of each other allows for many deployment and pricing models.

Target audience: We envision that VanarSena would be primarily used by amateur developers, or small app shops who lack resources to perform thorough testing of their apps. A majority of apps (on any platform) are developed by such entities. However, note that our evaluation did not focus on only such apps, and we were able to crash so-called professionally developed apps as well. Apart from app developers, VanarSena can also be used by app store owners during app ingestion and approval pipeline to test submitted apps for common faults. The extensibility of the fault inducer, and not requiring source code, are both significant assets in realizing this scenario.

Testing Games: Many games requires complex, free-form gestures. Thus, trace replay [19] may be a more appropriate testing strategy on game apps, than randomized monkey actions. Our monkey can easily support trace replay, although collection and validation of such traces is a challenging problem that we plan to address in future. We also note that we cannot test certain other kinds of apps with the current version of VanarSena. Some apps launch other apps (e.g. web browser) and terminate. Testing such apps requires keeping careful track of different app contexts – something which we have not yet implemented. This, however, is an engineering challenge only - not a fundamental one.

Overhead: On average, our instrumentation increases the runtime of transactions by 0.02%. This small overhead is unlikely to affect the behavior of the app.

False Positives: The binary instrumentation may itself be buggy, causing "false positive" crashes. We cannot prove that we do not induce such false positives, but careful manual analysis of crash traces shows that none of the crashes occurred in the code VanarSena added.

Combination of fault inducers: We evaluated apps by injecting one fault at a time to focus on individual faults. In reality, multiple faults may happen at the same time. We plan to investigate this in future.

Beyond Windows Phone: VanarSena currently supports Windows Phone applications. However, its techniques are broadly applicable to mobile apps and can be extended to other platforms. In [37], we have described how the instrumentation framework can be extended to other platforms.

Beyond crashes: We currently focus on app crashes only. However, app developers also typically care about performance of their app [37], under a variety of conditions. Extending VanarSena to do performance testing is the primary thrust of our future work.

9. RELATED WORK

At a high level, VanarSena consists of two components: (1) dynamic analysis with a monkey, and (2) fault injection for app testing. Below we discuss how VanarSena compares with prior works in these two aspects.

Static and dynamic analysis of mobile apps. Several prior works have statically analyzed app binaries to uncover energy bugs [35, 39], performance problems [28], app plagiarism [12], security problems [16, 21], and privacy leaks [30, 18]. Static analysis is not suitable for our goal of uncovering runtime faults of apps since it cannot capture runtime issues such as poor network condition and corrupted or unexpected responses from cloud services. Several recent works have proposed using a monkey to automatically execute mobile apps for analysis of app's runtime properties. AppsPlayground [36] runs apps in the Android emulator on top of a modified Android software stack (TaintDroid [14]) in order to track information flow and privacy leaks. Authors evaluate the tool with an impressive 3,968 apps. Recently A^{3}E [8] and Orbit [41] use combinations of static and dynamic analysis to automatically generate test cases to reach various activities of an app. Eprof [34] uses dynamic analysis (without a monkey) for fine-grained energy accounting. ProtectMyPrivacy [4] uses the crowd to analyze app privacy settings and to automatically recommend app-specific privacy recommendations. AMC [29] uses a dynamic analysis to check accessibility properties of vehicular apps. All these works differ by their end goals and specific optimizations. Similarly, VanarSena differ from them in its end goal of uncovering runtime faults of apps and its novel monkey optimization techniques: hit testing and accurate processing completed event. The optimizations are general and can be used for other systems as well. We cannot directly compare the performance of our monkey with the other systems since all of them are for Android apps.

VanarSena uses AppInsight [37] to instrument Windows Phone app binaries. For Android apps, one could use similar frameworks such as SIF [24] and RetroSkeleton [13].

Mobile app testing with a monkey. As mentioned in Section 1, mobile app testing poses different challenges than traditional "enterprise" software, motivating researchers to develop mobile-specific solutions. Researchers have used Android Monkey [20] for automated fuzz testing [6, 7, 17, 25, 33]. Similar UI automation tools exist for other platforms. VanarSena differs from these tools is two major ways. First, the Android Monkey generates only UI events, and not the richer set of faults that VanarSena induces. Second, it does not optimize for coverage or speed like VanarSena. One can provide an automation script to the Android Monkey to guide its execution paths, but this approach is not scalable when exploring a large number of distinct execution paths.

Closest to our work is DynoDroid [31] that, like VanarSena, addresses the above problems, but with a different approach: it modifies the Android framework and involves humans at run-time to go past certain app pages (e.g., login screen). Another fundamental difference is that it manipulates only UI and system events and does not inject faults due to external factors such as bad network or event timing related to unexpected or abnormal user behavior, which are among the most common root causes in our real-world crash reports. A^{3}E [8] and Orbit [41] use static and dynamic analysis to generate test cases to traverse different app activities, but do not inject external faults. ConVirt [11] is a concurrent project on mobile app fuzz testing; unlike VanarSena, it takes a blackbox approach and uses actual hardware. All these systems could benefit from our crash analysis insights to decide what faults to inject.

Other software testing techniques. Software testing has a rich history, which cannot be covered in a few paragraphs. We focus only on recent work on mobile app testing, which falls into three broad categories: fuzz testing, which generates random inputs to apps; symbolic testing, which tests an app by symbolically executing it; and model-based testing. Fuzz testing is done with a monkey and is discussed above.

As mentioned in Section 1, symbolic execution [27, 10, 33] and its hybrid variant, concolic execution [7, 26] have found limited success in testing real-world apps due to path explosion problem and difficulty in modeling real-world execution environment with network, sensors, and cloud.

"GUI ripping" [32, 23, 6] systems and GUITAR [22] use model-based testing to mobile apps. Unlike VanarSena, it requires developers to provide a model of the app's GUI and can only check faults due to user inputs. Applicability of these techniques has so far been very limited (e.g., evaluated with a handful of "toy" apps only).

10. CONCLUSION

VanarSena is a software fault detection system for mobile apps designed by gleaning insights from an analysis of 25 million crash reports. VanarSena adopts a "greybox" testing method, instrumenting the app binary to achieve both high coverage and speed, using *hit testing* and generation of `ProcessingCompleted` event. We found that VanarSena is effective in practice. We tested it on 3000 apps from the Windows Phone store, finding that 1138 of them had failures. VanarSena uncovered over 2969 distinct bugs in existing apps, including over 1227 that were not previously reported. Each app was tested, on average, in just 1.5 hours. Deployed as a cloud service, VanarSena can provide a automated testing framework to mobile software reliability even for amateur developers who cannot devote extensive resources to testing.

11. REFERENCES

[1] `http://www.magomedov.co.uk/2010/11/navigation-is-already-in-progress.html`.

[2] App platform compatibility for Windows Phone. `http://msdn.microsoft.com/en-US/library/windowsphone/develop/jj206947(v=vs.105).aspx`.

[3] Fiddler. `http://fiddler2.com/`.

[4] Y. Agarwal and M. Hall. Protectmyprivacy: Detecting and mitigating privacy leaks on ios devices using crowdsourcing. In *Mobisys*, 2013.

[5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, 1994.

[6] D. Amalfitano, A. R. Fasolino, S. D. Carmine, A. Memon, and P. Tramontana. Using gui ripping for automated testing of android applications. In *ASE*, 2012.

[7] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *FSE*, 2012.

[8] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *OOPSLA*, 2013.

[9] T. Ball and J. Larus. Efficient Path Profiling. In *PLDI*, 1997.

[10] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[11] M. Chieh et al. Contextual Fuzzing: Automated Mobile App Testing Under Dynamic Device and Environment Conditions. MSR-TR-2013-92. In Submission.

[12] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *ESORICS*, 2012.

[13] B. Davis and H. Chen. Retroskeleton: Retrofitting android apps. In *Mobisys*, 2013.

[14] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Seth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, 2010.

[15] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[16] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security*, 2011.

[17] S. Ganov, C. Killmar, S. Khurshid, and D. Perry. Event listener analysis and symbolic execution for testing gui applications. *Formal Methods and Software Engg.*, 2009.

[18] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Trust and Trustworthy Computing*, 2012.

[19] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *ICSE*, 2013.

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

[21] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: Scalable and accurate zero-day android malware detection. In *Mobisys*, 2012.

[22] GUITAR: A model-based system for automated GUI testing. http://guitar.sourceforge.net/.

[23] D. Hackner and A. M. Memon. Test case generator for GUITAR. In *ICSE*, 2008.

[24] S. Hao, D. Li, W. Halfond, and R. Govindan. SIF: A Selective Instrumenation Framework for Mobile Applications. In *MobiSys*, 2013.

[25] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *AST*, 2011.

[26] C. S. Jensen, M. R. Prasad, and A. MÃÿller. Automated testing with targeted event sequence generation. In *Int. Symp. on Software Testing and Analysis*, 2013.

[27] J. C. King. Symbolic execution and program testing. *CACM*, 19(7):385–394, 1976.

[28] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek. Mantis: Automatic performance prediction for smartphone applications. In *Usenix ATC*, 2013.

[29] K. Lee, J. Flinn, T. Giuli, B. Noble, and C. Peplin. Amc: Verifying user interface properties for vehicular applications. In *ACM Mobisys*, 2013.

[30] B. Livshits and J. Jung. Automatic mediation of privacy-sensitive resource access in smartphone applications. In *USENIX Security*, 2013.

[31] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *FSE*, 2013.

[32] A. Memon. Using reverse engineering for automated usability evaluation of gui-based applications. *Human-Centered Software Engineering*, 2009.

[33] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, 2012.

[34] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof. In *Eurosys*, 2012.

[35] A. Pathak, A. Jindal, Y. C. Hu, , and S. Midkiff. What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps. In *Mobisys*, 2012.

[36] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic security analysis of smartphone applications. In *CODASPY*, 2013.

[37] L. Ravindranath et al. Appinsight: Mobile app performance monitoring in the wild. In *OSDI*, 2012.

[38] L. Rizzo. Dummynet.

[39] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal. Towards verifying android apps for the absence of wakelock energy bugs. In *HotPower*, 2012.

[40] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying Page Load Performance with WProf. In *NSDI*, 2013.

[41] W. Yang, M. R. Prasad, and T. Xie. A Grey-box Approach for Automated GUI-Model Generation of Mobile Applications. In *FASE*, 2013.