SPADE: Scalable App Digging with Binary Instrumentation and Automated Execution

Felix Xiaozhu Lin *Rice University*
Lenin Ravindranath *Microsoft Research*
Suman Nath *Microsoft Research*
Jie Liu *Microsoft Research*

Abstract

We present SPADE, a system for quickly and automatically analyzing runtime states of a large collection of mobile apps. Such tasks are becoming increasingly important for app stores: e.g., for checking apps' runtime security and privacy properties, for capturing and indexing data inside apps for better app search, etc. SPADE uses two key techniques. First, it uses binary instrumentation to automatically insert custom code into app binary to capture its runtime state. Second, it executes an instrumented app in a phone emulator and automatically navigates through various app pages by emulating user interactions. SPADE employs a number of novel optimizations to increase coverage and speed.

We have implemented three novel applications on SPADE. Our experiments with 5,300 apps from Windows Phone app store show that these applications are extremely useful. We also report coverage of SPADE on these apps and survey various root causes that make automated execution difficult on mobile apps. Finally, we show that our optimizations make SPADE significantly fast, allowing an app store to process up to 3,000 apps per day on a single phone emulator.

1 Introduction

Mobile apps are becoming increasingly important: there are more than a million mobile apps available today in major app stores, with around 15,000 apps being released per week [6]. In this paper, we make a case for *digging* these apps. We define *app digging* as the process of capturing runtime states or analyzing runtime behaviors of a large collection of apps, by automatically executing them and navigating through various parts of the apps.

An app digging system can enable many novel applications that greatly benefit app stores (or third parties providing services on large app collections). For example, it can enable an *App Crawler* application that captures various information apps present to users during runtime. As we show later, indexing such data in an app store's search engine can significantly increase its query hit rate, compared to the current practice of indexing app metadata alone (e.g., app name, category, description, etc.). Another example is an *App Compliance Checker* that checks if an app satisfies app store's privacy and security policies during runtime. Today, app stores use mostly-manual or semi-automatic processes for this purpose. As we show later, such techniques are not thorough and can easily miss many compliance violations.

Large scale app digging is challenging for several reasons. First, the app digging system needs to automatically execute apps and navigate to various parts of the apps by simulating necessary user interactions (e.g., clicking a button, swiping a page, etc.). The system needs to understand various UI controls to interact with them. For scalability, this needs to be done without any human in the loop. Second, app stores have app binaries only. The app digging system needs to capture/analyze apps' runtime states without access to source code or any help from app developers. Moreover, the system should be flexible so that the app store can use it for various applications. Third, for scalability, the system should be as fast as possible. The app store may want to use the system periodically (e.g., for the App Crawler to capture contents that change with time) or on demand (e.g., for the Compliance Checker to see if apps pose a recently discovered privacy threat)—fast digging can save valuable time and resource. This is nontrivial for various reasons. For example, for fast navigation, the system needs to interact with an app as soon as all processing due to the last interaction is complete; robust detection of such completion is challenging due to the asynchronous nature of apps.

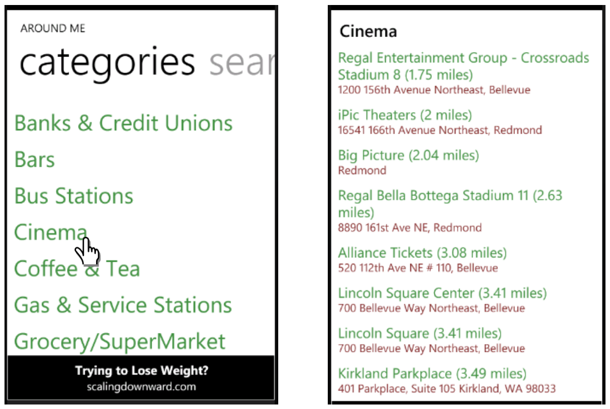
Our contributions. We make three contributions in this paper. First, we describe SPADE (Scalable and Practical App Digging Engine), a system that we have built to quickly dig a large collection of apps (§3). SPADE can take an app binary, automatically launch it, efficiently navigate it by simulating user interactions and capture the runtime state of the app. To achieve this, SPADE instruments the app binaries and employs a tool that automatically interacts with the instrumented apps running in a phone emulator. SPADE does not make any modifications to the phone OS or the emulator, and hence is readily deployable.

We make SPADE scalable with novel optimizations (§4). In particular, when programs are written in a highly asynchronous fashion, we show how to reliably find when all processing initiated by one UI interaction is finished so that SPADE can capture the correct runtime state and immediately initiate the next interaction. We also show how to detect interactable UI controls so that the system does not waste time interacting with non-responsive parts of the UI. SPADE also uses several optimizations for cases when the same app needs to be digged repeatedly at different times or with different location inputs (§5).

Note that our goal and approach are different from various recent efforts on automated execution of Android apps [1, 2, 11, 16]. They aim to help individual developers to automatically test their apps. In contrast, we aim to help *app stores* to perform *a wide variety of tasks* on *a large collection of app binaries*. Thus, unlike existing work, we aim for scalability—good digging speed and good coverage for a large variety of third party apps, and practicality—supporting various types of digging applications beyond app testing.

Second, to demonstrate the utility of SPADE, we have built three novel applications (§6): (1) an *App Crawler* that captures dynamic content that apps present to users and indexes it to improve the app store's search engine, (2) an *Ad Fraud Detector* that automatically finds apps that adopt various unacceptable ways of using ad controls, and (3) a *Contextual Ad-keyword Miner* that extracts prominent ad keywords inside apps, to be targeted with contextual ads. We have run these applications on 5,300 randomly chosen Windows Phone apps. Our experience is very encouraging: the App Crawler captured app data that can increase query hit rate of app store's search engine by 25%; the Ad Fraud Detector captured 80 apps that adopt four different fraudulent usage of ad controls to claim more money from the ad network (these apps have been in the app market for more than 1.5 years but such frauds remained undetected); and the Ad-keyword Miner extracted a large number of ad keywords (median 20 per app) that could be successfully matched with bidding keywords in Microsoft's ad network.

Third, we evaluate SPADE's coverage and speed by using 5,300 apps from the Windows Phone app store (§7). Recently proposed automated execution techniques for smartphone apps have been evaluated with a handful of toy apps [1, 2, 11, 16]. No study exists on how



Cinema

Regal Entertainment Group - Crossroads Stadium 8 (1.75 miles)
1200 156th Avenue Northeast, Bellevue

iPic Theaters (2 miles)
16541 166th Avenue Northeast, Redmond

Big Picture (2.04 miles)
Redmond

Regal Bella Bottega Stadium 11 (2.63 miles)
8890 161st Ave NE, Redmond

Alliance Tickets (3.08 miles)
520 112th Ave NE # 110, Bellevue

Lincoln Square Center (3.41 miles)
700 Bellevue Way Northeast, Bellevue

Lincoln Square (3.41 miles)
700 Bellevue Way Northeast, Bellevue

Kirkland Parkplace (3.49 miles)
401 Parkplace, Suite 105 Kirkland, WA 98033

Figure 1: Two app pages from a mobile app. Clicking on the `List Item` 'Cinema' on the first page leads to the second page.

well automated execution techniques work on many third party apps from app stores. We fill this void. To the best of our knowledge, this is the first large study of this kind. Our results show that fully automated execution can achieve good coverage ($> 50\%$) for almost half the apps. We survey various reasons behind poor coverage of remaining apps—some of the reasons are fundamentally hard to address without putting human in the loop. Our results also show that our optimizations make SPADE $3.5\times$ faster than the baseline implementation, enabling it to dig $3,000$ apps per day on a single phone emulator running on a desktop-class computer.

2 Background

SPADE's design is guided by characteristics of mobile apps, their UI structures, contents, and execution model.

Mobile Apps. Today, popular mobile platforms maintain centralized app stores through which apps are distributed to mobile users. There are over a million apps in major app stores written by third-party developers. Developers build apps and submit only the app binaries to the store which are then made available for download. For digging apps, *app stores have to deal with only app binaries, without any access to their source code or any help from developers.*

UI Structure. Mobile apps typically display contents organized as a set of *App Pages* that users can interact with and navigate between. Each app page usually takes the entire screen real estate. An app page can have a set of UI controls such as textboxes, buttons, lists, images, etc. Certain UI controls (for e.g. buttons) are *interactable* as opposed to controls that just display content. A user can navigate between various app pages by interacting with these controls or by using certain gestures such as swiping across the screen. Some phones also provide a physical *back button* that the user can press to go to the previously visited page.

Figure 1 shows an example with two app pages from a mobile app that lets users search for nearby businesses. The first page shows a `List` of business categories, where individual `List Items` are interactable (i.e., clickable with a touch). When a user clicks on a `List Item` (e.g. Cinema category), he navigates to the second app page that shows a `List` of nearby businesses for that category. The user can further click these `List Items` to see the details of the business.

App pages are typically designed as *templates* which are instantiated at runtime with dynamic contents. A typical app can have up to tens of page templates and up to hundreds of page instances. In the example above, the second page is designed as a template which is instantiated with different content for each business category.

An app page with a set of UI controls can be represented as a *UI tree* where each node represents an *UI element*. A UI element can be a *container element* that can recursively contain more UI elements. Smartphone runtime provides a set of standard UI controls with well defined behavior. However, app developers can define new UI controls (often derived from the base controls). *An app digging system needs to automatically navigate between app pages by understanding its UI controls.*

App Data. We use the term *App Data* to denote contents that an app presents to the user during runtime. Most smartphone apps heavily interact with the cloud and show dynamic app data to users. Some apps also get app data from local resource files embedded within the app. App data downloaded from the cloud can change based on time, user's location, user input, or other sensor data. Therefore some applications such as the *App Crawler*, which needs to dig as much recent data as possible, *need to dig apps repeatedly, possibly with multiple locations and a variety of user inputs and sensor data.*

Execution model. Mobile apps are event-driven programs where UI controls register *event handler* functions to be invoked on various *events*. For example, when a user clicks on a button, the registered click handlers are invoked. App developers typically include complex logic in the event handlers to render a page. The asynchronous nature of apps [18] increases the complexity of execution even further. For instance, an app can execute a number of parallel threads and asynchronous calls to individually render different controls in a same page [18]. So, *the digging system needs to carefully keep track of the app execution to efficiently navigate it*.

3 SPADE Design

3.1 Goals and Design Choices

We have the following set of design goals for SPADE:

G1 Given an app binary, automatically launch the app and navigate through its app pages, without access to the app source code or any human in the loop.

G2 Perform certain tasks while navigating through app pages. The task is application specific. For instance, the task of the *App Crawler* is to log all data shown to user, for later indexing.

G3 Do the above very fast. This is crucial for saving time and resources. Typical app digging applications run on a large number of apps. Moreover, some applications need to be run repeatedly, on a routine basis. Making the applications run as fast as possible saves valuable resources.

Before we delve into the details of the SPADE architecture, we briefly discuss some of design choices.

Device vs. emulator. We execute apps in a phone emulator instead of real devices. All major mobile SDKs come with emulators to enable developers quickly test their apps without deploying on a device. Compared to running apps on real devices, running on the emulator is faster and it allows us to scale to many concurrent executions. It also simplifies interaction with the apps and enables various optimizations as we show in the following sections.

Tight-coupling vs. loose-coupling with the emulator. There are two approaches to build SPADE using the emulator. One approach is *tight-coupling* with the system software - i.e., to modify the phone OS and emulator to make SPADE part of it. The alternative approach is *loose-coupling*, i.e., to keep the system software and the emulator unmodified and build SPADE as an external entity. We opt for the latter approach. SPADE achieves loose-coupling by instrumenting only the app binary (as opposed to system software) and using an external program that can automatically navigate the instrumented app inside the emulator.

Our rationales for loose-coupling are as follows: First, compared to the tight coupling approach, we do not need to modify the system software. Smartphone system software is complex and diverse, making tight coupling challenging and difficult to debug. Moreover, some system softwares are not available for modification (e.g., OSes other than Android). Internals of system software are subject to frequent changes, even though the interfaces do not change. Thus, a tightly coupled solution may require expensive maintenance. Second, a loosely-coupled solution offers better modularity and portability, enabling our infrastructure and the emulator to evolve independently and be distributed easily. Finally, a loosely-coupled solution can run apps more reliably on the same system software developers tested their apps (instead of a custom version that integrates SPADE into it).

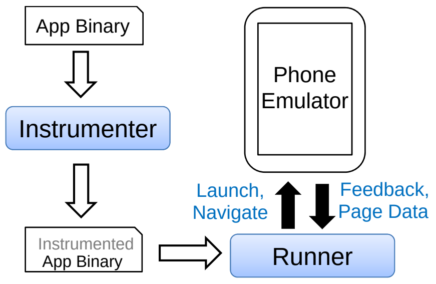


Figure 2: SPADE architecture.

3.2 SPADE Architecture

Figure 2 shows the overall architecture of SPADE. SPADE consists of two main components. (1) the *Instrumenter* which instruments the app binary and (2) the *Runner* which automatically navigates the instrumented app in the emulator.

Instrumenter. In SPADE, we only need the app binary. We do not need access to the source code of the app or any help from the developer. This lets us work with any app that is already in the app store.

However, SPADE needs visibility into app's runtime state/behavior and communication between the loosely-coupled Runner and the phone emulator. This is because: (1) the Runner needs to inspect the current app page to decide the next interaction, and (2) app digging applications need to capture/analyze app's runtime state/behaviors.

The Instrumenter instruments an app binary to achieve the above. Given an app binary, it uses binary instrumentation to add various pieces of code that we discuss in Section 4. We consider Windows Phone apps and our binary instrumentation framework is based on Microsoft's Common Compiler Infrastructure [4] and similar to the one in [18]. The current implementation is designed for apps written using the Silverlight framework [17], compiled to MSIL [13] byte code. Silverlight is used by a vast majority of the apps in the Windows Phone app store. The instrumentation requires no support from the Silverlight framework or the OS.

Runner. The Runner is an external program that automatically launches and drives the instrumented app in the emulator. The Runner automatically launches the phone emulator, installs a given instrumented app, and launches the installed app. It then automatically controls mouse events on the host machine to drive the app in the emulator as if a user on the machine is driving it.

The Runner can initiate different mouse events that map to all gestures that a user could do on a real device such as click, flick, swipe, drag, pinch, zoom etc. It can also click on physical buttons (e.g. the back button of the emulator) and can provide keyboard inputs to textbox controls inside the emulator.

The Runner coordinates with the instrumented app to automatically drive it. The instrumented app constantly communicates the execution state and the UI structure of the current app page to the Runner so that the Runner can make informed decisions on how to drive the app. It identifies interactable controls on a page and interacts only with those controls (as opposed to randomly interacting with the app). It maps coordinates of UI elements on an app page to coordinates on the host screen to initiate mouse events on appropriate UI controls.

The Runner also has the ability to feed different sensor and I/O inputs during app execution. For example, the Runner can run a location-aware app with different location inputs, to crawl its contents at various locations. We will explain this in Section 4.3.

The Runner maintains a history of previously visited app pages. By analyzing the history, it can prioritize navigation to *important* pages, in order to achieve better coverage in a limited time. We will explain some of our heuristics in Section 5.

Finally, the Runner also incorporates application-specific logic. For most applications, it helps store the data transmitted by the instrumented app for offline analysis. For some applications, it analyzes the data online and uses it as a feedback to drive the navigation.

4 Binary Instrumentation

The Instrumenter injects code[1] into a given app binary to do the following things. (1) Find if the app has finished processing an UI interaction by automatically tracking asynchronous calls and thread execution (§4.1). This helps the Runner navigate the app faster without having arbitrary timeouts between UI interactions. (2) Inspect the app's current UI structure, find interactable controls and inform the Runner as the app is being navigated (§4.2). (3) Interpose between the app and the framework libraries so that Runner can control certain inputs to the app (§4.3). For example, the Instrumenter rewrites calls to the location API so that the Runner can feed in different locations to the app. (4) Do application-specific tasks (§6). For instance, the App Crawler needs to capture page contents as the app is being navigated and store them for offline indexing.

In addition, we sanitize the app to remove tasks such as maps, email, sms etc. that might take the Runner outside the app to external programs. This helps us keep the navigation within the app boundary.

4.1 Done with processing?

One key piece of information the Runner needs to know while automatically interacting with an app is whether

[1]The Instrumenter injects code at the MSIL level.

the processing for an outstanding UI interaction is complete. For example, in Figure 1, after clicking on the 'Cinema' category, the Runner needs to know when all the processing—navigation to the new page, download of data, rendering on the new page, etc.—is completed.

The information is crucial for two key reasons. First, after each interaction from the Runner, the instrumented app inspects the resulted page and communicates UI controls in the new page to the Runner so that the Runner can choose the next UI control to interact with. The app has to communicate this information only **after** the page has reached a stable state, i.e., after all the outstanding operations initiated by the interaction has completed. Otherwise, the UI controls communicated to the Runner could be in transient states. If this happens, by the time the Runner does the next interaction, the page could have changed and the interaction would not be as expected. This can affect the correctness of the digging operation. Second, some digging applications need to capture runtime state of apps; and doing this makes sense only after the page has been loaded and reached a stable state.

One straightforward solution to address the above concern is to make the Runner wait for some time between successive interactions. The wait time needs to be long enough to ensure that the processing of an interaction is completed and it is *safe* to initiate the next interaction. As we show in Section 7, processing time of an interaction can be highly variable—from a few milliseconds to a few tens of seconds! Hence, the timeout value has to large enough (5.5 seconds at the 95% percentile) to ensure correctness for most of the interactions.

Since one of our primary goals is speed, we cannot afford such large timeouts. We cannot sacrifice correctness either, by using a small wait time. Hence, we need to know exactly when the processing for a UI interaction is complete so that the Runner can immediately perform the next interaction.

► Limitation of `Page Loaded` event. Platforms such as Windows Phone and Android do provide a `Page Loaded` event. But those events only indicate that the navigation to a page is complete—it does not account for the fact that the app can initiate asynchronous processing to further update controls in a page. In fact, processing in mobile apps are highly asynchronous [18].

We illustrate this with an example in Figure 3 that shows the app execution for the example in Figure 1. The thick horizontal lines represent that a thread is executing and the dotted lines link asynchronous calls to their corresponding callbacks [18]. In this example, when the Runner clicks on a list item, (1) it calls a click handler method inside the app (2) which in turn makes an asynchronous call to navigate to the next page (for e.g. nearby businesses page). (3) Once the navigation is complete,

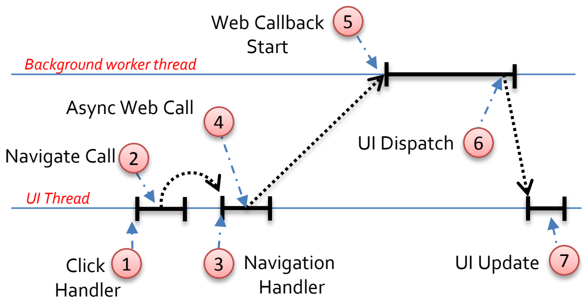


Figure 3: A sample execution trace of an app when navigating a page. After the page is navigated, it makes an asynchronous call to the web, processes the response and updates the UI asynchronously.

the system calls a navigation complete handler (this is essentially the page loaded event that most mobile platforms provide). (4) In the navigation handler, the developer makes an asynchronous call to the web to download the list of businesses. (5) Once the data is downloaded, the system invokes a callback on a background thread. This thread processes the data and (6) initiates a dispatcher call to (7) update the UI (the list control) on the UI thread.

As evident from this example, a page can be updated asynchronously and hence we cannot rely on the `Page Loaded` event provided by the platform. Moreover, certain UI interactions can modify only controls on the current page through an asynchronous call without navigating to another page—`Page Loaded` event will not be fired in such cases.

►Our solution: `Processing Complete` event. The execution graph in Figure 3 represents a transaction [18] initiated by the UI interaction. Our aim here is to know when the entire processing of the transaction is complete so that we are sure that the UI won't be updated anymore. We achieve this by automatically monitoring the app execution and providing a reliable `Processing Complete` event for every interaction. The app inspects the UI structure and informs the Runner only when this event is fired.

Our technique to generate a `Processing Complete` event stems from the following observations on the transaction graph. The transaction graph is a connected graph with two types of edges: (1) the thread execution edges that connect the start and the end of a thread execution in the app (the thick horizontal edges) and (2) asynchronous edges that connect the asynchronous calls to their respective callbacks (the dotted lines). When the execution of a transaction is active, at least one of these edges is active (i.e., either an asynchronous call is waiting for a callback or a thread is executing). When the transaction completes, all the edges have run to completion.

So, our idea here is to monitor these edges and keep track of the outstanding edges at any point in time during execution. We raise the `Processing Complete` event when there are no more outstanding edges for the corresponding transaction. To monitor these edges, we borrow the instrumentation techniques from [18]. But the key difference here is that, we need to keep track of the outstanding edges online during execution instead of writing logs for offline analysis.

Tracking thread execution. We use the same heuristics as in [18] to identify Upcalls and instrument its start and return points. Upcalls are calls made by the platform into the app (event handlers and asynchronous callbacks are Upcalls). Tracking the execution of Upcalls essentially tracks the execution of threads inside the app. When the instrumentation point at the start of the Upcall is hit, we add an outstanding edge. The edge is cleared when the execution hits the return instrumentation point. We match the Upcall start and end using dynamically generated method ids.

Tracking asynchronous calls. We identify asynchronous calls and instrument them. We also detour the callbacks [18] so that we can match the callback to its respective async call. When the app makes a async call, we add an outstanding edge. The edge is cleared when its callback is fired.

We ignore timer calls that provide periodic async callbacks from the list of outstanding async edges. Timers are typically used to do periodic tasks. We do add the callback execution of the timer as an outstanding edge because it might be used to update an UI periodically. During such scenarios, we would be generating `Processing Complete` event for every periodic update and informing the Runner about the updated UI. Hence, the Runner can always make the right decision.

We also appropriately take care of thread synchronization calls that could block a thread indefinitely until a semaphore is fired. To handle these cases, we segment the thread execution edges at these blocking calls. We don't get into the details due to lack of space.

As we show in Section 7, our technique to generate a `Processing Complete` event is reliable and helps in speeding up the Runner.

4.2 Inspecting the UI Structure

The Instrumenter injects code such that, every time a `Processing Complete` event is fired, it inspects the current page. It obtains handle to the current page and traverses the UI tree to inspect each of the UI elements on the page. For each UI element, it obtains the element's properties such as size, position, name and the content displayed by it. It then sends the serialized information to the Runner so that it can initiate its next interaction on an appropriate UI control.

Before sending UI tree information to the Runner, the instrumented app adds two extra pieces of information to each UI element in the tree. (1) a unique identifier, (2) a flag indicating whether the UI element is interactable and what interactions can be made on it.

Unique identifiers for UI elements. Each UI element needs an identifier that is unique within and across runs. Uniqueness within the same run helps the Runner avoid possible navigation loops. Uniqueness across multiple runs helps the Runner build a history of UI and its interactions, which we use to optimize Runner's navigation decisions for improved coverage (details in Section 5). Thus the identifier should be independent of the UI element's dynamic properties such as its object id, screen location, etc. We generate an identifier as a hash of the following static properties: the type of the UI element, its level in the UI tree, the identifier of the parent node and properties such as name and content.

Identifying interactable controls. A UI element can either be interactable or non-interactable. Typically, label controls that display content to the user are not interactable (clicking on them will not take you anywhere). Hence the Runner should not waste time interacting with them. Controls such as buttons are interactable. The Runner has to identify and interact with them.

When we traverse the UI tree, we identify if each UI element is interactable or not. And if it is interactable, we identify the types of interactions that can be made on the element. This is one of the key features of our system that helps the Runner achieve high coverage quickly.

Our technique is based on the following observation. In event-driven programs, the developer needs to add event handlers to get callbacks when a interaction happens and processing is done on those callbacks. Hence, if no event handler is added to a control by the developer, no app processing happens. We assume that such controls are non-interactable. For example, even if you add a button to the UI but do not add a click handler to it, interacting with the button is not useful as it does not trigger any processing. Similarly, we assume that controls that have at least one manipulation event handler added to it as an interactable control. Further, by looking at the types of event handlers, we are able identify the types of interactions that can be done with the control.

Since we have handle to each UI control during inspection, ideally, we should be able to query the UI control's property to identify the event handlers associated with it. But, the Windows Phone framework restricts read access to these properties at the app level (note that our instrumented code runs at the app level). To circumvent this problem, we instrumented all the API calls in the code that adds and removes event handlers to controls. When those calls are executed, we record the control and the type of the event handler[2]. When inspecting the UI controls, we lookup the event handlers associated with them and communicate it to the Runner.

4.3 Controlling the Inputs

We control two kinds of inputs to the app: (1) Location (2) Data from the network. We achieve this by rewriting calls in the app code to those framework APIs.

GPS. There are many location-based apps that change content or behavior based on user's location. So, we need to execute them with different location inputs to capture their content/behavior at different locations. To achieve that, we replace app's calls to system's location API with calls to the Runner, which provides the location data. This way, the Runner can run an app for any arbitrary location.

To decide whether the Runner needs to execute an app for multiple locations, we first statically analyze the app binary to check if it calls the location API. If it does, we use the following heuristic to determine the granularity of locations (e.g., zip code, city, state, or country) the app is sensitive to. We run the app multiple times with locations of multiple near-by zip codes in the same city and check if the content of the app changes for various zip codes. If not, we try the same for multiple near by cities in the same state and so on. We label the app with the largest granularity of location for which its contents change and use the granularity in subsequent runs.

Network data. We co-locate a HTTP network proxy with the Runner and make the Instrumenter rewrite all HTTP web calls in the app to go through the proxy. We use the proxy to speed up the navigation by providing cached data to the app instead of fetching it from the network. We explain this in more detail in Section 5.2.

5 Optimizations for Repeated Digging

Some applications need to dig apps repeatedly, on a routine basis or on demand. For example, App Crawler needs to be run frequently, for different locations, to dig the most recent app data. Ad Fraud Detector needs to be run repeatedly as new fraud signatures are discovered. In this section, we describe a few optimizations that can further expedite such repeated app digging applications.

5.1 History-based Prioritization

SPADE offers a *quick digging* mode where SPADE is given a time limit for digging a collection of apps, and the goal is to maximize the app digging utility within the time limit. Quick digging is useful when there is not much time or resource to perform a *deep digging*, where the goal is to explore all app pages by interacting with all interactable UI controls, without any restriction on time. For instance, an application can perform deep digging once a week and quick digging once every day.

Since quick digging may explore only part of an app, the key goal is to prioritize digging "important" apps and "important" parts of the apps to maximize some application-specific utility. For applications whose key goal is to extract as much app data as possible, a utility metric can be the number of new (with respect to the previous crawls) words/phrases extracted per second. From historical crawled data, SPADE maintains relative utility scores of (1) various apps (apps that refresh data frequently have more utility), (2) various locations for a given location-aware app (locations for which the app contains more data have more utility), and (3) various UI controls of a given app and a given location (UI controls that lead to more content-rich pages have higher utility).

Given these utility scores, SPADE can prioritize its selection for apps, locations, and UI controls with a simple greedy algorithm. Intuitively, the algorithm allocates the total time for all apps among various apps and various locations, in proportional to their utility values. After SPADE has decided how much time to spend for digging an app, it explores various pages by interacting with UI controls in decreasing order of their utility values. After the app has been digged for its allotted time, SPADE moves to the next app.

5.2 Aggressive Network Prefetching

Mobile phone OSes and apps use various conservative network optimizations such as batching, lazy download, etc. to save energy. Since we run SPADE in a desktop machine, we override these energy settings for improved performance. In particular, in repeated digging, SPADE aggressively prefetches network data before an app even needs it. SPADE runs a local network proxy and reroutes all HTTP web calls from of an app through the proxy. The proxy predicts and prefetches HTTP resources so that most HTTP requests are served from the local proxy, saving a round-trip to the cloud.

Predicting urls to prefetch. Predicting urls to prefetch is a challenging problem in general. Fortunately, in SPADE, the Runner explores various app pages and downloads various urls in the pages systematically and their sequence remains the same across repeated digging sessions. Hence, SPADE can exploit history of past digging sessions of the same app to predict what urls the app will fetch in near future and prefetch them in the proxy.

One challenge comes from the fact that, in many apps, urls change across sessions and hence urls learned from past sessions may not be valid for the current session. In many such cases, urls typically follow simple patterns. This is because most apps use RESTful web services and

[2]In Windows Phone framework, event handlers can also be added through XAML files. We identify them by statically analyzing the app.

HTTP GET requests, where the url encodes all necessary parameters to uniquely identify the resource. Only some parameters (e.g., session id, user id, location, etc.) of the urls change across sessions.

SPADE exploits this regularity as follows. First, by observing the sequence of urls for the same UI control, SPADE learns *url templates*, that keep the base urls, static parameters with values, and dynamic parameters without values. Second, during a digging session, it infers values of dynamic parameters in url templates by comparing templates with concrete urls. Once the value of a dynamic parameter is inferred, it can be used in all url templates in the current session to generate concrete urls to prefetch. We omit the details for lack of space.

6 Applications

We here describe three applications that we have built on SPADE for the Windows Phone app store. We have evaluated utility of these applications on 5,300 randomly chosen apps from the app store; for lack of space, we report only the highlights of the results.

6.1 App Crawler

App stores' search engines index various app metadata such as app name, category, and developer-provided description; unlike web search engines such as Bing and Google that index webpage content, they do not index the app content (i.e., data apps show to user during run time, possibly by downloading from the Internet). Thus, a search query can return an empty result if the query does not match any app metadata, even though the query might match app data of some app. For example, a search for "mushroom risotto" returns no results in Windows Phone app store, even though several apps (e.g., `BigOven`) show its recipes to users. The situation is grossly similar in Android and iOS app store as well. The goal of the App Crawler is to crawl app data for indexing them in the app store search engine.

We build this application on SPADE as follows. First, we give *crawling code* to the Instrumenter to inject it within the app binary. The crawling code, after the `Processing Complete` event is fired, traverses through the UI tree of the current app page and logs all textual data in all UI elements. The data is then sent back to the Runner. Second, we write *logging code* in the Runner that logs all data sent by the crawling code in a local database. After SPADE has processed all apps, we index the crawled app data, along with app metadata, with Apache Lucene[3], an open source text indexing tool.

Results. As mentioned, app stores' default search engines index app metadata alone. To illustrate the utility of crawled app data, we compare App Crawler's index over app data + metadata with an index over app metadata alone. We then take a 4 month long trace of queries in Microsoft (app) Store and replay it on both indexes. We found that index over app data + metadata satisfies around 25% more queries than the index over metadata alone.

[3]`http://lucene.apache.org`

Table 1: Various types of ad frauds and example Windows Phone apps detected by using SPADE

Fraud Description	Apps
Create **multiple** ($n > 1$) ad control objects within a page. (*To claim $n\times$ more ad impressions*)	`BatterySaver` (+32 apps)
Make ad control **too small**, compared to its standard size	`UnitConverter` (+9 apps)
Place ad controls **outside screen**. (*To accommodate more ad controls per page or to create an "ad-free" illusion*)	`PhotoEffect` (+47 apps)
Hide ads behind other controls. **Overlap** ads with clickable controls. (*To hide ads or to get inadvertently clicks. Developers make more money for clicks than impressions.*)	`LogicGames` (+17 apps)

6.2 Ad Fraud Detector

Ad-supported mobile apps include *ad controls* that fetch ads from backend ad networks and show them to users when they visit particular app pages. The first page in Figure 1 shows an ad control at the bottom of the page.

Many ad networks, including the ad network of Microsoft Ad Control used by more than 75% of Windows Phone apps, pay app developers based on impression count. To maximize the effect of ads, ad networks want apps to follow certain policies. For example, Microsoft Ad Control expects (1) an app to show at most one ad at a time per page, (2) the ad to be shown in its default size, and most importantly, (3) the ad to be clearly visible to users. However, some developers deviate from these guidelines for making more money. Few examples of such fraudulent behavior are shown in Table 1. Currently, Windows Phone app store uses various mostly-manual, ad hoc techniques to detect such fraudulent behaviors. We use SPADE to automate the process.

The Instrumenter for this application injects code that, after the `Processing Complete` event is fired, inspects all ad controls in the current page. In particular, it counts the ad controls (to capture multiple ads), checks their x,y coordinates and sizes (to capture too small and out-of-screen ads), and compares their positions and sizes with those of other UI controls in the page (to capture overlapping and hidden ads). Once a possible fraud is detected, the code also takes a screenshot of the screen (as a proof) and sends it, along with the fraud information, to the Runner for archiving in a local database.

Results. Not all apps in our app set are ad supported.

We picked 353 top Microsoft Ad Control-supported apps and ran them through the Ad Fraud Detector. We discovered a total 80 apps with fraudulent behaviors. Table 1 shows several example apps that we found to have the fraudulent behaviors. Some of the apps have multiple types of frauds. These apps had been in the Windows Phone app store for over 1.5 years, but were not detected for these frauds. We have reported our findings to Microsoft Online Forensics team for further actions.

6.3 Ad Keyword Extractor

Contextual advertising, where ads in a page are chosen based on the content of the page, is common and effective in the Web (e.g., Google AdSense). It works as follows: the ad network crawls webpages offline and label them with important ad keywords extracted from the pages. When a user visits a webpage, the ad network shows an ad whose bidding keywords match the ad keywords in the page.

The process does not trivially translate to mobile apps since ad networks cannot crawl app data (without a SPADE-like tool). We address this by digging apps with SPADE. Instrumentation injects code similar to that for app crawling, but in addition to capturing all words/phrases in the page, the code also logs their display attributes such as their frequencies, positions, font sizes, capitalizations, etc. The display attributes are important for keyword extraction algorithms. The crawled words/phrases along with their attributes are then sent to the Runner, who logs it in a local database. As data for each page arrives at the Runner, it is fed into KEX [20], a well known keyword extraction tool, which returns a list of ad keywords with their significance scores. Finally, each app page is tagged with prominent ad keywords in that page, to be used for selecting contextually relevant ads during run time.

Results. We evaluated the quality of ad keywords extracted from the above application with a 1-week long trace of bidding keywords from Microsoft ad network. Our results show that (1) page data of half the apps contain more than 20 ad keywords that could be targeted with contextual ads in Microsoft ad network, and (2) For $> 85\%$ apps, page data contains more ad keywords than app metadata (e.g., description on app store). Thus, app data can be a potential goldmine of keywords for effectively matching contextual ads.

7 Evaluation

In this section we evaluate SPADE's coverage and performance with 5,300 randomly chosen Silverlight apps from the Windows Phone app store.

7.1 Coverage Methodology

Most app digging applications deal with app pages, hence we evaluate how good SPADE is at visiting various pages of an app. We quantify this with *digging coverage* metrics.

As mentioned in Section 2, a typical app has up to few tens of *page templates*, each of which can be instantiated multiple times with different data, resulting in up to few hundreds of different *page instances*. Some app digging applications (e.g., the Ad Fraud Detector) care about certain properties (e.g., ad control) of page templates only. Some other applications (e.g., App Crawler) care about extracting as much data as possible from various page instances. We therefore use two digging coverage metrics: *page template coverage* and *page instance coverage*, defined as the fraction of page templates and page instances, respectively, explored by SPADE.

7.1.1 Template Coverage Results

To compute template coverage of SPADE for a given app, we count the number of page templates explored by SPADE and divide that by the actual number of page templates, determine by static analysis of the app.

Difficulty in achieving good coverage. Based on our experiments on all 5,300 apps, we find that achieving a good coverage is difficult with SPADE's fully automated app execution.
► **R1:** Some apps depend on text inputs from users on its start page. The inputs include username/password (e.g., Facebook app) or search keywords (e.g., Wikipedia app). At least 1% apps in our app set are of this flavor.
► **R2:** Some apps create custom controls that the Runner has no knowledge about. For example, a category of eBook apps implemented a custom `ScrollMotion` control, which SPADE failed to interact with.
► **R3:** Some apps require the user to interact in a certain way. For example, the popular Wordament app instructs the user to swipe finger over the letters 'P' 'L' 'A' 'Y' on the start screen to go the next page. An automated Runner cannot understand such instructions.
► **R4:** We configured SPADE to wait for 10 seconds for an app to load in the emulator; a small fraction of apps (e.g., AlQuran, 120MB in size) could not finish loading in this time.
► **R5:** At least 5% of the apps crashed while running in the emulator due to their own flaws.
► **R6:** A small fraction of apps crashed due to our instrumentation, often in the first one or two pages. E.g., Instrumenter removes ad control during instrumentation, but some apps wants to manipulate ad control during run time and hence fail.

We believe that some of the above factors (especially, R1, R2, and R3) are fundamentally hard to address with-

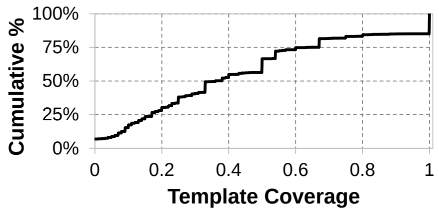


Figure 4: CDF of template coverage, for all 5,300 apps.

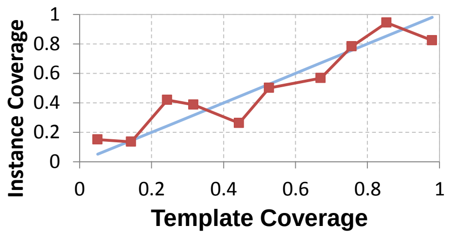


Figure 5: Instance coverages for apps with various template coverages.

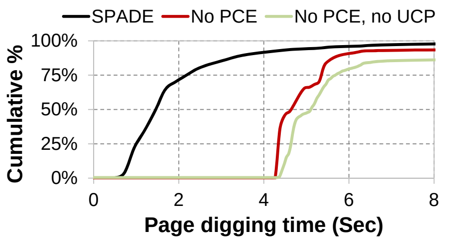


Figure 6: CDF of page digging time, with and without optimizations.

out involving human in the loop. If our goal were to automatically execute a handful of apps (like previous studies [1, 2]), we could have addressed them, e.g., hard coding username/password (to address R1) and including custom logic for interaction (to address R2 and R3). However, this is not a scalable solution when we intend to deal with thousands of apps. So we keep this study limited to understanding how much fully automated execution can achieve.

Coverage numbers. Figure 4 shows the CDF of template coverage of all 5,300 apps. SPADE achieves 100% coverage for roughly 15% of the apps and more than 50% coverage for roughly half (48%) the apps. For the other half, the coverage is poor due to the aforementioned reasons (R1 to R6). The average, median, and standard deviation of coverage is 0.46, 0.40, and 0.3 respectively. For 3.8% of all apps, at least one of the above reasons (R1 to R6) happened before SPADE could explore the first page, and hence SPADE got a zero coverage. We ignore these apps for rest of the study.

Possible solutions to improve coverage. It is possible to improve the above coverage results. Addressing R4, R5, and R6 is easier: R4 could be addressed by increasing the wait time, R5 by rerunning the crashed apps and hoping that their crashes were transient, and R6 by further tuning our Instrumenter for those apps. Addressing R1, R2, R3, however, would need to include humans in the loop. To enable this, we plan to incorporate an *interaction record* feature in SPADE. This feature requires an app to be run manually once in SPADE (possibly through crowdsourcing). SPADE records all user inputs and interactions and augments that with its automated execution to explore app pages that it cannot explore without human help. Exploring this is part of our future work.

7.1.2 Instance Coverage Results

Page instances are created during runtime. So, to compute page instance coverage, we manually execute the apps to count their actual page instances. Since the process does not scale, we limit our study to a smaller set of 30 apps. In particular, we divide all 5,300 apps into ten buckets: 1st bucket has all apps with template coverage between 0-10% (based on previous experiment), 2nd bucket has all apps with coverage 10-20%, and so on. We then pick three random apps from each bucket and compute their average instance coverages and average template coverages.

Figure 5 shows average instance coverages for various template coverages. We see that instance coverages roughly follow template coverages. If this holds for all other apps, the CDF of instance coverage would roughly resemble the CDF of template coverage in Figure 4. This is expected because if SPADE can visit at least one instance of a page template, it most likely can visit all instances of the template. The only way instance coverage can differ from template coverage is when there are variable number of instances for various page templates. For example, consider an app with two page templates P1 and P2, only one of which is visited by SPADE. If P1 has 1 instance and P2 has 99 instances, instance coverage will be 1% or 99% depending on which pages SPADE visits, even though template coverage is 50%. With many apps, the average variability would become small.

7.2 Performance Results

We now report how fast SPADE can dig various apps and the effects of various optimizations we implement. We use three desktop machines, each with Intel Core2 CPU (2-4 cores), 4 GB memory and 512 GB hard drive.

We report the metric *digging time* of a page, which is the time interval between the Runner digging two pages. More specifically, digging time includes the times required for (T1) a page to be completely loaded (i.e., the `Processing Complete` event is fired) by the app after the Runner has interacted with the UI control leading to the page, (T2) traversing the UI tree of the page, (T3) sending the tree back to Runner, and (T4) the Runner to analyze the UI tree, save it to disk, and take a screenshot, before initiating the next interaction. (T1) is typically the dominating component of the digging time. For certain apps, (T2) and (T4) are non-negligible, but we believe that, those processing times can be significantly reduced by optimizing the operations we do in the critical path.

Figure 6 shows the CDF of digging time for all pages in all 5,300 apps. The average, median, and standard deviation are 2.06, 1.45, and 2.46 sec respectively. The digging time is small for most of the pages, except for a small fraction of apps that need to performing time-consuming operations in (T1), such as network calls or storage. The average number of page instances of an app in our app set is 13.32; combining that with the average page digging time implies that SPADE can dig around 3000 apps on a single phone emulator running on a desktop computer in a day.

Core Optimizations. The above speed of SPADE comes due to several performance optimizations. To illustrate their effects, we compare SPADE with two unoptimized versions: (1) SPADE without the `Processing Complete` Event (PCE)—this version uses a static timeout to wait for the processing to complete. We use a timeout value of 4 seconds, which represents the 92th percentile of page load time (Figure 6). (2) SPADE without PCE and without UI control pruning (UCP)—instead of ignoring non-interactable UI controls, the Runner interacts with all available UI controls on the current page and waits for the timeout for processing to be finished before interacting with the next UI control.

Note that in the absence of a PCE, the Runner has no way to tell whether the page has been fully loaded or not. The Runner just assumes that the page is loaded after the timeout. However, since we use a timeout value equivalent to the 92th percentile of page load time, 8% of the pages won't be fully loaded after the timeout period. On such failed interactions, the Runner simply moves on to interact with the next UI control. With no PCE, our reported dig time for a page includes all timeouts the Runner experiences due to failed interactions immediately before digging the page. (Without accumulating the timeouts, the page digging time for No PCE case would be slightly more than 4 seconds for all apps). Another side effect of using a fixed timeout is that it reduces the digging coverage—the Runner cannot dig (i) some of the 8% pages that fail to load within the timeout period plus (ii) additional pages reachable from those pages only.

Figure 6 shows the CDF of page digging times for the three versions of SPADE. As shown, the optimizations significantly improve the digging speed of SPADE. More specifically, the average (median) page digging time is 2.06 sec (1.45 sec), while it is 5.58 sec (4.62 sec) for 'No PCE' and 7.13 sec (5.12 sec) for 'No PCE + No UCP'.

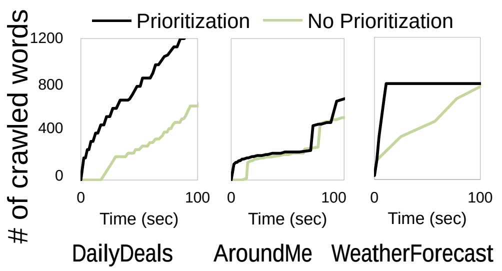


Figure 7: Crawling with prioritization, for three apps.

In other words, the two optimizations make SPADE 3.5× faster on average.

Finally, we briefly report the effects of optimizations we use for repeated execution of apps. Since the optimizations apply to only a certain types of apps, we show the results for a handful of representative apps.

History-based Prioritization. We illustrate the benefits of prioritization by using three apps of different characteristics. We consider the App Crawler application and use number of words crawled per second as the utility metric. Figure 7 shows the accumulated number of crawled words from these apps as a function of time. For example, had we stopped digging after 100 seconds, App Crawler would crawl 1517 words from `DailyDeals` with path prioritization and 628 words without it.

We selected three apps to illustrate various prioritization. In `DailyDeals`, contents change periodically and execution paths are prioritized based on past history. Major benefit comes from visiting pages with most content. In `AroundMe`, contents are location-dependent and paths are prioritized based on digging results of other locations. Major benefits come from directly going to the pages with more dynamic contents, rather than spending time on static pages (settings, terms, etc). In `WeatherForecast`, all contents are presented with the same type of UI controls (`List Items`). Major benefit comes after SPADE learns this and prioritizes List Items in subsequent digging.

Network Prefetching. We illustrate the usefulness of prefetching by using three networked apps that have slightly different properties: (1) `RageComics`, with large downloads, (2) `WeatherForecast`, with long RTT to the server, and (3) `AroundMe` with small RTT to the backend server. For these apps, our URL prediction achieves a hit ratio larger than 90%.

For content rich apps such as `RageComics`, the benefit of prefetching is significant—we saw the page digging time to reduce by up to 2.5×. Prefetching is also beneficial for apps with long RTT to back end server; for `WeatherForecast`, prefetching reduces average page digging time by 36%. We did not see significant benefit of prefetching for `AroundMe` that has small RTT to back-end server.

8 Related work

Today, mobile platforms like Android provide UI automation tools to test mobile apps. For example, Android SDK comes with a Monkey [10] and a MonkeyRunner [9] that can automatically generate UI events to interact with an app. But these tools depend on the developer to provide automation scripts. They also do not provide any visibility into the app runtime and hence are not efficient and cannot be easily used for digging applications.

Recent research efforts have built upon these tools to provide full app automation. But their main focus has been on two specific applications (1) helping developers automatically test their apps [1, 2, 7, 11, 16] and (2) automatically identifying privacy and security issues [5, 8, 19, 14]. Most systems that help developers automatically test their apps evaluate their system only on a handful of apps and many of their UI automation techniques are tuned to those apps. Systems that look for privacy and security violations execute on a large collection of apps but they only use basic UI automation techniques. Their main focus is on novel techniques to find violations and not efficient automation. None of these systems care about performance. In contrast, SPADE is designed for *performance and scale* to automatically dig *a large collection of apps* for a *variety of applications*.

Automated app execution has long been used outside mobile apps. Symbolic execution [3], which automatically runs various parts of an application with symbolic inputs, has been successfully applied to automatically generate test cases. (Similar techniques have recently been applied to mobile apps as well [2, 16].) The techniques have been shown to scale to large input domains, however, they do not produce concrete values of application's runtime states (such as texts shown to user after downloading from the Internet), which fail to achieve some key goals of digging, e.g. crawling contents. ATUSA [15] and AjaxTracker [12] automatically executes Ajax web apps for specific applications, but these techniques do not deal with mobile-specific challenges (e.g., extracting UI structure from app's runtime), do not aim scalability, and do not aim to support a wide-variety of applications.

9 Conclusion

Our long-term goal is to automatically dig hundreds of thousands of apps in an app store quickly and to get high digging coverage. While there is still a long way to go to reach this goal, our results show that our system SPADE makes a very important first step towards this goal. On 5,300 Windows Phone apps, SPADE achieves $> 50\%$ digging coverage on roughly half the apps and can dig roughly 3,000 apps in a day with a single phone emulator. Digging coverage can be significantly improved by including humans in the loop: we are currently working on extending our fully automated technique with navigation hints from humans (e.g., through crowdsourcing).

We believe that the potential of an app digging system like SPADE is huge. We demonstrated this with three novel applications and we are working on several others.

References

[1] 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.

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

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

[4] Commmon compiler infrastructure. `http://ccimetadata.codeplex.com/`.

[5] 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.

[6] S. Freierman. One million mobile apps, and counting at a fast pace. `ttp://www.nytimes.com/2011/12/12/tecnology/`one-million-apps-and-counting.html, December 2011.

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

[8] P. Gilbert, B. Chun, L. Cox, and J. Jung. Vision: automated security validation of mobile apps at app markets. In *MCS*, 2011.

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

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

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

[12] M. Lee, R. Kompella, and S. Singh. Ajaxtracker: active measurement system for high-fidelity characterization of ajax applications. In *WebApp*, 2010.

[13] S. Lidin. *Inside Microsoft .NET IL Assembler* . Microsoft Press, 2002.

[14] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou. A whitebox approach for automated security testing of android applications on the cloud. In *AST*, 2012.

[15] A. Mesbah and A. van Deursen. Invariant-based automatic testing of ajax user interfaces. In *ICSE*, 2009.

[16] 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, Nov. 2012.

[17] C. Perzold. *Microsoft Silverlight Edition: Programming Windows Phone 7*. Microsoft Press, 2010.

[18] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *OSDI*, 2012.

[19] Y. C. Vaibhav Rastogi and W. Enck. Appsplayground: Automatic security analysis of smartphone applications. In *CODASPY*, 2013.

[20] W.-t. Yih, J. Goodman, and V. R. Carvalho. Finding advertising keywords on web pages. In *WWW*, 2006.