App Crawler: Opening Mobile App Black Boxes

Suman Nath and Jie Liu

1 MOTIVATION

A June 2011 report\(^1\) showed that on average, a U.S. smartphone user spent 9\% more time using apps (on smartphone) than browsing the Internet (on PC or smartphone). This growth in app usage is fueled by an increasingly large number of diverse and useful mobile apps: there are more than a million mobile apps available today in major app markets and around 15,000 apps are released per week (compared to about 100 movies and about 250 books released per week worldwide).\(^2\)

Many of these apps such as recipe apps, news apps, deals apps, utility apps, etc. contain useful information. Unfortunately the information is hidden behind so-called “wall-gardens” that cannot be easily tapped into like other information on the Web. For example, the information inside apps is not crawlable by Web crawlers, and hence users cannot search it in existing search engines. Thus, users are relying more and more on in-app search, instead of using horizontal search engines such as Bing or Google. This is evidenced by a 2010 study\(^3\) on iPhone users that reveals that 75\% of overall mobile query volume are made inside apps. Similarly, ad systems do not have visibility into what app page contents a user is interacting with, and how, and hence they cannot show contextually relevant ads.

It is strategically important for Microsoft to find ways to get visibility inside smartphone apps and to tap into this information source. In the AppCrawler project, we take a first step in addressing several challenges in achieving this goal.

2 SCENARIOS

We envision enabling the following scenarios.

Search.
- A user searches in Bing for “how to make Windows Phone battery last longer”. Bing shows relevant results from various Windows Phone apps, including the Tips and Tricks app (along with Web results). Bing can also suggest the user to buy this paid app.
- A user searches in Windows Phone Marketplace for apps with recipes of “soupe de chataignes”. Results contain the app Italian Cooking, as it contains a recipe for it. (Currently the Marketplace returns an empty result.)
- A user subscribes to the phrase “LCD TV Deals” and receives an alert whenever any of the deals-related apps he has installed on his phone gets such a deal.

All these scenarios require crawling and indexing app content.

In-app Ad. A user, while searching for near by parking in the Around Me app, sees ads on nearby parking places and their costs. Another user checks for stock prices in the My Stock app and sees ads on investments. Showing ads based on screen content requires online scrapping of current app screen.

Currently, Microsoft Ad Control for Windows Phone is not aware of the content the user is consuming or interacting with, and hence its ads are not relevant to content. (E.g., it may show real-estate ads when the user is looking for a parking spot.)

App Development. An app developer drags and drops a “search gadget control” on his app (in a similar way he includes an ad control) and the app becomes searchable! A user of the app can now use the in-app gadget to search for any in-app content. The gadget may also point to Bing for additional results.

3 TECHNICAL CHALLENGES

Enabling above scenarios require addressing the following two key challenges.

3.1 App-page Scrapping

A typical smartphone app consists of multiple pages containing consumable information and various UI controls such as buttons, lists, swipable tabs, etc. A user interacts with the UI controls to navigate through various pages. Typical app pages contain data from two different sources (Figure 1): local resource files such as a database file or a CSV file that are installed and occasionally updated with the app, and remote data that are downloaded, typically in json format, on demand from a cloud-based service. Our study shows that almost 50\% Windows Phone apps download some data from the Internet. One might consider crawling resource files and sniffing network traffic of an app; however, this
is not very useful in practice because such data does not contain enough semantics for information extraction. For example, the *My Stock* app downloads a sequence of numbers as a json stream and it is not clear what they mean until they are presented in the UI as current prices of various stock symbols. Moreover, data can be stored locally or downloaded from remote service in compressed or encrypted form, making the task of capturing data even more difficult.

One approach we have found to avoid the above problems is to capture data at the presentation level—to scrape the content of an application page after it has been rendered completely. This allows us to capture exactly the data (and the layout) the user sees and to produce a mark up document representing the content of the page that can be used by subsequent information extraction algorithms. The solution is also agnostic of data sources and storage/communication formats of the data.

Ideally, the data scraping functionality should be implemented in the OS or in the .Net framework. For faster prototyping we have, however, taken the approach of instrumenting app binaries and running them on unmodified OS and .Net framework. Given a Windows Phone app binary, we inject logging code into it such that whenever a user navigates to an app page, the content of the page is dumped into the phone’s isolated storage as a markup document. We use Common Compiler Infrastructure (CCI) for binary instrumentation.

While the search scenarios need to scrape all app pages offline, some scenarios, such as in-app ads, require scrapping current page in an online fashion. The ad system then need to extract useful information from the scrapped page. Existing keyword extraction techniques can be a good starting point here; however, they need to be trained with sufficient amount of data crawled from apps. Moreover, since scrapping and information extraction are done online, these tasks should have small memory and CPU footprint to avoid any perceived effect on app performance.

**3.2 Automatic App Navigation**

Most applications that download data from the Internet do so only on demand: data required for a page is downloaded only when the user navigates to the page. Even when the data is read from local resource files, relevant data is rendered on screen only when the user visits a page. Therefore, data scrapping requires navigating through various app pages so that the scraper can capture on-screen data when it is shown to the user.

Capturing all data in an app requires visiting all its page. Doing this manually may be feasible for a small number of apps, but we must automate the process in order to crawl many tens of thousands of apps. The automatic process, referred as a “monkey” hereafter, will run an app and systematically go through all possible execution paths (i.e., all app pages); in parallel, the scraper tool will capture contents of all the visited app pages. Figure 2 shows the process: given the Windows Phone *Tips and Tricks* app, the monkey first launches the app. On the first screen, it discovers various clickable controls and taps/clicks on the first one (i.e., Application) to navigate to the second screen. It then clicks on the back button of the phone to come to the top screen and clicks on the second clickable control (i.e., Battery Power) and navigates to the fourth screen. There, it swipes on the screen to go to the next tab under Battery Power and continues.

Designing robust and efficient “monkey” that, given an app binary, can automatically visit all its pages by navigating through all possible execution paths is challenging for various reasons. First, the monkey needs to understand semantics of various UI controls so that it knows what actions (tap/swipe/scroll) to take on them. This can be tricky as many apps implement their own custom controls. If the monkey fails to discover some UI control or fails to act on it, it will miss navigating some execution paths. Second, exploring all pages in an app may take a long time. This is due to a combination of factors. An app may consist of a large number (e.g., up to several thousands) of unique pages each of which may

5. A monkey can be useful for automated testing of apps as well. Windows Phone team has a few of such tools. For example, Tux.Net for Windows Phone 7.5 can, given a manually defined navigation script, automatically drive an app, Hopper for Windows Phone Apollo can drive an app through a randomly selected execution path.
need to download data from the Internet. Page rendering may take nontrivial time. The monkey may need to wait for a while to make sure that the page rendering is completed. Moreover, contents of many apps are location-dependent, and hence the monkey must execute the app for various locations. Last but not the least, contents of many apps change over time (e.g., news and deals apps have new contents every day), and hence the monkey needs to run them repeatedly. All these ads overhead to have new contents every day), and hence the monkey must execute the app frequently the app should be crawled (e.g., news and deals apps are crawled every day, but movie apps are crawled for every country). For an app with dynamic content, it learns from history how often it is accessed and to navigate to all its pages. It can use various heuristics to increase crawling coverage within a given time budget. For example, it aggressively prefetches networked data to reduce page loading time and prioritizes execution paths to first explore pages with larger volume of contents. For location-aware apps, it carefully decides based on history at what granularity of location the app should be run (e.g., local apps are crawled for every city, while movie apps are crawled for every country). For an app with dynamic content, it learns from history how frequently the app should be crawled (e.g., news and deals apps are crawled every day, but movie apps are crawled once a week).

4 CURRENT STATUS AND OPEN ISSUES
We have been developing several app crawling tools for the Windows Phone platform.

- **Content scrapper**: We have implemented a lightweight content scrapper that can dump the content of a given app page as an XML file containing all texts, urls, and UI controls in the page. The scrapper can be injected into existing Marketplace apps by using binary rewriting: e.g., when a user uses our modified Groupon app, all the pages he visits are scrapped and written to the phone’s isolated storage. The scrapper can also be embedded inside a custom UI control: e.g., when a user uses an app with our custom ad control, the scrapper inside the ad control scraps all app pages the user visits and stores them to the phone’s isolated storage.

- **Monkey**: We have implemented a monkey that given a Silverlight app from the marketplace, launches it in Windows Phone Emulator and navigates through various pages of the app. We are currently implementing some of the optimizations mentioned in the previous section for faster navigation.

- **Crawler**: This is essentially a combination of the monkey and the content scrapper. Given an app from the Marketplace, we first inject our content scrapper and then use the monkey to navigate through various app pages. When the monkey is finished navigating to all pages, the isolated storage of the phone contains all the crawled data.

- **Keyword Extractor**: We have ported a keyword extractor to work on Windows Phone and trained it with app data.

- **Smart Ad Control**: We have implemented an ad control that dynamically scrapes the current app page, extracts important keywords from it, and shows contextually relevant ads. Preliminary study shows that this ad control shows much better ads for many Windows Phone apps.

There are still a large number of open issues that require significant research and engineering efforts.

**In-app navigation.** Apps assume that pages will be accessed in a sequence (e.g., a specific recipe page will be accessed only from a top level index page) and, unlike Web, they do not allow randomly accessing any given page. This can be problematic in some scenarios. For example, if search results include in-app data, the user would expect to click on a search result to navigate to the app page containing the data (similar to what he does on Web results). Without support for random page access, it is not clear where the user should be navigated to on such a click. One option could be to launch the app in an emulator and to automatically navigate to the target page (by using the monkey); however this may be a slow process and unusual user experience. Another option could be to show a screenshot of the target page and to ask user to navigate through the app himself.

We believe that the underlying app programming model and runtime should allow encoding each app page as an URL or link and using that to navigate to an arbitrary page from any other page. This will help not just the search scenarios above, but will truly open up the apps and enable in-app information discovery and indexing in a way they are done for Web pages today.

**Privacy.** Crawling application data, especially user-specific data, may raise privacy concerns. This can be partly addressed by crawling for frequent keywords, not crawling user-specific data, and not sharing crawled data across users. Understanding real privacy concerns and and devising principled solution require further research.

**Windows 8 and Apollo apps.** The potential of app crawling is likely to be even bigger for Windows 8 and Apollo apps. However, Windows 8 has a different application model than Windows Phone 7, and hence our current solution most likely will not work out of the box. We need to investigate how we can port our existing solutions and enable the scenarios mentioned before for Windows 8 and Apollo apps.