The System Design of Mobile Application Crawler and The Implementation of Some Key Technologies

Hao Li
Abstract

The System Design of Mobile Application Crawler and The Implementation of Some Key Technologies

Hao Li

Smartphones have become ubiquitous in people's daily lives, due to their pervasiveness, continuous connectivity to the Internet and significant computational power. And mobile applications have played a role as an entry point for people to access a mobile network. To help build a malicious content monitoring system and make large amount of app content publically indexed, we aimed at designing and implementing a mobile app content collection tool in this master thesis with collaboration to Tsinghua University. All key critical technical points have been discussed in this thesis report. The content crawling module has been evaluated in a open source app and the active network performance module has been evaluated on several popular apps too. The system built in this thesis project is a proof-of-concept and so future work has also been described at the end of report.
Acknowledgments

I would like to express my deepest gratitude to my supervisor Hao Yin at Tsinghua University for his patience and kindness in supporting continuously and generously guiding.

I would like to appreciate my reviewer Matteo Magnani for master thesis report and providing valuable suggestions.

I would like to thank my colleagues and friends in Tsinghua University and Uppsala Sweden for providing precious helps and suggestions.

Special thanks to my parents and all other people who gave me great support during the whole thesis process.
# Table of Contents

1 Introduction 3  
  1.1 The widespread use of mobile applications ................................. 3  
  1.2 Malicious content inside the apps .............................................. 3  
  1.3 The unknown hidden web ............................................................. 3  
  1.4 Deficiencies of existing technologies ........................................... 4  
  1.5 Outline of the thesis ................................................................. 5  

2 Related Work 5  
  2.1 Existing web crawler technologies ............................................. 5  
  2.2 Existing app indexing technologies ............................................ 6  
  2.3 Existing apps' measurement and testing methods .......................... 6  

3 System Description 8  
  3.1 Requirement Analysis ............................................................... 8  
  3.2 Crawling Strategy ................................................................. 13  
    3.2.1 Operation Recording and Replaying ..................................... 13  
    3.2.2 Activity Exploration ....................................................... 17  
  3.3 System Structure ................................................................. 20  
    3.3.1 Management & Interface .................................................... 22  
    3.3.2 Content Module .............................................................. 23  
    3.3.3 Active Performance Measurement Bed ................................. 26  
    3.3.4 Database Service ............................................................. 27  

4 Results and Conclusions 29  
  4.1 Content Crawling Result ......................................................... 29  
  4.2 Active Performance Measurement Result .................................... 29  

5 Future Work 30  
  5.1 Saving and Restoring System Status ........................................ 30  
  5.2 Simulate the Circumstance to cope with Client-based Dynamism ...... 32  
  5.3 Modeling Form Submissions to Handle Input Dynamism .................. 32  
  5.4 Continuous crawling to cope with Temporal Dynamism .................. 33  
  5.5 Expand Usage of Current Mobile Application Crawler to HTML5 App 34  

Appendices 38  

A List of Acronyms 38  

B Programming Portals 38
1 Introduction

1.1 The widespread use of mobile applications

Due to convenient features inherent in smartphones, the number of smartphone users is rapidly increasing and smartphones have become ubiquitous in people’s daily lives. People use smartphones for various purposes, for instance web browsing, sending emails and making phone calls. All these usages are done via installed mobile applications, provided by Google or third party companies. Currently the main source of obtaining mobile applications is through Google Play. By 2015, according to relevant statistics, the number of applications in Google Play has exceeded 1.8 millions. Considering other application market such as Amazon AppStore for Android, the total number of mobile applications must have far exceeded this number [1].

1.2 Malicious content inside the apps

Not all uploaded applications in app store are benign, and a few reports have already shown the presence of malicious apps in these market places. For instance, DroidDream [2] and DroidDreamLight [3] were detected from the official Android Market in May 2011. These malicious applications may leverage the vulnerability of systems to cause some irreparable damages, for instance information leakage and monetary loss [4]. Researchers in Pennsylvania State University also reported that inappropriateness of in-app advertisements gave birth to negative influence towards children mental health [5]. Consequently, scholars in Tsinghua University believe that introducing some malicious content monitoring mechanism into the app market would definitely be meaningful to society and could also introduce some deep categorization into the current app market. In addition, some malicious mobile apps not only display malicious content but also have unauthorized network-layer behaviors. For instance, DroidDream was detected from the official Android market in spring 2011 and in September 2015 over 344 IOS apps including mainstream social applications like WeChat and even some financial apps, were found infected by XCodeGhost embedded in unofficial Xcode IDE. Both infected DroidDream and XCodeGhost would send private data to unauthorized servers without users’ permissions. So a malicious app monitor system should not only focus on content monitoring, and should also trace the network behaviors of the apps. To build this system, we need to know what content this app displays to users so scholars need at first to figure out how to crawl the data from mobile applications.

1.3 The unknown hidden web

In another aspect, if we consider the web, the searchable information is only the tip of the iceberg. Due to technical limitations, quite a significant amount of data has
not been indexed by conventional search engines. To explore hidden information would for sure provide tremendous value not only for traditional search engines but also for big data analysis. So some research has been done to explore this hidden web, for instance Stanford university has developed a system called HiWE for content retrieval in the hidden web [6], and since 2013, Google has also developed a white-box technology making some activity pages searchable if developers change the app code by adding some code snippets. This technology is named App indexing [7]. By using app indexing technology, developers add several code snippets into their app program to make app content indexed by Google search engine. If Google finds that the indexed app content is searched by some users, it will add one install button beside the target result in results page, promoting users to install the app which contains the wanted content. So developers can significantly increase the install volumes of the app by using this methodology. This technology has successfully opened a window for users to access apps’ content through conventional search engines. According to Google’s statistics, an average person only uses 26% of their apps, while one of four apps would never be used at all. And Google has also claimed that since 2014 more than 100 billion deep links into apps are available today and 40% searches now return app indexing results [7]. But when we consider the huge number of mobile applications, the already indexed app content is still the tip of the iceberg, and we can not access quite a large part of apps’ source code. Consequently, how to introduce a black-box method without adding code snippets would make the remaining part of apps publicly searchable and this still remains a question.

1.4 Deficiencies of existing technologies

In current web technologies, web pages are organized in HTML form which consists of several tags and these tags could contain URL links leading to next layer’s pages. The jumping starts when users click on these target URL links and in most cases the navigation begins from a home page which is shown at the most front side of web sites. People depend on web crawlers to fetch target content from the web and store it in local databases. Web crawlers have at least one first in first out waiting queue for saving links, and each time the crawler retrieves one link from the queue and it creates mirror images of pages in local storage for further content analysis [8]. If the crawler encounters one link, this new link would be put into the end of the waiting queue as a seed. This process is supposed to terminate when the working crawler can not find more seeds from the queue or some boundary condition is met. As the web crawler is the mainstream technology for retrieving web content, when building app content retrieving system as mentioned above, we believe this tool should be some kind of new evolution of web crawlers. Or should at least have borrowed some existing useful technologies from web crawlers. In the next section, we will have a quick glance at the current web crawling technology. After reviewing current techniques, some areas where we can improve to meet the project’s requirement have been listed below.
1. Current web crawlers could only crawl the HTML pages which are viewed by browsers. Web pages displayed in PC’s and mobile browsers are written in HTML markup language. Even though currently some difficulties still exist when crawlers retrieve content from these pages, t.e.x Handling of Ajax requests, the crawling process could still be regarded as file processing. Unfortunately pages of smartphone app are not in any markup format and can in some aspect be viewed as some state transition. Existing web crawlers are not suitable for crawling smartphone app page content.

2. Current web crawlers do not have the capability to operate data processing, which means that except HTML pages they can not crawl more information such as visited IPs. When web crawlers get some page, they start processing them as a file reader. But no network interaction requests and responses are considered. As discussed above, to record network interaction data is helpful to detect malicious programs. Consequently, adding this new detection functionality is necessary.

1.5 Outline of the thesis

Based on the statements above, this thesis will focus on developing a proof-of-concept system for crawling smartphone app content without changing the app’s source code. We define the app content to be all content users can view through activities from smartphone screen and speakers, and all information we can understand from the network interactions. The system should have the ability to finish the job without adding any code snippet to the app source code, which means that the system can let a user who is not the member of the development team operate the whole process. The system should have friendly management interface, a working module and some database for storage. To verify the functionality and assess the crawling ability, we will run the system on some open source app. More advanced functionalities and further work will be proposed at the end of the report. The rest of the paper is organized as follows. Chapter 2 would introduce related work including latest web crawler technologies. Chapter 3 discusses the general design of our current system and its implementations. Chapter 4 describes the experimental results and chapter 5 discusses future work.

2 Related Work

2.1 Existing web crawler technologies

A web Crawler is a software/program or some kind of programmed script which browses the World Wide Web in a systematic and automated manner. The web has a graph structure and the links presented in the pages can be thought of as directed
edge connecting two nodes. A web crawler follows the links in this graph and as a replica of each web page it visited to store into the local repository. Web crawlers could be divided into three categories according to its crawling technologies [9]: general purpose crawler, focused crawler and distributed crawler. Current researches on web crawler mainly focus on retrieving information from hidden web [10] [11] [12] [13] and distributed crawlers [12]. Sriram Raghavan and Hector Garcia-Molina at Stanford University have designed a task-specific hidden web content retrieving framework called HiWE [6]. Researchers at Polytechnic University of New York USA have successfully built a distributed web crawler which can be scaled to hundred pages per second and is resilient to the system crashes [14]. However all of above research papers are limited in crawling the web pages.

### 2.2 Existing app indexing technologies

Mobile applications play a role as the one of main entries point for users to access to the Internet, and thus contain a vast amount of information that has not been crawled and indexed. Currently, there have been more than 80 app stores world wide and nowadays more than 1.8 millions mobile applications have been put inside the Google Play. Since 2013, Google published its unique and innovative technology to index certain activities of an app and enable normal search users to access this app activity from the search results. They called it app stream or app indexing. They claimed that quite a significant number of apps have been included by this technology. Obviously this technology has successfully expanded the indexed information boundary, but some drawbacks could not be ignored. For instance, the cooperation of developers and necessary changes on the source code are required, and the retrieved data could not be stored in other places.

### 2.3 Existing apps’ measurement and testing methods

Besides apps’ displayed content, researchers and engineers have also spent a lot effort to measure smartphone application performance and network condition, including detecting vital bugs and network workload. The application performance has a vital influence on smartphone user experience, and current industry has not worked out any effective and mature products to handle this area, so all this effort is meaningful and would definitely contribute to bridging this gap. To check whether apps could function well in the wild, researchers in Microsoft research designed AppInsight, a system that instruments mobile app binaries to automatically identify the critical path in user transactions, across asynchronous-call boundaries and multi-threaded environment [15]. AppInsight is lightweight, it does not require any input from developers and it does not require change in app source code or operating system either. AppInsight has been tested in 30 apps and has already been carried out with 30 users in field trial. Based on a study for common patterns of 70 real-world performance bugs collected from eight large-scale and popular Android applications,
researchers in China implemented a static code analyzer named PerfChecker to detect their identified performance bug patterns [16]. PerfChecker has been evaluated on 29 popular Android applications that comprise 1.1 million lines of Java code. These two mentioned toolkits are static code analysis, which aims at analyzing the application code to find out the status transition graph and other wanted information. Besides looking into the app code directly, existing measuring methods also include embedding tracing SDK in the app, which can make developers and researchers see how apps behave during the runtime environment. As typical representatives, New Relic [17], AppDynamics [18] and Dynatrace [19] provide similar solutions, embedding code snippets into source code and recording performance data afterwards. These methods have the following drawbacks. (1) **Additional overhead on smartphones:** Additional monitoring code inside an application will consume scarce resources such as battery and network bandwidth, which in turn hamper user experience. (2) **Threat of business privacy:** Application providers are generally speaking blind to actual behaviors of third party SDK or libraries. Concerning business privacy, big companies may have weak willingness to embed these third-party toolkits.

To address concerns such as app correctness, performance and security of smartphone applications, industry has introduced several mobile app testing approaches. The Monkey tool can send random event streams to an app to execute so called stress testing [20], but this limits exploration effectiveness and activity coverage. To achieve a better exploration effectiveness and page coverage, frameworks such as Monkeyrunner [21], Robotium [22], Troyd [23] and UiAutomator [24] support scripting and sending events. But scripting takes manual effort, and is not flexible during the evolution of apps and requires to be adjusted to various apps. Because these tools employ the keyword action paradigm in order to replay user-interface events. In this paradigm, input events are abstracted from the concrete GUI level to a higher-level representation that uses a GUI object’s handle or name within the system to interact with it. So if the GUI is changed, the written script should also be changed to adjust to new GUI, which makes it not flexible. To address this challenge, Tanzirul and Iulian at University of California, Riverside developed the $A^3E$ system to allow substantial Android apps to be explored systematically without requiring access to the app’s source code [25]. The insight of their approach is to use a static code analysis tool to construct a control flow graph, and then developed an exploration strategy named Targeted Exploration and Depth-first Exploration based on the graph. According to their published paper, the $A^3E$ system has be evaluated on 25 popular mobile apps including BBC News, Gas Buddy etc. In terms of the similarity to this thesis project, some key technologies could be borrowed. But $A^3E$ system focuses only on increasing activity coverage and method coverage, while ignoring the diversity of content on each activity. For instance, a ListView could display hundreds of items in its list, but only a few is displayed in the scope and the rest needs the view to be scrolled down. In this aspect, the ideal system should have some ability to intimate human scrolling the list. But the $A^3E$ system only cares if the navigation has reached the activity the ListView locates instead of how much
content can be displayed in the ListView. Moreover, the A³E system does not need to care about the diversity of page content introduced by different transition paths to the same activity. For example, if we consider coming to a product detail page, when a user clicks on a bike item, the product detail page displays the detail of the target bike while it would show the details of a car if the user clicks a car item. To tackle this challenge, the system should have some functionality to check if current page content has been duplicated in the past exploration. After careful investigation, more drawbacks of the A³E system have been found under the assumption that it is for app quality testing. When we consider crawling app content, we need something innovative.

The motivation for this thesis project is to design a content crawling tool for the mobile applications and this is not the only project researching on this area. The Reality Mining project at MIT focuses on leveraging the pervasiveness, sensing capabilities, and computational power of mobile apps to understand their social dynamics and influence [26]. Researchers can collect various types of data from different situations and model it to figure out common patterns. A few sub-projects have already collected a significant amount of data. In the badge dataset, Daniel et al. utilized some wearable badges to detect physical proximity and physical activity level et al. to understand how patterns of behaviors shape individuals and organizations, in order to predict employees’ self-assessments of job satisfaction and their own perceptions of group interaction quality [27]. Nadav et al. from MIT deployed customized mobile apps in a young family community to collecting social related data including SMS logs and credit card information et al. in order to operate so-called social fMRI to look into connection between people’s social interaction and their financial status and decision [28]. All included datasets and projects have achieved convincing results, but there are still some blank areas where this thesis project can compensate. All mentioned projects in Reality Mining mainly utilized built-in sensors to replace social self-report experiments, and have not considered what is displayed to users through screen and speakers. Consequently, in some sense this thesis project could provide some innovative data collection methods for Reality Mining or similar projects to expand the data sources.

3 System Description

3.1 Requirement Analysis

According to the OSI model made by ISO and CCITT, the network structure could be divided into seven layers, including physical layer, data-link layer, network layer, transport layer, session layer, presentation layer and application layer. When we consider a mobile app as an entry point of a group of network interaction activities, we in some sense could retrieve diverse types of data according to this OSI model. As shown in the figure 1, the content and information of a smartphone mobile application could be divided into three layers like a pyramid. The top of content
Figure 1: The content of smartphone pyramid is the displayed content that users could see from the touch screen or listen from speakers. It consists of text, image, video and audio. This layer represents the data we can retrieve from the application layer. The layer in the middle concerns the network performance, by measuring the network conditions, researchers can know metrics like transmitting speed, round-trip time and DNS resolution time. These types of data represent the data we can get from the network layer, transport layer and session layer. By decoding the IP address contained in the transmitted packets, people could figure out the exact locations of deployed servers and even topology of connected servers. This layer of pyramid could be mapped to the physical layer or data-link layer of OSI model and even some data which can represent the connection situations among different instances of OSI models in the real network. When we consider the object of this thesis project, only the following categories are considered:

1. Text
2. Image
3. Video
4. Audio
5. Network Interaction Information

To crawl this various information and store it into a relevant database requires the smartphone operating system to provide some kind of communication channel making external programs able to visit the OS or get data from the OS. But
among current mainstream smartphone operating systems, none of them has this functionality and only Android is open source so that developers could customize programming portals. Consequently, all work in the rest of the thesis report was done on the Android System. Since Android kernel code may be changed, we picked up Android-x86 virtual machine as working environment and this system can give us root priority which can accelerates our development and research.

The mobile applications running on Android according to how page elements are displayed belong to three categories:

1. Traditional mobile app, displaying its page elements as View children classes.
2. HTML5 app, written using some HTML5 framework, e.g. PhoneGap.
3. Smartphone game, which displays its page element using canvas.

To crawl all above three types of app’s information will definitely increase the complexity of the system. Since this thesis work is preliminary research on this field, the following research will focus on how to fetch content data and network requests/responses from the first category: the native app.

The studies done by Stanford University have already noted that a tremendous amount of Web content is dynamic, and these dynamic pages take a variety of forms [6]. Based on the studies done by Lawrence and Giles, close to 80% of pages are dynamically generated. When researchers at Stanford University designed their system framework to address the problem retrieving content from hidden web, they categorized the pages according to their types of dynamism and generative mechanism [6]. In comparison to the web pages, as the native mobile app pages have similar hierarchical structures and can also run client-server interaction, we borrowed the categorization method that the scholars at Stanford introduced to divided the mobile app pages into the following categories:

Categorization based on type of dynamism :

Temporal Dynamism A mobile app page containing time sensitive dynamic content exhibits temporal dynamism. For instance, an app displays the trading records of stocks and exchange rates.

Client-based Dynamism The page content depends on specific user profile and certain conditions. For instance, the app page for displaying user’s profile information or weather records of a certain city.

Input Dynamism Displayed information decided by user’s input or operations has input dynamism. For example, after choosing the leaving city and arriving city, the mobile app displays all result flights between these two cities.

Categorization based on generative mechanism : 
Static Content A page containing static content would not be changed by server’s responses and other user interactions. That means no matter how we reach this page from other activities by what operations, this page content always remains the same.

Server-side Content The server contains the code snippet that generated the page content and the activity’s displayed information would be changed or decided after some interaction with the server.

Cross-Activity Content An activity page contains content that depends on other activity’s interaction. In this sense, how a user operates on some activity would influence the other activity. For instance, the user inputs the user name at the login page and the home activity displays the user name and its relevant information.

In the final implementation of the crawling system, various dynamisms and multiple generative mechanisms would for sure introduce complexity and make the problem tougher. Temporal dynamism of pages would require the retrieving system continuously repeat the retrieving process after a certain time interval. Client-based dynamism requires the crawling process to be operated in different places and situations, or in another aspect the supporting system should simulate the circumstance. In the content retrieving framework HiWE designed by Stanford, researchers modeled the submitted forms and selected the input set which may contain most value to obtain the access to the information behind the hidden web. As for handling multiple generative mechanisms, the complicated influence source implies some kind of saving and restore of the system’s status. And this may lead to a deep research into the operating system and even the hardware. When considering the fact that this thesis work is a preliminary research of the mobile app crawler, to limit the complexity of the crawling target can make the problem more clear to address and introduce an iterative solving process. Consequently after careful investigation, the crawling target is shrinked to the following app cases:

1. **Native Android App** - The apps developed by Android SDK using Java, only where displayed activity page elements are subclasses of View class.

2. **Static Content with no Dynamism** - The targeted content should not be dependent on specific location and circumstance, and also should be isolated from manual input. In some sense, the targeted content looks the same to every user or customer.

3. **No Customized key reaction and view element** - The targeted activities should not contain any customized view element, which means that all activity page items are children classes of View. No customized key reaction means that neither `onBackPressed()` nor `onKeyDown()` is overloaded.

Android apps could accept diverse kinds of touchscreen user action inputs. Based on the survey done by researchers at University of California, Riverside and Los
Angeles [29], these available touchscreen actions could be divided into the following categories:

**Press-and-Release:** The primary touchscreen input action is the simple press, in which the user presses on a small area of touch and releases quickly. For instance, the user clicks on a button or on a hyperlink to jump to another activity.

**Press-and-Hold:** This touchscreen input action requires users to tap on a tiny screen area and keep pressing for a significant time before releasing. This action is usually used to invoke an additional operation menu for certain item. For instance, the users long click a app icon on the home desktop to remove the app or long click on a displayed image to invoke the sub-menu to save the image to local disk.

**Swipe:** This input gesture needs users to start pressing on a tiny screen area, then keep pressing to move to another position of screen before releasing. This type of input action is often required to browse a long list to move the invisible items into the visible display area.

**Zoom-and-Pinch:** This input gesture differs other ones because it requires multiple fingers to operate on the touchscreen. This gesture involves pressing and moving on touchscreen through more than one contact points. For instance, the users use two fingers to zoom on a displayed image to make it bigger to display.

To support confortable user experience and provide diverse functionalities in development for developers, Android kernel support overridden gesture callbacks to enable customizing app reactions to diverse types of touchscreen input actions. From this point of view, the user interaction of mature product apps in app store could be quite complicated. Additionally, according to researchers at the University of California, Riverside and Los Angeles [29], different types of touchscreen input actions are operated at different frequenices. We limited the production targets to only considering most commonly used touchscreen input gestures: Press-and-Release, Swipe. The recognition and processing of more complicated input actions would be left to future work.

After the analysis above, we can determine the targets for this thesis project, the type of app our current crawling system is going to crawl in this phase:

1. **Focusing on text, images and network interaction data.**
2. **Traditional native app with static page content, no content dynamism.**
3. **Only consider simple click and swipe touchscreen input action.**
3.2 Crawling Strategy

The crawling strategy contains two parts: operation recording and replaying strategy and activity traversing strategy. For several mobile apps, users need to swipe screen or click some button to enter the home activity, especially the first time users use the application. To successfully carry out the main traversing strategy, the retrieving system needs to record how operations go and replay them when necessary. Therefore the following descriptions would be divided into two parts.

3.2.1 Operation Recording and Replaying

Quite a significantly large part of mobile applications require some user operations to enter the home page during their first execution after installation. In order to successfully crawl apps, we need to let the system jump over this operation sequence automatically when real crawling begins. In addition, occasionally the crawling system needs to fast-forward the execution of certain apps in order to reach some specific app state faster than in the original execution without any additional manual input, which is called time warping. For instance, in a page with a few fragments, if the crawler requires to start crawling from a certain fragment this time warping mechanism should function to drive the app from the home page to this fragment. In another aspect, this record and replay functionality is also for automatic testing and bug reproducing. For instance, iTestin provide a desktop tool enabling users to record app operations at their own xks and then reproduce the recorded operation sequence in the cloud for compatibility testing and stress testing [XU]. In this thesis project, two different approaches have been discovered and tested, and details are described below.

Smartphone apps are distinguished in part by the expressive variety of user actions possible through touchscreens. The most common ones are described below: **Press-and-Release**, the primary input action is the simple press, in which the user taps an area on the screen and releases quickly, for instance clicking a button or typing text with the on-screen keyboard. **Press-and-Hold**, the press and hold consists of the user tapping an area on the screen and releasing after holding, in the same position, past some threshold amount of time. This action is usually used to access secondary menus or hidden options. **Swipe**, swiping on the touchscreen occurs when a user presses down on the screen from position \((x_1, y_1)\) and, while continuing to hold down on the screen, moves to a new position \((x_2, y_2)\), and releases. Swiping has been used in many apps, such as the ListView in a news app and the game Angry Bird. **Key Input**, aside from the touchscreen, users can provide manual input through physical keyboard or buttons on the phone. These key input includes home button, power on/off and volume up/down.

**Approach 1: From Android Kernel** In case of security issues, the Android system forbids third-party applications to catch all system events. For instance, to ensure all applications could exit to the Launcher activity, the home button event is blocked from all applications in the latest version of Android, even
though in early version, a running program can catch home button event from
the system log. From this perspective, to record all stimulated events, we have
to investigate the system internals. The figure 3 displays the structure layers
of Android. The applications and application framework layers are Java envi-
ronments. The libraries and Android Runtime part run in JNI environment.
Motion events and key events are delivered to the running applications. But
they are different in the ways being processed.

Initially a Linux input system has a unique file for each input device including
touch screen and every button. When a new device event occurs, the Linux
input system triggers the Input Dispatcher thread that runs on the libraries
layer to package the original input event. If the event is a motion event trig-
gerated by the touch screen, it will be directly delivered to the displayed and
focused view class to cause UI action. If the incoming event is key event, the
input dispatcher thread will pass the event to the framework layer, to the in-
put manager and window callback class for being filtered or other processing.
This is how the system works originally as the figure 4 shows.

Our change here is to modify how events are delivered. When the input dis-
patcher thread receives an event, no matter it is key event or motion event,
it will be delivered to the input manager and window callback class first. In-
side the window callback class, all coming events are saved into the Android
embedded SQLite database. The motion event is delivered to the view and
the key event is passed to certain action program. After the whole operation
process is recorded, a customized application program running at Application
layer retrieves all recorded event data and translate them into the replay script,
which in my design is the UiAutomator script.

With respect to the figure 4, below we list all program files which are required
to be changed:

**Input Dispatcher** :
- frameworks/base/services/input/InputDispatcher.h
- frameworks/base/services/input/InputDispatcher.cpp

**Input Manager** :
- frameworks/base/services/jni/com.android.server.input/InputManagerService.cpp
- frameworks/base/services/java/com/android/server/input/
  InputManagerService.java

**Window Callback** :
- frameworks/base/policy/src/com/android/internal/policy/impl/
  PhoneWindowManager.java
- frameworks/base/services/java/com/android/server/wm/InputMonitor.java
• frameworks/base/services/java/com/android/server/wm/
  WindowManagerService.java
• frameworks/base/core/java/android/view/IWindowManager.aidl
• frameworks/base/core/java/android/view/WindowManagerPolicy.java

**Content Provider**:
• frameworks/base/core/java/android/provider/EventRecords.java
• frameworks/base/packages/EventRecordProvider (new project)

**Approach 2: From Adb Debug Shell**

Adb is the main communication bridge between the Android device and external equipment. The connected external equipment can run shell commands on Android kernel through adb. The shell commands `getevent` and `sendevent` enable external programs to monitor and inject the Linux input core system events directly. Since in the final implementation, we used Android-x86 as the virtual machine and let Android run on a simulated tablet device, there is only one input source that is the touch screen of the tablet, and thus there is only one input file to check. We used the `getevent` command to collect the input events and create a replay script as bash program injecting events back through `sendevent`.

The Android software stack consists of a custom Linux kernel and libraries, the Dalvik Virtual Machine (VM), and apps running on top of the VM. When users interact with an Android app, the Android device’s sensors create and send events to the kernel via the `/dev/input/event*` device files. Each event has a standard five-field format (timestamp: device: type: code: value), where the timestamp stands for the time elapsed from the last system restart, the device names the input file that has created the input event, and the remaining fields depend on specific event information. Touchscreen gestures are encapsulated as a stream of touchscreen events as shown in the figure 2, which means that even a simple gesture could consist of several fired events. The figure 2 provides an illustration of a stream of events for a swipe action. A swipe action consists of one press, one release and several move actions. The first column represents the elapsed time, the second column represents the device file and the last three columns represent the exact values for events, using hexadecimal system if no readable labels are added. Once one coordinate is changed, one input event is fired and then another synchronization event is triggered to report the updated value to the upper system. The reported Linux input events are capsulated into motion events or key events as described in the mentioned first approach. To replay the events, we can use the `sendevent` command and a sample code snippet is shown below:

Listing 1: Sendevent bash code snippet

```bash
#!/bin/bash
sendevent /dev/input/event1: 0003 0000 0000015e
```
These two approaches all have been implemented and tested in the experiment, and both of them could smoothly record how a user operated the target mobile application and replay it when necessary. In comparison, the former method which modifies the Android kernel source code can lead to more flexibility, because the events stored in SQLite database have already been packaged and can easily be translated into different types of replay script, for instance UiAutomator or Appium automatic UI testing framework. Another advantage of this approach is that each event has been packaged so that thus we have avoided dealing with tricky hardware issue and differences on infrastructure system. However this method also contains drawbacks: it slows down the running and reaction speed of the whole system, even though the effect is not obvious. Initially in the early version of Android system, both motion event and key event used to be delivered to the application framework layer for callback processing, but soon system developers found that program running in Java circumstance can not react as they expected. To accelerate the program reaction, motion events are delivered to the view directly through pipe. Our first strategy rebuilt this callback processing for motion events and both actions for motion events and key events have to interact with the embedded SQLite database. We
are using virt-manager to manage our virtual machine and system images. From the virtual machine status condition panel, we could simply observe the Android-x86 system having kernel changed has heavier workload than other running pure system images. Some unexpected system reboot happened and interrupted the running program, the system after changing became unstable. Secondly, the first approach based on key word paradigm is useful in conveying more semantic information to users, but it requires the high-level taxonomy be updated each time a new type of gesture or sensor becomes available. The second approach can replay sophisticated touchscreen gestures while being unaware of their high-level action meaning, but it still has some shortcomings. Once the X or Y coordinates are changed, the values we got in input event stream are not values in pixel coordinates. To get this mapping, we first need to get the window size in pixels via command `wm size`. Then we can get the maximum value of input event via `getevent -p` command, and thus we can get the ratio of maximum pixel value by maximum input event value. By multiplying the new real values from event streams with this ratio, we get the real coordinates in pixels. Since we have not employed sophisticated gestures in this thesis project, the second approach is sufficient. Consequently, in the latest development, we adopted the second method.

### 3.2.2 Activity Exploration

Activity exploration here means that the content crawler walks inside an application from one page to another. Pages in applications are dynamically generated each time
when input events triggers an event handler, for instance clicking on a clickable UI element, for instance one TextView. An application can be formulated as a directed graph $G(V, E)$. Each node $v$ in graph $G$ stands for a page in the application. While an edge $u \rightarrow v$ indicates that the crawler can jump to page $v$ from page $u$ through some interaction through the application. As for this graph, we call it the static activity transition graph (SATG). Constructing SATG and determining the correct order in which GUI elements of an Android app should be explored is challenging. The main problem here is that the control flow of Android apps is non-standard: there is no `main()`, but rather apps are centered around callbacks invoked by the Android framework in response to user actions or background services. This makes reasoning about control flow in the mobile app difficult. The transition among activities is based on a generic intent passing logic. In some sense, intents play a role as "glue" during the activity transition process. According to the official Android documentation, an Intent is an "abstract description of an operation to be performed." [31] Intents can be used to start activities by passing the intent as an argument to a `startActivity`-like method. Intents are also used to start services, or send broadcast messages to other apps. After some investigations, we have realized that intent passing and consequently SATG construction can be achieved via static data-flow analysis, more specifically taint tracking. Hence the key insight is that SATG construction problem has become converted to a taint analysis issue. SCanDroid is a static analysis tool for android application [32]. It uses IBM's WALA framework [33] and does the static analysis directly with dalvik bytecode. Currently it is set up to do taint tracking analysis, though it can be easily modified to do other types of static analysis as well. In some sense, SCanDroid could be modified as taint tracking
tool and taint tracking could help researchers in future work to assess the crawling efficiency.

Algorithms used for directed graph exploration could be divided into two categories: depth-first (DFS) and breadth-first (BFS). Both of the two categories require some kind of status tracking strategy enabling the system to return some status in the past. The breadth-first approach requires the status to be stored in a first-in-first-out queue and the depth-first approach requires the status to be kept in a stack. Since app developers usually prefer putting many items together in a single page, instead of building a deep page hierarchy while arranging few elements in a single page, a depth-first solution seems to need less storage space than a breadth-first approach. From this point of view, we chose depth first exploration algorithm as our method. Nevertheless, DFS still requires status keeping and the ideal way is to keep a mirror image of the system. Unfortunately, this method is only available for the system running on virtual machine, such as Kernel-based Virtual Machine (KVM). Moreover the huge expense of time remains a problem. To take a snapshot of the system only takes a few seconds but the system image loading process will last for a few minutes. After considering the large number of activities for a mature app, this is expensive. In addition, generally speaking a smartphone user does not go too deep in page transition. We mean a normal user only browse the home page and a few layers deep and does not go through all page content. So based on this assumption, the content hidden too deep requiring too much navigation may not be interesting to users any more. This thesis project aims at implementing a proof-of-concept prototype system, we limit the DFS to a limited depth DFS and only consider the conditions when only back button is valid for returning to the past status of app. The limited depth DFS is shown in the Algorithm 1.

**Algorithm 1 Limited-Depth-First Exploration**

```plaintext
Require: Entry Activity A
        Limit Depth LDepth
        Current Depth CDepth
1:    procedure LDFT(A, LDepth, CDepth)
2:    if CDepth > LDepth then
3:        return
4:    end if
5:    Content ← ContentExtractor(A)
6:    clickableSet ← ViewControlExtractor(A)
7:    for each clickable in clickableSet do
8:        Click and trigger new Activity Aᵢ
9:        LDFT(Aᵢ, LDepth, CDepth + 1)
10:   end for
11:   end procedure
```

When the mobile application crawler reaches the home activity or home page,
the crawling process should begin. To traverse multiple activity pages on different hierarchies, we need to retrieve the content data as well as the hierarchical structure of page elements. In our implementation, one customized programming portal named appcrawler has been added into the system and has compiled together with Android-x86 source code. The appcrawler portal could be run from adb shell and has following functionalities as shown in the table 3.

The crawling program can get the hierarchical elements from the programming portal appcrawler as an XML string. Inside this XML string, the program traverses from the top to bottom, from outer to inner layer. When the crawling program encounters the elements like TextView or ImageView, the crawler will store its text string or take a snapshot of specific area by appcrawler screenshot command. Tables 1 and 2 show how the data is stored in MySQL database. The text is inserted into the MySQL database directly, and the image is assigned a randomly created name to be put at a specific place. The unique name is stored in the MySQL database. The processing strategies differ among various types of view elements.

To design the algorithm strategy to browse the hierarchical elements of a page, we need to consider the delivering and processing order of page elements. Generally speaking, the processing order of the motion event and the key event is different. When delivering a key event, the key event is delivered to the root view element to process and the parent page element has the highest priority to process the key event. When delivering a motion event, the motion event is delivered to the top focused child element and the element will run its overloaded OnTouchEvent method to process. If the child element returns false, the motion event will continuously be delivered to its parent element to process, in this aspect the child element has higher priority than its parent to cope with motion event. So when looking for a clickable element that can lead to next layer page, we make the program injects one click event on the blank area of parent element that has not been covered by its children. If the application does change focused activity, then the program would intimate user to click on the child element. When the crawler finishes traversing the current activity page and it needs to go back, it will inject one back event to the system to make the system return to the last status. In the current implementation, we assume that the application has not customized the reaction to back key pressing, but actually in reality it indeed can happen and will make the cross activity traversing more complicated. If the customization of back key is done, the crawler would have to use some kind of system status saving technology to restore the last status.

3.3 System Structure

The whole mobile app crawling system is a comprehensive project, as shown in the figure 5, it should be at least divided into four modules. Management & Interface, Content Module, Active Performance Measurement and Database Service. Each of them has different responsibility and works smoothly and they are connected with
Figure 5: Overall System Structure

Figure 6: Content Module
<table>
<thead>
<tr>
<th>field</th>
<th>property</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>int(5) Prilmary Key auto_increment</td>
<td>the auto generated id for each entry</td>
</tr>
<tr>
<td>page</td>
<td>varchar(100)</td>
<td>the harsh value of page xml string</td>
</tr>
<tr>
<td>package</td>
<td>varchar(100)</td>
<td>the package name of page</td>
</tr>
<tr>
<td>activity</td>
<td>varchar(100)</td>
<td>the activity name of page</td>
</tr>
<tr>
<td>text</td>
<td>text</td>
<td>the text string</td>
</tr>
</tbody>
</table>

Table 1: MySQL text data table

<table>
<thead>
<tr>
<th>field</th>
<th>property</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>int(5) Prilmary Key auto_increment</td>
<td>the auto generated id for each entry</td>
</tr>
<tr>
<td>page</td>
<td>varchar(100)</td>
<td>the harsh value of page xml string</td>
</tr>
<tr>
<td>package</td>
<td>varchar(100)</td>
<td>the package name of page</td>
</tr>
<tr>
<td>activity</td>
<td>varchar(100)</td>
<td>the activity name of page</td>
</tr>
<tr>
<td>image</td>
<td>varchar(100)</td>
<td>the name of the stored image</td>
</tr>
</tbody>
</table>

Table 2: MySQL image data table

each other to work efficiently. This thesis project mainly includes working effort in Management & Interface, Content Module, and Database Service. The Active Performance Measurement is mainly designed and implemented by my colleagues in Tsinghua University. They are integrated into a completed unit to finish the job and details of implementation will be described below.

### 3.3.1 Management & Interface

The management & interface plays a role interacting with clients or users, and this system could be implemented as some kind of desktop or mobile software. In our development, we developed this front-end as a user-friendly webpage. Clients can view collected data and control the whole process via buttons at the right-bottom corner or the Android touch screen at the left-top place. To provide a user-friendly workflow, we used the Guacamole framework to enable VNC connecting on our webpage. Guacamole ([http://guac-dev.org/](http://guac-dev.org/)) is a light weighted clientless remote desktop control protocol, and it is based on HTML5. Guacamole can be supported by most mainstream browsers on the software market. Guacamole has its servlet running in tomcat web container and the guacd daemon process will interact with the remote desktop server via VNC or RDP protocol but will provide a unified data stream to browser pages using guacamole protocol based on websocket. In terms of the Guacamole framework, the management interface runs on a single stand-alone server and send or receive control order via REST requests. By pub-sub system, the back end server pushes crawled content data lively to pages via RabbitMQ. RabbitMQ is a message delivering framework [34]. The server pushes one packaged message to the stand-alone message buffer and the client side will receive coming messages from this specific buffer based on websocket. When receiving messages, the browser
As shown in figure 9, the home page of the management interface enables user to upload the apk file of target mobile application. After the uploading, if the procedure is successful, the management interface would jump to the work page as shown in figure 10. At the work page, the panel at the left side displaying a screen of Android virtual machine is the VNC channel. User could view and even operate on this screen to control the Android environment directly. The column to the right side of the VNC panel is the data display and control area. Users operate the whole process through the buttons at the right-bottom corner. For additional information such as past crawling history and more details about this project, users can click on the links at the right-top corner of these two pages.

3.3.2 Content Module

SandBox is the isolated environment in which the mobile operating system runs. The sandbox entities can either be real mobile devices or virtual machines. We chose emulated virtual OS from two aspects of considerations: (1) Monetary cost: using real mobile devices will definitely increase the monetary cost while it is easy to find free virtual machines which can be deployed in servers. Additionally, using virtual machines can easily eliminate the gap in different hardwares. (2) System security hacking: this thesis project has succeeded collecting text, image and network data from some open source app, but how to record stream video and audio still remains an open question. In terms of the system security reason, to record these two types of data may require to hack the Android OS. Obviously to adopt a virtual machine will be easier for further development. Since the thesis project requires some modification of Android source code and we chose Android-x86 as the virtual machine for running our target mobile application. We have chosen Android-x86 based on 4.4.2 Kitkat as the based source code and it has been compiled to ISO format system image on Linux system. Because our server is x86 architecture and the CPUs for the server has also x86 instruction set, we chose Android-x86 as the source code for virtual machine.

The customized programming portals have been put under the folder frameworks/basecmds. To run or communicate with those programming portals needs to enter the adb shell environment. And the description for each program file is listed as below:

**AppCrawler.java** The main logic for the program of this portal.

**UiAutomationShellWrapper.java** The program file for connecting the main logic program to the UiAutomation framework.

**App Driver** is a component designed for imitating human beings to interact with mobile applications and driving the application to explore activities. This
component can drive an application in two different manners. The first approach is named ERR (Event Recording and Replaying), which supports recording a series of input event stream, and replaying it later when needed. This agent can play a role as time warping to fast forward the app to some specific status automatically. In our implementation, this ERR agent has ability handling sophisticated GUI gestures with exact timings. The second approach is to explore activities in a limited depth DFS algorithm.

**Data Extractor** is a logical component which is responsible for extracting data from target mobile applications. Currently displayed text, images and network layer data are supported, meeting the demand for monitoring the target mobile application. Text is fetched from UI xml strings and images are fetched by taking snapshots via `appcrawler screenShot` command using provided parameters to target a designated area. The recording of network data is mainly done through the tcpdump toolkit and the program reads the IP addresses, transmitting speed or other standards from tcpdump toolkit. The program reads the traffic value from `/proc/net/dev` device file. All these sources are later processed by a named `network_analysis.sh` bash script periodically and output to console time to time. In the real practice, this bash script is performed repeatedly after certain time intervals.

The content module runs as a REST server deployed at the same server with the virtual machine and communicates with the virtual machine with `adb shell` commands. The server utilized Jersey framework to implement REST portals and iBatis [35][36] for database operation. It keeps a close synchronization with Android-x86 virtual machine and the target mobile application running on it. The figure 7 displays how the crawling module works; the whole workflow would be applied during the whole crawling process. The program first retrieves the content xml string of current focused activity page from `adb shell appcrawler dumpView` command. Then the program traverses the hierarchical element tree to analyze the elements one by one; if one element contains content data, the program will then store the text into the MySQL database or take a snapshot of a specific area. If the element is clickable or can apply swipe action, the program will inject related events into the system by using `adb shell appcrawler click` or `adb shell appcrawler swipe` commands. If one new activity page is entered, the program will store current crawler status, for instance which element the crawler is focusing on and which activity page the crawler is looking at. Then the crawler will re-obtain the content xml string again and start a new traversing. Since during the element analysis section, the number of Android view element types is quite large, and some customized view element may still exist, we only focus on crawling on a customized application that has limited number of elements, for example TextView and ListView. This is a proof of concept, in the future we make this mobile application crawler prototype more completed and mature.
Figure 7: The Work Flow for Crawling a Page
3.3.3 Active Performance Measurement Bed

The information of a mobile app we are interested in, as shown in figure 1, could be divided into three layers. The crawling system for the highest layer *Displayed Content* and the middle *Network Performance* has been implemented partially by my thesis project. But as for the *Server Infrastructure* layer, my colleagues in Tsinghua University has already implemented it as a distributed system, so-called Active Performance Measurement Bed. The data and information got by this Active Performance Measurement Module will for sure contribute to further research and help researchers understand the hidden network behaviors of an app better. All our work together will make up a complete system for retrieving information from a mobile app.

The Active Performance Measurement Bed which is a distributed system consisting of over 60 measurement nodes placed in 40 cities spreading across 32 provinces and regions. This measurement bed is designed and implemented by developers at CDN lab in Tsinghua University, and here we connected this measurement bed to existing system to provide necessary performance data. As ISPs are structured at the level of provincial districts and the Internet user population is clustered around a number of large metropolitan districts. We placed our measurement nodes in a way that covers the full range of ISPs and large city areas to ensure the most representative network access scenarios.

The measurement platform takes the URLs from the Link DB filtered by content module. The tasks of this active performance measurement bed include: (1) quantify the usage of infrastructures used across regions and (2) measure the performance of service provided by these infrastructures. Toward these goals, our measurements and analysis follow the procedure as shown in the figure 8:

1. **Domain name extraction**: we extract domain names from the input web links, eliminate duplicate ones and get unique domain names prepared for DNS lookup.

2. **Distributed DNS lookup and analysis**: on each measurement node, we utilize the nearest public DNS server to resolve all the domain names and obtain their CNAME, IP addresses and NS information. These data will be sent to a centralized data analysis node to remove redundancy.

3. **CDN correlation**: we compare the DNS analysis results against a CDN domain name table and correlate the domain names of original links with CDN providers (if exist) that serve the contents. In detail, we try to understand what objects of which application are served by which CDNs.

4. **Infrastructure performance analysis**: using the IP addresses resolved from local DNS lookup, we utilize performance measurement tools to evaluate performance metrics like DNS lookup time, packet delay and loss rate.
3.3.4 Database Service

The Database Service layer provides some data storage service for its upper level systems, and it should have enough scalability as well as efficiency. In current experiment progress, there are three types of data which need to be kept in the database service, by default it is a relational database. They are content data including text and images produced by the Content Module, the URL links information fetched by the Content Module as intermediate input source for Active Performance Measurement, and the performance measurement results generated by Active Performance Measurement module. We have deployed MySQL relational database in the same server machine with the back-end system. As for the content data, table 1 shows organization of text information and table 2 shows how images are recorded in the relational database. The text is inserted into the MySQL database directly, and the image is assigned a randomly created name to be put in the file server. The unique name is stored in the MySQL database. In future, the relational tables for videos and audio could be added, but so far have not been considered. As for the Performance DB, the measurement results are written into separate files stored in the disk. In the future, there will be a need to record the operation path to reach some app status, some graph data storage solution may be necessary, such as Neo4j.
Figure 9: The Home Page of Overall System

Figure 10: The Working Page of Overall System
4 Results and Conclusions

The proof-of-concept crawling system which was implemented in this thesis project has partially succeeded. The system could be regarded as a prototype of a transparent agent architecture to help app developers or any researchers to build some malicious content monitoring system and make unindexed app content publicly searchable. The monitoring for mobile applications covers three aspects: displayed activity content, network-layer infrastructure and network-layer behavior. Two function modules, content module and active performance measurement bed, are tested separately, and results are discussed below.

4.1 Content Crawling Result

Limited depth DFS has been implemented in this function module, and to evaluate whether the algorithm could work as we expected we ran our program on a open source app StartupNews (https://github.com/sqbing/StartupNews). This open source mobile app is a quite popular application and has already been deployed on a significant number of smartphones. We deleted its UI settings and files for being displayed on pads, and only reserve the UI programs for being displayed on smartphones. Complexity of activity UI elements has been reduced, and unnecessary content dynamism has been removed which means that only static content is left on the app. As for the activity transition display, we made the crawling program to wait for some display time before checking whether the activity has been changed. We set up this display time by our experience in the experiment, which is ten seconds. During the traversing among activities, the content module has successfully retrieved all text and took snapshots of all images it encounters. The data extractor has also recorded all network interaction data, such as RTT time, for all network connections. During the traversing, the content module has also taken correct actions when facing different page elements. For instance, when the content module encounters a ListView, it will make App Driver part to send a corresponding scrolling action to the ListView with the length of the longest displayed item in the List. Once the new coming items appear, Data Extractor reads the item and check if the item could be any navigation to the next level. During the whole crawling procedure, normal widget UI elements are fine, but some complicated gesture failed to be handled. For instance, at some second layer activity, a new picture would appear only when user scroll horizontally over the old displayed picture. This picture area is shown inside an ImageView element, and obviously the app customized the event listener of the ImageView. This little mistake requires some further research in future, we need to figure out how to detect whether event listener has been overloaded.

4.2 Active Performance Measurement Result

The mentioned active performance measurement bed has already deployed and began running at Tsinghua University CDN lab. To evaluate this measurement bed,
we chose 40 popular mobile Android apps from download ranking list by 7 categories: News(8), Service(8), Shopping(7), Video(9), Audio(4), App Market(3) and SNS(1). When considering the limitations of current crawling module, some manual effort has been introduced. Apps are installed in the Android environment that is for the content module, we operate them to generate network interaction as much as possible. Network-layer behaviors are collected, object links are extracted and an at-scale measurement with links are performed. These links are distributed to the measurement nodes and perform measurement tasks as: DNS query, RTT measurement and CDN mapping.

As the measurement results shown in figure 11, all evaluated apps have used CDN services and 50% of them used only one CDN provider. All apps from members of BAT (Baidu, Alibaba, Tencent) consumes only one CDN provider, that is the CDN service deployed by themselves. Detailed statics of each link measured from different measurement nodes are also collected. In terms of DNS resolution time plays a critical role in quality of service [37], DNS resolution has been used as metric to judge the quality of CDN service. Therefore we have drawn the CDF plots of DNS resolution time from different CDN providers. The second plot in the figure 11 indicates that CDN services from BAT (Baidu, Alibaba, Tencent) are about equal to each other. The second plot in figure 11 indicates the data for top 3 CDN services, the three CDN provider companies with highest market occupation rate. The last plot in figure 11 the three advertisement platforms utilized by apps vary a lot, so thus will definitely have impact on the speed of advertisement loading time.

5 Future Work

Current research and development work has successfully worked out a proof of concept, which has proved that a mature and robust mobile application crawler can be continuously developed starting from current framework and structure. Even though the current demo has successfully retrieved the activity page content and recorded each network interaction metadata, the program still has a few drawbacks and needs future improvements.

5.1 Saving and Restoring System Status

If one page element is clickable, the crawler would inject one click action to the element to enter next page. When the crawler needs to come back to the previous page, in most cases, one click on back button would work. But in some special case, the reaction of back button has been customized. In addition, the visit to current page may cause content change in the previous page. Therefore, clicking the back button would not ensure to be back to the page the crawler used to stay before. In this aspect, a mature mobile application crawler should have the ability to save and
restore the system status in acceptable and reasonable time.

Our virtual machine is KVM and we used virt-manager to manage the virtual machine system images. Libvirt provides the toolkit to interact with virtualization capabilities of recent Linux systems and it has system snapshot functionality as expected. The commands for taking system snapshot are listed below:

**taking snapshot** virsh snapshot-create-as domain VM_NAME name snap01 description Description of snap01 for VM_NAME

**view snapshot** virsh snapshot-list VM_NAME

**restore snapshot** virsh snapshot-revert VM_NAME snap02

During our testing, to take a snapshot of the virtual machine system, Libvirt toolkit requires five minutes and to restore a system image snapshot requires thirty seconds. When consider the vast number of possible potential activity pages, we decide to limit the crawling depth to three levels of depth, because a normal user usually visits pages at maximum a few levels of depth and the app designers are inclined to put the most important content at the most front pages. As each page contains ten clickable view elements which can lead to next layer activity pages, to
crawl all these three layers’ activity pages would need to handle 100 activities’ transferring and would need 600 minutes for taking snapshots and restoring snapshots. In practical usages, a more complicated application containing more activity transferring may take more than one day to take and restore system snapshot. Obviously, this is quite expensive and a better solution is required.

5.2 Simulate the Circumstance to cope with Client-based Dynamism

The pages having client-based dynamism have content which is related to specific locations or specific user profiles. To crawl completed content from these pages requires the crawler to be able to traverse all possible paths, which means the crawler should be able to visit the app in all possible statuses. The app retrieves location or user information from the system sensors, which means that if developers can control the sensors in some way to let them be some fixed values as expected the system can simulate to be in some specific environment.

In 2005, Google published a patent on a distributed testing system. Some company has also already developed its distributed mobile application testing system [30][38][39], in which mobile apps are operated on servers distributed in different locations and testing probes collect network request/response information for analysis. To simulate the various location information, we may borrow the idea from this system to let the mobile application crawler fetch content at different locations and use different user accounts to log in. In another aspect, when we consider the complexity of a mobile application, a distributed crawling strategy may simplify the crawling process in a significant level.

5.3 Modeling Form Submissions to Handle Input Dynamism

Based on studies performed in 1997, Lawrence and Giles [40] estimated that close to 80% of the content on the Web is dynamically generated, and that this number is continuing to increase. The web information containing dynamism is called ”hidden web”. To retrieve this tremendous amount of content information from this part of hidden network area, researchers have developed a few technologies to hack the entry point by mimicking users’ behaviours, for instance the HiWE done by scholars at Standford. Even though there has not been any research on the percentage of hidden we occupying all web information, the entry points for dynamic web data have been observed in many mobile applications. Consequently, to retrieve data from those entry points can make researchers and developers discover many interesting things that have not been searchable by users from the search engines.

The entry points for hidden web on mobile applications are part of interface elements, t.e.x EditText and Spinner. Here we define the term domain as the possible values that could be filled up into these input elements. According to how many possible values could be put into these input elements, they could be divided into
two categories, with finite domains and with infinite domains. When handling input with finite domains, the crawler can simply try each possible value exhaustively. The tricky part is to handle input with infinite domains, which requires a more careful investigation. Possible domains are infinite, but not all possible values are reliable which means the crawler should be more confident when crawling from some specific domains than others. As shown in the figure 12, this is the classic workflow designed by HiWE at Stanford. After choosing one clickable interface element, if it were a form, the program would analyze the form property to see whether it is with finite values or with infinite values, and also check the proper way to model the form. After checking the properties of the form element, the program would assign possible values to the input and submit the form. To calibrate the domain set and value modeling, the system will analyze the response to check whether the domain model should be changed.

5.4 Continuous crawling to cope with Temporal Dynamism

The activity pages containing time sensitive information have temporal dynamism, and to retrieve content from these pages requires the crawling actions to be repeated after certain time intervals. For instance, a stock app displaying the trading data of stock market is updated every day and to retrieve data from this type of application requires the crawling strategy to be repeated each day. In addition, to reduce the workload and reduce the complexity for crawling, intermediate pages before updating activity can be skipped and the operation sequence entering the target activity
will needs the operation recording and replaying toolkit we developed in this thesis.

In practice, to retrieve content data from a mature mobile application requires the development of framework flexible enough to handle mainstream applications. To investigate whether some specific crawling framework can handle all applications, some categorization of applications may be necessary based on general navigation patterns of applications. Because the application on current software market have some common design and navigation routes for developers to follow. More over, to help record operation sequences for some framework and provide some way to replay these operation sequences, some toolkit will be necessary.

5.5 Expand Usage of Current Mobile Application Crawler to HTML5 App

This thesis work focuses on how to retrieve content data from native application on the Android system, but quite a significant part of mobile applications has adopted cross platform development framework using HTML5 and Javascript, PhoneGap and Worklight. A cross platform development framework can remarkably reduce the cost and workload for developing mobile applications. On the Android system, the web app developed by PhoneGap framework will finally be packaged into a WebView page, and its running depends on the embedded Webkit or Chromium engine. So in some sense, the problem how to crawl a HTML5 app could be converted to the problem how to intercept WebView elements to retrieve their content.

References


Appendices

A  List of Acronyms

URL  Uniform Resource Locator
SDK  Software Development Kit
GUI  Graphical User Interface
DNS  Domain Name System
JNI  Java Native Interface
OSI  Open System Interconnect
ISO  International Organization for Standardization
CCITT  International Telegraph and Telephone Consultative Committee
KVM  Kernel-based Virtual Machine
VNC  Virtual Network Computer
RDP  Remote Desktop Protocol
CDN  Content Delivery Network
ISP  Internet Service Provider
ICP  Internet Content Provider
RTT  Round-Trip Time

B  Programming Portals
<table>
<thead>
<tr>
<th>command</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>appcrawler dumpView</td>
<td>print out the hierarchical element as xml string format</td>
</tr>
<tr>
<td>appcrawler dumpFocusedWindow</td>
<td>print the information of focused window</td>
</tr>
<tr>
<td>appcrawler dumpFocusedApplication</td>
<td>output the information of focused application</td>
</tr>
<tr>
<td>appcrawler listInstalledApplications</td>
<td>list all installed application’s details on current system</td>
</tr>
<tr>
<td>appcrawler startApplication</td>
<td>start the application based on offered information</td>
</tr>
<tr>
<td>appcrawler startActivity</td>
<td>start the activity according to provided data</td>
</tr>
<tr>
<td>appcrawler screenShot</td>
<td>take a snapshot of specific area by provided coordinates</td>
</tr>
<tr>
<td>appcrawler swipe</td>
<td>inject stimulated swipe gestures into the system</td>
</tr>
<tr>
<td>appcrawler click</td>
<td>inject stimulated click action into the system</td>
</tr>
<tr>
<td>appcrawler key</td>
<td>inject one key event into the Android system</td>
</tr>
<tr>
<td>appcrawler events start</td>
<td>start recording operation sequence for specific application</td>
</tr>
<tr>
<td>appcrawler events stop</td>
<td>stop recording operation sequence</td>
</tr>
<tr>
<td>appcrawler events clear</td>
<td>remove all recorded operation sequence stored in SQLite database</td>
</tr>
<tr>
<td>appcrawler events list</td>
<td>output all stored events to the screen</td>
</tr>
<tr>
<td>appcrawler events isRecording</td>
<td>check if currently system is recording the operation sequence</td>
</tr>
<tr>
<td>appcrawler events targetPackageName</td>
<td>return the name of the target application to which system is recording</td>
</tr>
<tr>
<td>appcrawler events targetUid</td>
<td>return the unique user id of the target application</td>
</tr>
</tbody>
</table>

Table 3: appcrawler commands