Copyright ⓒ 2018 KSII

An Automatic and Scalable Application Crawler for Large-Scale Mobile Internet Content Retrieval

Mingyi Huang, Yongqiang Lyu and Hao Yin
Department of Computer Science and Technology, Tsinghua University
Beijing 100084, P.R. China
[e-mail: tobyxdd@gmail.com (M. Huang)]
[e-mail: luyq@tsinghua.edu.cn (Y. Lyu)] *
[e-mail: h-yin@tsinghua.edu.cn (H. Yin)]
*Corresponding author: Yongqiang Lyu

Received February 28, 2018; revised April 19, 2018; revised April 28, 2018; accepted May 31, 2018; published October 31, 2018

Abstract

The mobile internet has grown ubiquitous across the globe with the widespread use of smart devices. However, the designs of modern mobile operating systems and their applications limit content retrieval with mobile applications. The mobile internet is not as accessible as the traditional web, having more man-made restrictions and lacking a unified approach for crawling and content retrieval. In this study, we propose an automatic and scalable mobile application content crawler, which can recognize the interaction paths of mobile applications, representing them as interaction graphs and automatically collecting content according to the graphs in a parallel manner. The crawler was verified by retrieving content from 50 non-game applications from the Google Play Store using the Android platform. The experiment showed the efficiency and scalability potential of our crawler for large-scale mobile internet content retrieval.

Keywords: Mobile internet, app, crawler, content retrieval, automatic test

http://doi.org/10.3837/tiis.2018.10.013 **ISSN : 1976-7277**

1. Introduction

Usage of smart devices, such as mobile phones and tablets, has been growing rapidly in recent years, exerting significant influence on the daily lives of humans around the globe. Advancements in computer hardware, software and network technologies mean that smart devices are no longer limited to manufacturers' pre-built functions. Instead, they serve as multi-purpose platforms that allow users to install additional applications to enjoy more features and enhanced functionality.

The wide variety of messaging services, news sites, social platforms, blogs, and microblogging applications available to existing smart devices have become a ubiquitous source of information and user-generated content. However, many of the services restrict the way users access their content. These services are not actually incorporated into the open Internet [1], and it is difficult to retrieve their content with traditional methods (e.g., web crawlers). The technical differences between traditional web pages and mobile applications require new techniques to collect content from the mobile internet.

Most of the relevant studies on this topic have focused on automated tesing for applications [2],[3] and commercial advertising [4]. Few studies have worked directly on content retrieval. Compared to traditional web crawling techniques, there are several major challenges to be overcome.

The first challenge is the difficulty in locating and labeling contents. Lacking a mechanism like a URL, there is no universal standard to locate specific contents within an application. Traditional web crawling is therefore not useful for mobile applications.

The second challenge is the lack of a unified language for content presentation. Traditional web pages are delivered to browsers as HTML (Hypertext Markup Language). HTML makes it possible for traditional crawlers to parse content from different pages in standardized ways, but mobile application interfaces may call graphics APIs and implement custom behaviors. Newer standards have been developed to address such challenges, such as HTML5-based application frameworks like React Native [5]. However, the severely limited adoption of such frameworks in various applications still frustrates content collection from mobile application interfaces.

The last challenge is the diversity in application designs. Differences in hardware functionality and system designs between devices can be huge, resulting in applications that change content presentation and user interaction from device to device. For a traditional website, hyperlinks are usually the only way to link pages together. In contrast, mobile applications may use touches, swipes, shakes, or sensor interaction to present new content.

To address the challenges above, we propose an automatic and scalable mobile content crawler (MCC), which does not need to directly access the source code of target applications or their extracted resources. It can be executed by any third party to obtain content from released applications without manually customizing configurations. The crawler can automatically collect content from mobile applications running on "mobile device server farms" consisting of real devices, emulators, or virtual machines by traversing the interaction paths with the application interfaces.

Our primary contributions are as follows:

- A mobile crawler that automatically collects several types of content (e.g., text, images, audio, and video) in parallel from target mobile applications.
- An interaction-path graph generation (IGG) algorithm specifically designed for the crawler, which can analyze the target application structure and then direct the content crawler to execute interactive operations, traverse each interface, and collect the contents. The interaction-path graph is partitionable into independent subgraphs that enable the crawler to retrieve content in parallel for large-scale retrieval or multiple applications.
- Verification of our method's capability and efficiency with real applications using a statistical testing method for content coverage and accuracy.

We have organized our paper as follows. Section 2 discusses the background, basic rules, and requirements for the mobile crawler. Section 3 analyzes the design of each component. Section 4 describes the MCC implementation in detail. Section 5 reports the results of our performance tests on the proposed system. Section 6 concludes the paper.

2. Background and Motivation

2.1 Significance and Necessity of Retrieving Content from Mobile Applications

The importance of mobile applications, as opposed to web browsers, in users' online experiences has risen alongside the expanding use of mobile devices. The number of Android applications available from the Google Play store currently exceeds 2.2 million, with a similar number, 2 million, for iOS in the Apple App Store [6]. By 2015, U.S.-based social networks had 150 million mobile terminal users [7]. There are 574 million users in China [8]. Over half of the world's population (52.7%) uses the mobile Internet [9]. According to Cisco [10], the monthly data flow across the mobile internet has increased by 76%, from 2.1 EB in 2014 to 3.7 EB in 2015. Mobile internet traffic accounted for 8% of the world's Internet usage in 2015 and is expected to reach 30% in 2020.

However, technical and imposed limitations prevent web crawlers from retrieving significant portions of mobile internet content. The instant messaging service Telegram has a Channels feature for individuals and organizations to post content only available within the Telegram application. Xianyu, a popular Chinese trading site, has recently shut down its browser version completely, leaving their mobile applications the only way to use their services. The relocation of data from browsers to applications makes the acquisition and analysis of data more difficult. To show the significance of the issue, we tested 42 popular mobile applications on iOS and Android and determined the accessibility of their content using a web browser. **Table 1** shows the results, separated into three categories. "Functions limited" means that the operations that can be performed via the web are fewer than those from the mobile applications. "Content limited" means that the content is less accessible from the web than mobile applications. "Non-accessible" content is completely inaccessible from the web and encompasses both the "functions limited" and "content limited" categories.

Table 1. Level of content openness of 42 web applications

Application types	Non-accessible	Functions limited	Content limited
Social media (11)	9% (1)	27% (3)	9% (1)
Instant messaging (10)	30% (3)	60% (6)	30% (3)
News (14)	7% (1)	7% (1)	14% (2)
Education (7)	29% (2)	43% (3)	43% (3)
Total: 42	17% (7)	31% (13)	21% (9)

2.2 Requirements

After careful consideration, we propose the the following requirements for our new mobile application crawler.

Portability: The crawler should use only documented APIs and features of the target platform, avoiding the use of private interfaces or source code modifications to the platform. Using undocumented features or modifying the platform may result in undefined behaviors and severely limit the number of environments supported by the crawler.

Black box [11]**:** Application traversal and content retrieval should be performed in a black-box manner. That is, the crawler should not need to access target applications' source code. The main logic of traversal and retrieval should also be platform-independent to support implementation on other platforms with similar designs.

Sandbox [12]**:** Content collection from various applications automatically and comprehensively means that it is impossible to ensure that the behavior of the target applications can be fully trusted. The operating environment should be isolated to prevent potential malicious code in the target applications from damaging collected data or even the crawler program itself. This isolation ("sandboxing") prevents harmful actions from executing and interrupting the crawling process. The crawling environment can be restored to its original state at any time. Additionally, the use of these sandboxes permits our design to scale by using more virtualized crawler environments.

2.3 Challenges

The underlying technologies, development workflow, and user experiences of mobile applications differ significantly from traditional desktop web pages. In this subsection, we will discuss these differences and the technical challenges of creating a crawler for mobile platforms.

Locating and labeling pages: Web pages are usually located by URLs and linked to each other by hyperlinks. Standardization simplifies parsing of web page contents [13]. Mobile applications lack such a mechanism and, thus, a standard way to locate specific content. Lack of knowledge about the application's structure makes it harder to discover a consistent path to specific content.

Variety of content types and presentation methods: Traditional web pages are formatted and structured in HTML, which has a standardized parsing process for web crawlers. In contrast, mobile application interfaces are generated and rendered by the application itself and may contain more types of content. The absence of a unified standard increases the challenge in recognizing and retrieving mobile content.

Diversity in interaction methods: Additional hardwares in mobile devices provide more types of application interactions to the user. The crawler must support more methods to explore content accessible with specific interactions.

2.4 Comparison between the State-of-the-Art Techniques

Many tools, frameworks, and libraries are currently available to simulate hardware events and verify content across popular mobile platforms, mostly for automatic testing and performance evaluation. **Table 2** lists features and capabilitites of some of these packages.

Table 2. Android automation frameworks and tools

Technique name	Method	Operations supported	Data returned	Customization
Monkey	Automated testing (monkey test, crash test)	Random UI event	Application crash/error log	Number and frequency of random events
A3E [14]	Exploration for systematic testing	Bytecode analysis/UI events	Structure of the application	/
SmartAds [4]	Display ads	UI events	/	/
Robotium [15]	Automated testing	UI/hardware events	Script	Script
PUMA	Automated testing	UI/hardware events	Script	Script

Monkey is a tool developed by Google for Android. It sends a pseudo-random stream of user events into the system, which acts as a stress test on the target application. It cannot retrieve actual content from the application. Instead, it watches the system and generates reports if something goes wrong. It is a popular choice for monkey testing, but is not useful for content retrieval.

The Automatic Android App Explorer, commonly known as A3E, is an automated GUI testing toolkit for Android applications. It simulates user actions to explore and discover application functions, including those that might be hidden from the user. However, its main purpose is to analyze the application structure, so it offers only limited behavior and content retrieval customizations.

SmartAds is a contextual advertising technique that scrapes application content at runtime, extracting keywords and fetching contextually relevant ads. It does implement content retrieval and analysis, but these functions are limited to the content the user is currently viewing. Because it is designed for showing contextual ads, it does not need to automatically explore the application by itself.

Robotium and PUMA automate application testing. They are frameworks that support simulation of various inputs and events and retrieval of UI content. However, they can only execute pre-defined test cases written by test developers. They are unable to perform automated crawling.

3. Designs

In this section, we present the design of our MCC along with the methods it uses to explore application structures and collect content. We used Android as our target implementation platform, so we also provide a brief introduction to the Android operating system and application architecture.

3.1 Overview

Most modern mobile applications consist of multiple independent interfaces, where users interact with applications by touching and manipulating visual elements or by pressing hardware buttons. For example, a news application usually has three main components: a page showing a list of news titles and thumbnails, a page that shows content for a specific news item

including text, images, videos, etc., and a page that shows user comments and an input field to enter a new one.

To process such a structure, our crawling system consists of three major steps:

- Interaction graph generation (IGG);
- Task generation;
- Task execution (content collection).

Discovering the logic and structure of an interface is similar to sitemap discovery with a traditional web crawler, and the discovery is extremely helpful to the content collection logic. The process begins with IGG, an exploration of the application coupled with analysis of the interfaces to produce an interaction graph representing the application's structure. The crawler then uses this graph to generate the list of tasks, which then retrieve content concurrently.

3.2 Architecture of Android Applications

We chose to implement our proposal on Android because the platform currently has the highest market share among all such smart mobile devices [16], with a tremendous variety of applications to use as crawling targets. It is open-source, available for many hardware platforms, and supported by manycross-platform development tools. Thus, it is an excellent platform for testing our implementation of MCC. Our aim here is to present the general architecture of Android applications and the common components of their user interfaces in order to facilitate our presentation of the design and implementation of the MCC.

Android applications are typically written in Java. The Android SDK (Software Development Kit) is a prebuilt Java environment including all necessary libraries for mainstream Android development. The SDK compiles the source code into the DEX byte codes used by the Dalvik virtual machine. The byte code output, along with other resources, dependencies, and manifest files, are compressed and packed into Android package (APK) files for distribution. When installed on a user device, the byte codes are compiled to the device architecture and executed. Unlike many traditional desktop applications that use numerous different libraries or even directly render user interfaces with graphics APIs, most Android applications use the XML-based interface engine provided by the system. Doing so gives the Android system the capability to retrieve and manage content on the screen in support of auxiliary APIs such as accessibility services. The overall mechanism provides a reliable way for our crawler to query the content of target applications.

The term "activity" refers to the application component that directly interacts with the user. This component can be regarded as a single page of interfaces. An interface is created by filling the activity with elements such as labels, buttons, text boxes, images, or progress bars. The user interacts with the application via these elements. Each interaction action can alter the current interface or the application's status, or it can open a new activity. An application typically consists of many activities loosely connected to one another; different activities are used for different content. For example, a blogging application first shows an activity presenting titles of blog posts from all followed users. Touching an item in the list brings up an activity showing the content of a blog post, and touching an avatar brings up the user's profile. Other activities enable users to create new posts or change settings. **Table 3** shows the number of activities used by a few popular applications.

Table 3. The count of activities for some common applications

Names	Types	Number of Activities
Twitter	Social media	36
Dropbox	Cloud storage	16
Engadget	News	9
BBC	News	9
NYTimes	News	15

3.3 Generation of Interaction Graph

Each Android application has only one entry point, but different interactions can then lead to different content. One series of interactions forms one interaction path. **Fig. 1** shows an interaction graph of an ideal news application.

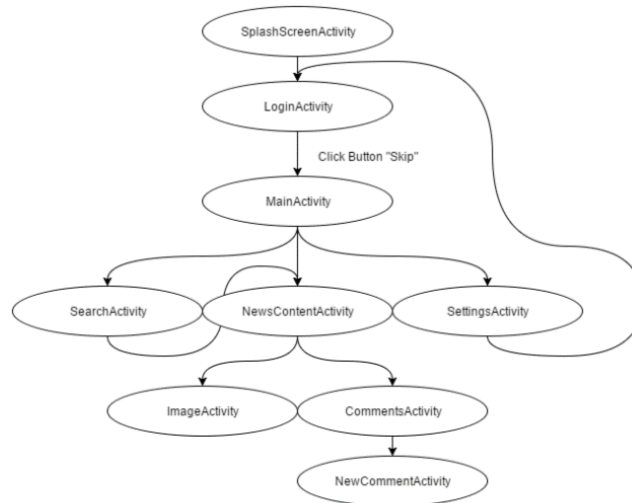


Fig. 1. Interaction graph of an ideal news application

The IGG algorithm first explores the structure of the target application and then automatically generates an interaction graph by collecting and analyzing elements within the interfaces for both type and content.

If we assume that the structure of an application's interfaces is a tree where each interface is a node in the tree, and the nodes are connected by some types of interaction (e.g., touching an item or swiping), then ideally, a depth-first search (DFS) is suitable for searching the whole application. With the main entry interface as the root, it is possible to interact with all elements sequentially, obtain all new elements after reaching a new interface, and then interact with them again. This process is repeated until every path has reached a node that has no child. **Fig. 2** shows the result of searching that ideal news application above in this way.

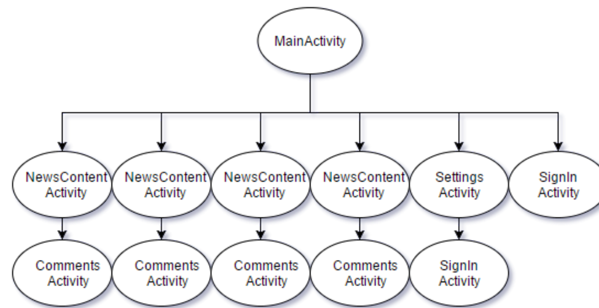


Fig. 2. DFS result of an ideal news application

The DFS results are inaccurate for multiple reasons. First, the path from the main activity to the news content page to the comment page is collected multiple times. This occurs because the main interface usually contains a long list of multiple news titles while the actual activity used for showing the content is a reusable NewsContentActivity. Second, both the main and settings interfaces provide an entry to the sign-in interface, SignInActivity, thereby generating two paths to it. The limitations of this simple DFS algorithm and tree structure prevent detection of the situation and the representation of them as the same interface.

The interaction paths of real-world applications are often more complicated than those in ideal applications. It unlikely, for example, that an application would ever have an interface with no elements leading to another interface. Consider a modern news application with a recommendation function. At the bottom of each content activity, there is a "similar articles" list offering other articles. Touching these items presents another news item which also contains a "similar articles" list, generating an endless path.

After testing and analyzing the interface structure of several popular applications, we grouped all nodes into 3 categories (**Table 4**) and identified 3 types of special structures that can lead to problems (**Table 5**, **Fig. 3**)

Table 4. Node catalogs

Names	Description
Identical	The nodes have identical types of elements and content.
Same type	The nodes with same logical functions and structures but provide different contents. (the content pages of different news articles / the profile pages of different users…)
Distinct	Nodes of completely different types, structures and contents that can be regarded as logically and functionally distinct.

Table 5. Special structures

Names	Description
Multi-entry interface	The user can enter several completely identical nodes by performing different actions
Interconnection of same-type interfaces	Nodes of the same type and same level connect to each other directly.
Cyclic interconnection	A certain interaction path leads to a sequence of interfaces being connected end-to-end, creating a loop.

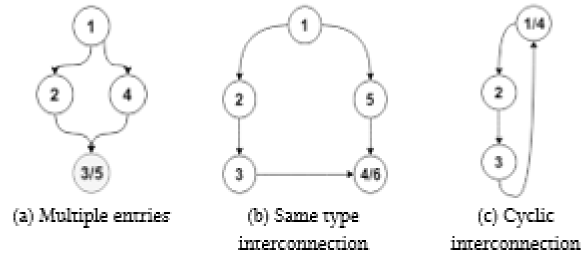


Fig. 3. Special types of connection

As a result, a robust IGG should also have the following features.

Depth-first search: In the breadth-first search strategy, all direct children of the root must first be traversed once before moving to the next level. In this project, the actions to access connected nodes themselves are time-consuming since they require simulated interactions with target applications. The target applications may require time to communicate with servers, perform calculations or play animations before showing the actual content. Due to the inability to switch directly between child nodes, BFS requires returning to the parent node every time before proceeding to the next child node.

If the target application has a full binary tree structure with depth d, the average time for one interaction is T.

The total time spent on interactions usinga DFS-based IGG to traverse the tree and determine there are no more nodes TT_{DFS} is

$$TT_{DFS} = T \times \left(2\sum_{n=0}^{d-1} 2^n\right) \tag{1}$$

A BFS approach would require

$$TT_{BFS} = T\times(2\sum_{n=0}^{d-1} 2^{d-n}(2n+2)) \tag{2}$$

Therefore, the DFS approach greaty reduces unnecessary switching, improving the performance of the crawler.

Appropriate approach to handle special structures: Detection schemes are required to handle the three special structures described above. If the comparison result indicates that the current node differs entirely from the previous node, it is added to the path as a new node. If the node is of the same type but has different content, it is necessary to verify whether it has the same parent as the previous node of the same type. If they share the same parent, an interconnection exists between the interfaces of the same type, and no new node should be added. If they do not share the same parent, the node is added as a new node. If the nodes are identical to one another by comparison, they are tested to determine whether they share the same parent or whether they are themselves parents. If they share the same parent, this operation is considered invalid (there is no change after interaction) and no new node is added. If they have different parents, there are two ways forward. One, if the identical nodes are in the same sub-graph, a cyclic interconnection exists, and no new node is added. Two, if the identical nodes are in different sub-graphs, they are labeled as a multi-entry interface, a reference node is established at the current location, and the previous node identical to the current node is pointed towards it.

Interaction count limitations: Some applications have a special list layout called a "waterfall layout". This enables users to view more content by scrolling down infinitely with the contents being loaded asynchronously. For this type of layout, our IGG algorithm has options to limit the maximum number of interactions a container (list, table…) can receive.

Algorithm 1: IGG-Simple (***A***, ***depth***, ***dlimit***, ***alimit***)

```
Input:   Entry point activity A;
         Current depth depth;
         Depth limit dlimit;
         Action limit alimit;
Output:  Interaction graph G;

if dlimit <= 0 then
         return;
end if
if !G.hasIdentical(A) then
         if !G.hasSimilar(A) then
                  add A to G;
         end if
else
         return;
end if
stack of controls C;
C = {all interactable elements in A};
while C.size() > 0 and alimit > 0 do
         tc = C.pop();
         tc.interact();
         ca = device.currentActivity();
         if ca != A then
                  IGG-Simple (ca, depth+1, dlimit-1, alimit);
                  device.back();
         alimit -= 1;
end while
```

3.4 Content Retrieval

After generation of the interaction graph, the MCC collects the content along each path based on the recorded interaction sequences. According to the described design, the crawler collects common elements on the application interfaces (e.g., text, images, audio, and video). Text and images for Android applications are usually implemented using common controls (e.g., TextView, ImageView, ImageButton), making it easy to extract them from the interface directly. Audio and video elements may use different presentation or rendering for each application. For these two content types, MCC performs packet inspection to collect the content from network requests generated by the application at the corresponding interfaces.

Text*:* After obtaining interface information from the device, the crawler analyzes and extracts the properties of all available text controls. Most of the text information in the Android text controls is available from the text or content-desc properties.

Images: The image control's properties provide the control's coordinates as well as the size and source of images. Accordingly, MCC first analyzes the image's source. If the images come from local application resources bundled with the application, MCC directly copies them based on the path. Otherwise, MCC obtains the images by capturing the entire screen and cropping the screenshot based on the coordinates and image size.

Audio and video: MCC analyzes the network traffic generated by the application. Because mobile applications usually communicate through HTTP [17], the crawler analyzes these types of network traffic and collects recognizable multimedia content (i.e., audio and video). MCC also attempts to decrypt encrypted HTTPS traffic by replacing original certificates with

self-signed ones. The “MCC CA” used for signing certificates is trusted in the crawler’s Android environment. Therefore, MCC can decrypt and analyze the HTTPS traffic generated by almost all applications except for a small number of applications using special hardcoded certificate checks (“key pinning”).

3.5 Concurrent Retrieval

An interaction graph is divisible into multiple subgraphs, each representing a different path for crawling different content. If we assign different tasks following unique paths to multiple devices with the same initial state, we can retrieve content from multiple interfaces simultaneously. Our current implementation uses a thread pool with a fixed number of threads (configurable by the user) and performs tasks on multiple devices at the same time. It can be further optimized by recording and predicting the time spent on each operation and scheduling the tasks in a more balanced way.

4. Implementations

We implemented MCC for Android to collect content from several real-world applications including WeChat, a messaging and social media application; Toutiao, a news application; and Jike, an information agent and social media application. This section describes our experiment, the results, and relevant technical solutions.

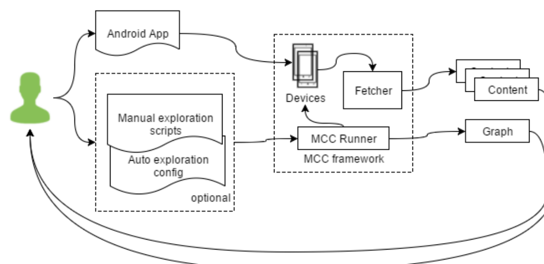


Fig. 4. The experimental flow of the content retrieval tests

4.1 Overview

Fig. 4 describes the architecture and operating process of MCC, which consists of two parts: the interaction path explorer and the content crawler. Both parts run on a desktop computer. The devices that run the applications can be real Android devices or virtual machines or emulators. The crawler installs a daemon application on the devices to enable communication with and remote control by the controlling computer.

MCC deploys the interaction path explorer first in order to determine the structure of the target application and generate the corresponding interaction graph. The explorer traverses and continually analyzes the status of the target application. It then constructs the graph and stores it as a file, which the crawler uses directlyA visualization of the graph is useful for reference and manual adjustments to the crawling rules.

The content crawler gathers information according to the graph. It runs either automatically or manually. In automatic mode, there is no need to write scripts to direct the process. With the graph and necessary device information, , the crawler runs and gathers automatically. The user initiates directed (but still automatic) crawling by pruning the previously generated graph. In

manual mode, the user must create scripts and specify the interaction pattern for the target application. This mode allows advanced users to extract specific information from specific elements in specific interfaces. In both cases, the crawler saves retrieved data as well as screenshots of the pages that appeared on the device in a hierarchical directory structure.

4.2 Environment

The official Android Emulator released by Google in the Android SDK and the Android 6.0.1 x86 Android Open Source Project image (AOSP; akin to a standard Linux distribution) are used as the MCC crawling environment. To protect the environment from potential malicious behaviors in the applications, we preserved a snapshot of the emulator's configured environment, ensuring that the system could be restored to its original state prior to each collection task. An x86 server running Ubuntu 16.04 LTS was used as the crawling controller.

4.3 Content collection

4.3.1 Devices

The automated testing framework UIAutomator was used to direct interaction with the Android devices via the MCC daemon. The daemon binds a TCP port on the controlled device to communicate commands and responses and transmit content over our internal protocol. Each device parses and executes the commands sent from PC. The internal protocol implements the remote procedure call protocol (RPC) wrapped within the JSON data exchange layer. HTTP was used to exchange commands and data between the PC and the controlled device. The daemon's command set includes: retrieval of device information and status, collection of all current interface components, simulation of physical button and sensor and screen events, and screenshot.

We wrote the daemon in Java and compiled it into a normal Android APK application. It is installable on any Android platform (Version 4.3 or above) that has the necessary APIs.

4.3.2 Traffic Analysis

As discussed above, we extract audio and video content in applications from network traffic. Because we are using an Android Emulator on the PC platform, the host operating system handles all network traffic generated by applications in the emulator. Our crawler uses libpcap [18], a system library designed to capture network traffic, to implement real-time packet capture, analysis, and manipulation. The crawler extracts image, audio, and video traffic transmitted via HTTP or HTTPS and saves it into files.

4.3.3 Controller

The daemons implement features such as content and status retrieval and event simulation using only abstracted platform-independent commands. Doing so allows us to limit the controlling server to the core tasks of exploration and content collection. We implemented our controlling server in Python. Each of the crawling tasks generated from an interaction graph is also a Python script that can be run independently. This achieves our requirement to implement the crawling mechanism in a platform-independent way.

4.4 Sample output

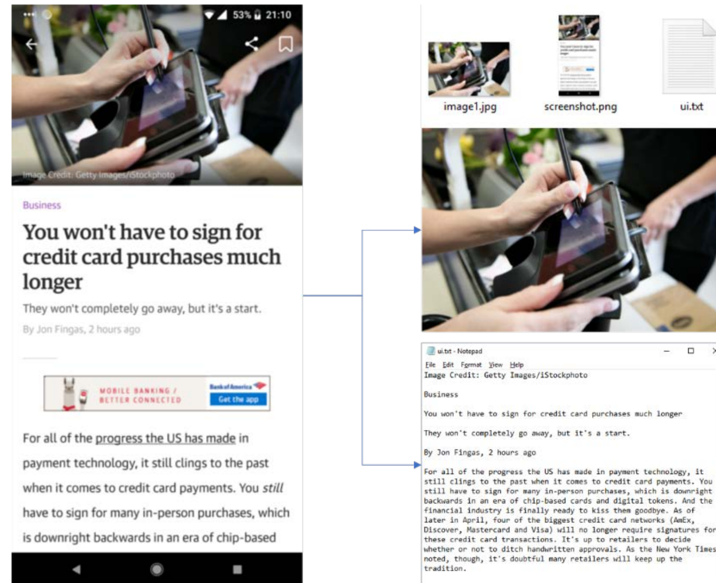


Fig. 5. Output files

Our current implmentation takes a screenshot on each page and saves supported media and text into files as shown in **Fig. 5**.

5. Evaluations

This section reports our evaluation of the function and efficiency of the proposed crawling system. The coverage of collected content is an important performance metric of the crawler; exploration and generation of the interaction graph in MCC is a significant factor influencing its coverage. We first tested the performance of IGG with popular applications and then analyzed the efficiency of the crawler accordingly.

5.1 Samples

We designed MCC for a broad variety of real-world applications. We downloaded the top 50 non-game applications from Google Play covering a range of news, social media, communication, music, video, map, and utility applications as summarized in **Fig. 6**. We ran the fully automated crawler first to generate the interaction graph and collect all the content without any human guidance or manual adjustments.

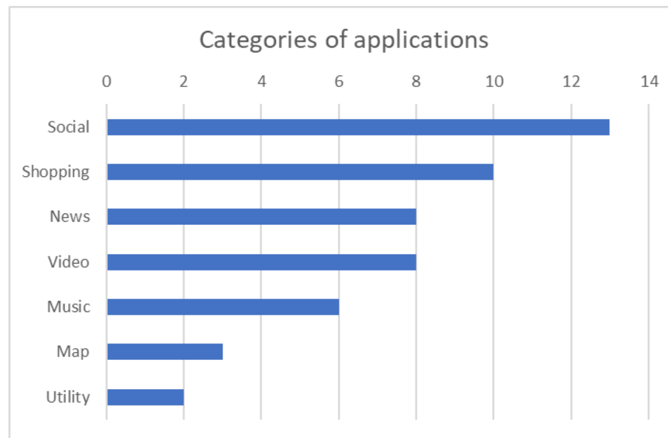


Fig. 6. Application types

5.2 Content Coverage

An activity in any real-world application is capable of representing multiple types of content with the same layout. For a hypothetical instant messaging application with 5 contacts, a single activity is sufficient to show conversations with all of them, while there are 5 pages of different content.

Only activities reachable during the exploration are added to the interaction graph and subsequently collected. Activity reached (AR) is the number of interfaces that are successfully covered by IGG during its execution. Activity defined (AD) is the total number of interfaces designed by the developers in the application. Activity coverage (AC) is defined as the ratio of the interfaces that can be explored by IGG.

$$AC = \frac{AR}{AD}.$$

Due to the limited ways to interact with applications, it is not always possible to reach all the interfaces. A higher AC value indicates that the gathered content represents a greater proportion of the total content. **Table 6** provides a sample of the IGG test results.

Table 6. Coverage of IGG

Application	Total number of Activities	Number of IGG explorations	AC	Types of activities not reached
Twitter	36	16	44.4%	Advertisement/function/setting/sharing
Dropbox	16	7	43.8%	Function/sharing/Inside
Engadget	9	6	66.7%	Advertisement/Sharing/ widget
BBC	9	7	77.8%	Function/Setting
NYTimes	15	7	46.7%	Sharing/Internal
YouTube	18	8	44.4%	Function/Setting/Analysis/Internal/Widget
Average			54.0%	

Significantly, some types of application activities were designed by the application creators to not show up at all. Some of them contain hidden activities for internal debugging, unreleased features, and the like. Launcher widgets that display application information on a user's Android home screen are activities that are never reachable.

We evaluated both Monkey and our own MCC with pages that presented actual content using an execution time of 120 s; **Table 7** provides the results.

Table 7. MCC vs Monkey

Applications	MCC: Number of pages	Monkey: Number of pages	Ratios
Twitter	39	22	177%
Engadget	25	10	250%
Bloomberg	31	16	194%
NYTimes	31	17	182%

5.3 Execution Efficiency

Table 8 and **Fig. 7** compare the total amount of time consumed for exploration and content collection. We also compare the execution efficiency across several devices to that of a single device.

Table 8. Execution time

Application	Visited pages	IGG (s)	Crawling Tasks	1 device (s)	4 devices (s)
Engadget	22	110	5	141	76
Bloomberg	43	267	9	470	186
NYTimes	29	116	4	244	153

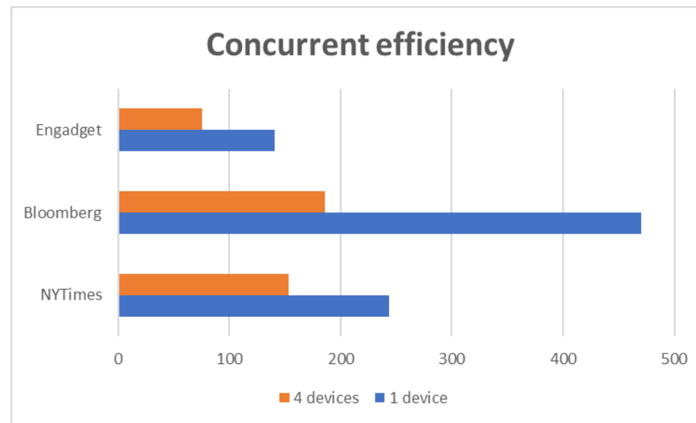


Fig. 7. Parallel efficiency

Fig. 7 shows that assigning tasks to multiple devices and executing them concurrently significantly shortened the time required to obtain the same amount of content. The specific UI designs of some target applications (e.g., “next” and “previous” buttons instead of an article list)), prevented large tasks from being broken down effectively, causing some tasks to take significantly longer than others. This explains why some applications showed no reduction in time even with an increase in the number of parallel tasks.

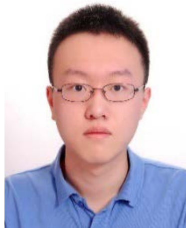
6. Conclusions

Although the mobile internet ecosystem is growing rapidly, the content generated in this ecosystem is often isolated from the open internet. It is trapped in a "walled garden"—hard to access, and even harder to collect with a unified approach. To address this challenge, we drew inspiration from traditional web crawling and analysis techniques. We designed and implemented a program to analyze and crawl mobile application content automatically. We also built a general scheme for us to discover the interaction paths of various mobile applications and to collect their data concurrently. The result shows that MCC can be a feasible solution for both automated and manually assisted mobile application content crawling.

References

[1] Karen Church, Barry Smyth, Paul Cotter and Keith Bradley, "Mobile information access: A study of emerging search behavior on the mobile Internet," *ACM Transactions on the Web (TWEB)*, vol. 1, no. 1, May, 2007. Article (CrossRef Link).
[2] Tanzirul Azim and Iulian Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 641-660, October, 2013. Article (CrossRef Link).
[3] Shuai Hao, Bin Liu, Suman Nath, William G. J. Halfond and Ramesh Govindan, "PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps," in *Proc. of 12th Annu. Int. Conf. on Mobile Systems, Applications, and Services (MobiSys'14)*, pp. 204-217, June 16-19, 2014. Article (CrossRef Link).
[4] Suman Nath, Felix Xiaozhu Lin, Lenin Ravindranath and Jitendra Padhye, "SmartAds: bringing contextual ads to mobile apps," in *Proc. of 12th Annu. Int. Conf. on Mobile Systems, Applications, and Services (MobiSys'13)*, pp. 111-124, June 25-28, 2013. Article (CrossRef Link).
[5] Facebook, "React Native, a framework for building native apps using React." Article (CrossRef Link)
[6] Statista, "Number of apps available in leading app stores as of June 2016." Article (CrossRef Link)
[7] Statista, "Number of smartphone social network users in the United States from 2014 to 2020 (in millions)," 2016. Article (CrossRef Link)
[8] CNNIC, "Statistical Report on China's Internet Development (2016.1)," 2016. Article (CrossRef Link)
[9] Statista, "Mobile Internet-Statistics & Facts." Article (CrossRef Link)
[10] Cisco, "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016–2021 White Paper," 2017. Article (CrossRef Link)
[11] Patton Ron, *Software Testing*, 2nd Editon, Sams Publishing, Indianapolis, 2005.
[12] Ian Goldberg, David Wagner, Randi Thomas and Eric A. Brewer, "A secure environment for untrusted helper applications: Confining the wily hacker," in *Proc. of 6th Conf. on USENIX Security Symposium, Focusing on Applications of Cryptography*, vol. 6, pp. 1, July 22-25, 1996. Article (CrossRef Link)
[13] Eda Baykan, Monika Henzinger and Ingmar Weber, "A Comprehensive Study of Techniques for URL-Based Web Page Language Classification," *ACM Transactions on the Web (TWEB)*, vol. 1, no. 1, article no. 3, March 2013. Article (CrossRef Link).
[14] 4Tanzirul Azim and Iulian Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 641-660. October, 2013. Article (CrossRef Link).
[15] RobotiumTech, "Robotium." Article (CrossRef Link)
[16] International Data Corperation (IDC), "Smartphone OS Market Share," 2016. Article (CrossRef Link)

[17] Ricardo Anacleto, Lino Figueiredo, Ana Almeida and Paulo Novais, "Server to Mobile Device Communication: A Case Study," *Ambient Intelligence-Software and Applications*, vol. 219, pp. 79-86, 2013. Article (CrossRef Link).
[18] Tcpdump & Libpcap, "TCPDUMP/LIBPCAP public repository," Article (CrossRef Link)
[19] Soumen Chakrabarti, Martin Van Den Berg and Byron Dom, "Focused crawling: a new approach to topic-specific Web resource discovery," *Computer Networks*, vol. 31. no. 11-16, pp. 1623–1640, May, 1999. Article (CrossRef Link).
[20] Ziv Bar-Yossef, Alexander Berg, Steve Chien, Jittat Fakcharoenphol and Dror Weitz, "Approximating Aggregate Queries about Web Pages via Random Walks," in *Proc. of 26th Int. Conf. on Very Large Data Bases (VLDB'00)*, pp. 535-544, September 10-14, 2000. Article (CrossRef Link)



Mingyi Huang is an undergraduate student of North China University of Technology, currently participating in research in Tsinghua National Laboratory for Information Science and Technology at Tsinghua University. His research mainly focuses on internet and security technologies.

Yongqiang Lyu, Ph.D., is an associate professor in Tsinghua National Laboratory for Information Science and Technology at Tsinghua University. His research interest focuses on the hardware-software fusion computer systems, including HCI, internet and security technologies. He leads an interdisciplinary team at Tsinghua University working on the innovative cyber-physical technologies and systems from a fusion perspective of computer science, art and medicine. The team also works with several international companies for consumer electronics and healthcare services to make their production applied in emerging applications.

Hao Yin is a Professor in the Research Institute of Information Technology (RIIT) at Tsinghua University. He received the B.S., M.E., and Ph.D. degrees from Huazhong University of Science and Technology, Wuhan, China, in 1996, 1999, and 2002, respectively, all in electrical engineering. He was elected as the New Century Excellent Talent of the Chinese Ministry of Education in 2009, and won the Chinese National Science Foundation for Excellent Young Scholars in 2012. His research interests span broad aspects of Multimedia Communication and Computer Networks.