MODESTUM
OPEN ACCESS

EURASIA Journal of Mathematics, Science and Technology Education, 2018, 14(7), 3159-3179
ISSN:1305-8223 (online) 1305-8215 (print)
Special Issue Paper https://doi.org/10.29333/ejmste/91668

An Automatic UI Interaction Script Generator for Android Applications Using Activity Call Graph Analysis

Yining Liu [1], Shih-Chi Wang [2], Yang Yang [3], Yeh-Cheng Chen [4], Hung-Min Sun [2*]

[1] Guilin University of Electronic Technology, Guilin, Guangxi, CHINA
[2] National Tsing Hua University, Hsinchu, TAIWAN
[3] Fuzhou University, Fuzhou, CHINA
[4] University of California Davis, Davis, California, USA

Received 31 October 2017 ▪ Revised 21 January 2018 ▪ Accepted 8 February 2018

ABSTRACT

As the Android's growth in global market share, the security problem of Android OS becomes more and more serious. According to statistics, there are 84% of smartphone users use Android OS. The popularity brings not only wealth into Android market but also more and more malicious applications. Malicious developers want to steal private information such as credit card number, contacts, or email from Android phones. Android has sustained security issue for a long time. Academics also have put many efforts to solve the problem. Dynamic analysis is one of the methodologies for Android malware detection. Current execution of dynamic analysis needs to deploy heavy human resources. There is always someone needed to access the user interface manually, or the work can hardly be finished. In this work, we propose an approach on Android UI automation. Our implemented system output an Android monkeyrunner scripts, which is custom made for input Apk. The script program can trigger UI event automatically and deal with exception conditions while executed in monkeyrunner.

Keywords: Android, malware detection, automation, activity call graph

INTRODUCTION

Smartphones popularity is growing rapidly over these years. Many people cannot live without a smartphone these days. Because they do all kind of things with their phone, including to read, entertain, socialize and work. A report from International Data Corporation (IDC) shows there is 85 percent global smartphone market share for Android OS till the first quarter of 2017 (Smartphone OS market share, 2017). And there are over 3.5 million applications on Google Play market since December 2017 (Google Play: number of available apps 2009-2017, 2017). These numbers indicate the leading position of Android and an enormous benefit that one can get by successfully spreading malicious apps on the platform.

Android has become a major target for malicious developers because it is widely used. Unfortunately, the openness of market even intensifies the problem. Security issue of Android applications has been criticized for a long time. Android developers can upload applications to market with a few screening process. Besides Google Play, there are still some third-party markets, such as Amazon and Samsung which provide their platform for the user to upload and download Android apps freely.

Motivation

Dynamic analysis on Android application getting more important since dynamic loading technique appears. Yet it is a resource-consuming process without automatic device access. During the inspection, there should be a human operator taking care of the process. Additionally, Android developer creates apps with a stunning speed.

✉ ynliu@guet.edu.cn ✉ hmsun002@yahoo.com ✉ yang.yang.research@gmail.com
✉ ycch@ucdavis.edu ✉ hmsun@cs.nthu.edu.tw **(*Correspondence)**

Contribution of this paper to the literature

- An approach can go through every version of Android OS and different environments.
- A system is build up specifically in the light of each input Apk analysis result.
- This collaborative model can extract information from Apk files and generate automatic UI scripts for the well-known tool "Monkeyrunner" to perform dynamic analysis in an automated manner.

Without automation, dynamic analysis can never catch up the upload rate of apps. That means screening process of new apps will eventually fail. Google deploys Bouncer (Hou, 2012) on Play market somehow supports our argument. The Bouncer basically is an automatic dynamic analysis tool. They do not want people involved in this vetting work. Although some malicious apps can still bypass Google Bouncer. We can still catch the concept and working on it in academics.

Someone uses Monkey, a semi-random UI event tool, to fulfill the job. We can question the result of taking a random UI event trigger. Because the control flow of an application can be very complex. Monkey may miss some corner of the whole picture.

In this work, we implement a system that provides an UI automation mechanism for Android applications. The system deploys Activity Call Graph analysis that ensures we reach every part of the application. Considering that exceptions may happen while executing. Our system provides alter steps for dealing with the unexpected situation.

Organization. In Section 2, we will get an overview of Android security related topic. Section 3 describes the concept of our proposing framework. Section 4 shows our implementation detail and result. At the last, we summarize this work in Section 6.

BACKGROUND AND RELATED WORKS

Android Security

Android is a Linux-based opensource operating system released by Google. Although most of the codes of Android are implemented in Java, they transform into Dalvik executable code and executes on Dalvik virtual machine (Android – Wikipedia, n.d.). The overall framework of Android is shown in **Figure 1** (Smieh, 2012). Google's policy on the openness of Android provides not only IDE tools and SDK but also the source code of Android operating system, giving developers least constraint playing in the ground. Google has successfully gathered a developer community that has never been seen. Yet they open the security net and cannot guarantee safety in the field.

Because of the popularity of Android phones, there are always malicious developers who exploit apps to steal users personal information. Credit card and contact book are major targets. Once a malicious app steals this information, they send back the data through SMS or internet. The competition between malicious apps and Android security system has been going for a long time. Android security system deploys permission analysis to break the exploit of message sending while the attacker then brings code transformation to avoid the detection. Many researches devote their contribution to Android security (Baskaran & Ralescu, 2016), yet the battle is still severe.

A lot of malware detection technique and tools has been proposed to counter this rapidly growing malicious application. But due to malicious application keep evolving, some of those technique such as traditional signature-based detection system become deprecated. There is an increasing need for alternative malware detection system to complement and rectify those traditional signature-based system. Because of the reason above, Android UI-based detection system starting to gain some popularity since it can detect the code-transforming malware with advance dynamic loading that is otherwise would be imposible for static detection such as signature-based detection to detect.

Android App Structure

Apk is an Android application archive file consisting folders and files as shown in **Figure 2** (Faruki et al., 2014). AndroidManifest.xml stores the meta-data including package name, permissions used, definitions of Activities(screen controller object), Services, Broadcast Receivers or Content Providers, supported version and libraries used. There are files of screen layout, images, icons, animations stored in the path /res/. Assets folder contains resources without compile. The execution code is compiled into Dalvik bytecode and stored in .dex files. META-INF have the signature of the app which is the third party developer identity.

There are several tools available for disassembling Apk files online such as Apktool (Apktool, 2017) and Androguard (Androguard, n.d.). So, everyone can easily unpack an Apk file and retrieve data from it. It is convenient for people to employ those tools to help them doing research. But it can also be exploited by anyone with simple unknown code insertion and repackage the archive back.

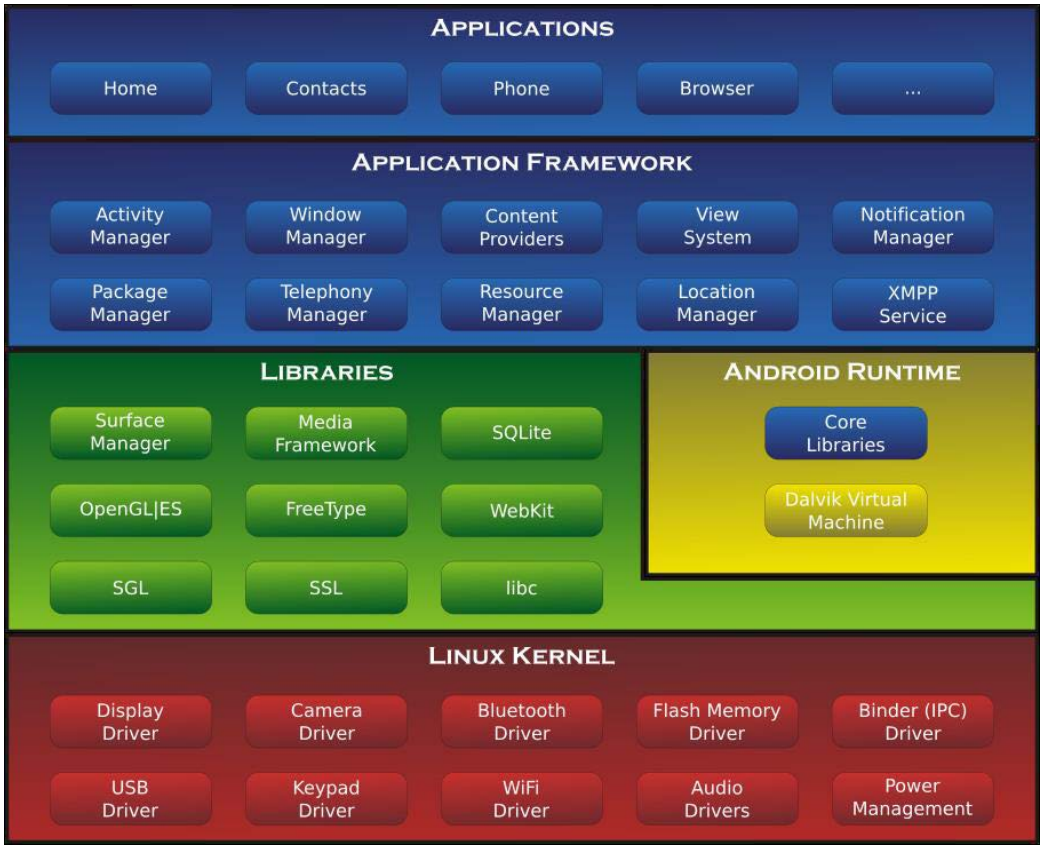

Figure 1. Android OS architecture

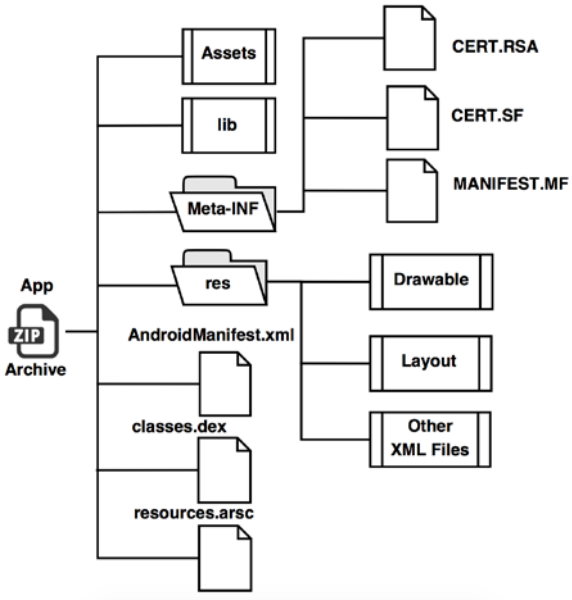

Figure 2. Android package (Apk) structure

Static Analysis

To detect malware from Android applications, static analysis is one basic way to get started. Static analysis means to inspect Android applications without executing it. One may disassemble and decompile the Apk file. Then look into permissions and other meta-data to find out if it is malicious. For more complex mechanism, there are approaches that extract signatures the compare the difference between benign and malicious apps.

Using graph analysis in static analysis is getting more interest in academics recently. In general, when it comes to take graph analysis in software, we make function as nodes, function call relationship as edges. Then we build up a function call graph. MIGDroid (Hu et al., 2014) is a system that can tell if an Apk is repackaged by analyzing its method invocation graph. They use the property of insertion code are usually apart from original code part. There is little interaction between these two division so MIGDroid makes decision under the standard.

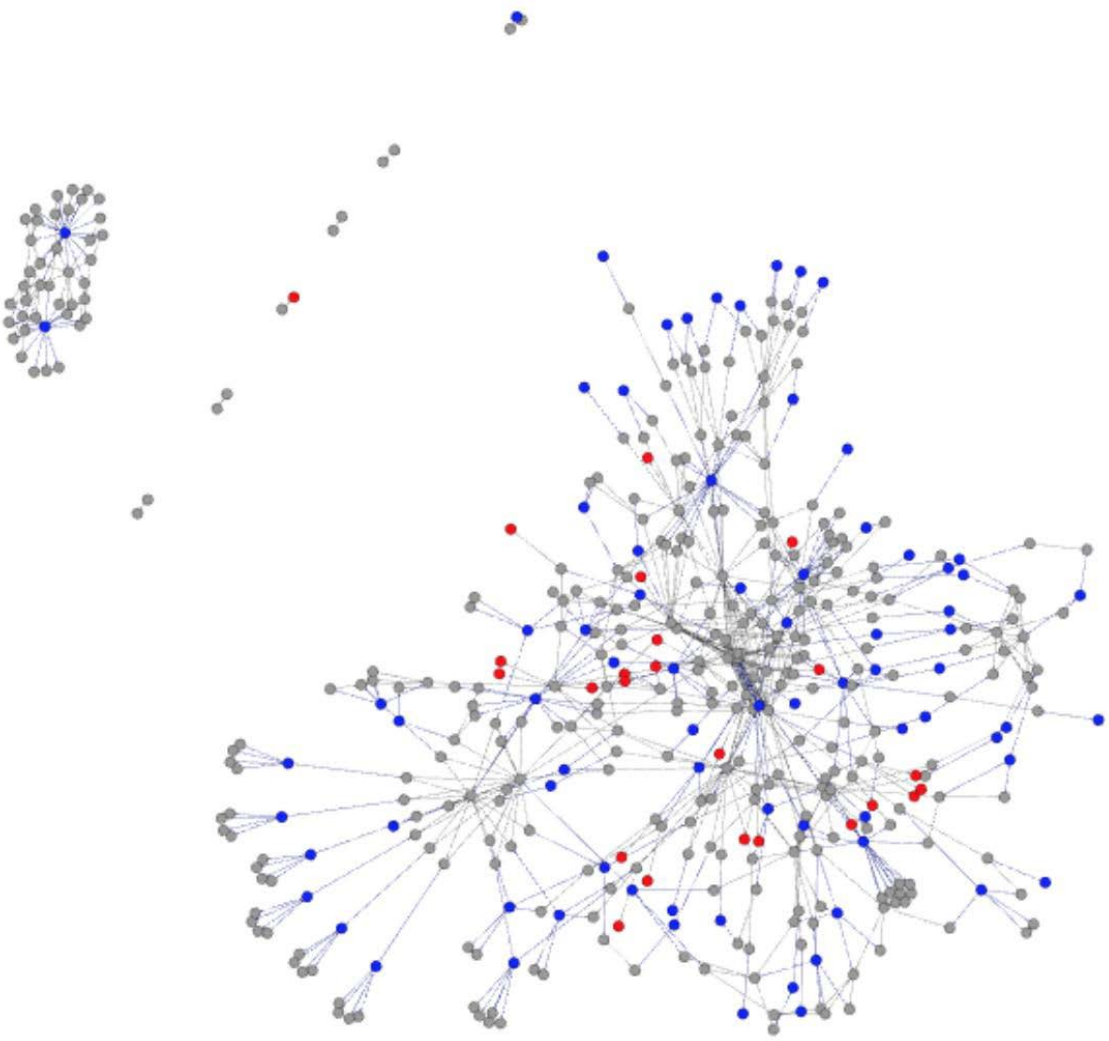

Figure 3. MIGDroid analysis Apk by method invocation graphs

Static analysis can perform a fast malicious detecting since there is no need to execute the program, but it would suffer from code transformation. It is found out vulnerable to advanced dynamic payload techniques. Researcher, lately, tend to deploy dynamic approach or hybrid approach for Android malicious app detection.

Dynamic Analysis

As long as mobile malware detecting technique improves, Android malware becomes more complex and use advanced dynamic loading to evade static detection such as Java reflection or native code execution. There will be necessary employing dynamic analysis to detect those evasion tricks. Research indicates that a lot portion (32.8%)

of apps downloaded from Google Play include dynamic loading behavior (Zheng, Sun, & Lui, 2014). Dynamic loading means a program can load libraries into memory at runtime, and jump to that address to execute functions in the dynamic libraries. In an explicit way, we can call it transformation of the program. It does not mean that every application with dynamic loading is malicious, but there surely some code we cannot retrieve under static analysis mechanism. Actually, dynamic payloads have become a tool that is used to hide malicious code in Android apps. As a result, inspecting the app while in execution is necessary to find out malicious behavior.

DroidTrace (Zheng et al., 2014) utilizes both static and dynamic analysis approach in its detection. The system made an improvement in detecting code transforming. Furthermore, they use a method called "forward execution" which physically modifies an Apk dex code to make the detection process can be done without manually access. It is noticeable they deploy their own automation mechanism into dynamic analysis, because not many researches work on this issue.

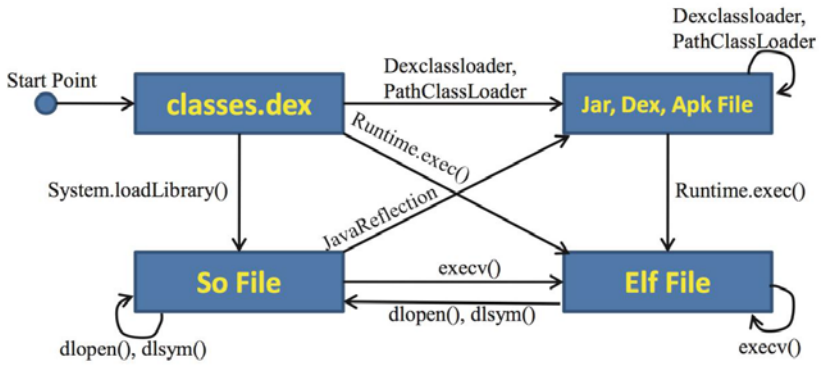

Figure 4. Relationships between dynamic payloads components

Dynamic Analysis Automation

Android applications are UI-based program. Traditionally, there would be manual effort while doing dynamic analysis. Therefore increases a huge amount of cost in the process. To avoid such cost, one may want to deploy automatic technique so that it is possible to verify apps by batches without heavy human efforts. On the other hand, as what we mentioned before, the total amount of apps growth rapidly in Android markets. There is not enough time and resources for a company to manually inspect every new uploaded application. Google has already deployed an automatic dynamic inspector called Bouncer for their Play market. It is necessary to build up automatic process for every dynamic analysis approach.

One of the most popular Android UI automation tools is called *Monkey* (Android monkey, n.d.). Monkey is a built-in tool in Android Debug Bridge (adb) which is provided within Android SDK. Originally, Monkey is made for Android developers to do load testing. It can generate semi-random stream of UI trigger events on Android apps so one can test their own app without human intervention. The benefit of deploying Monkey as automatic tool for dynamic analysis is that it comes along with Android SDK. Users do not have to install other program or environment and it can be called by using adb shell script which is easy to use. There are some concerns while using Monkey to automate dynamic analysis. Because it takes a semi-random stream to trigger UI events, it cannot guarantee ongoing analysis to reach every *Activity* (object type which hold a screen.) Or there can be redundant events which are the same be triggered several times that lower the efficiency of the process. Monkey does not know if the analysis is finished or not. User should set a duration or event trigger count before start using Monkey.

In Section 2.4, we find DroidTrace implement its own UI automation. To establish UI automation, DroidTrace employs repackaging technique to get a modified Apk for inspection. They insert UI trigger code after every UI object is declared. In this way, UI event will be triggered without manual access. Though we can see that it shows it's success in detecting dynamic loading, their automation is too sophisticated. Moreover, the modification will interfere if the UI calling structure was too complicated.

SmartDroid (Zheng et al., 2012) conducts the research that pursuits UI automation as well. However, it takes a lot more effort than DroidTrace. They modify one Android emulator so they can catch API calls inside the framework, also to ensure the application can only step in their designed path. SmartDroid's UI automation approach is wonderful, yet it will suffer from repeatedly modification while emulator updated.

Automated Dynamic Analysis Integrity

In most cases, dynamic analysis will deploy an emulator. Analysts run Android OS and the apps which will be analyzed on this emulator. Then they set up an out-side monitor recording APIs the app has used. That data will be collected and transferred into some factors to help decision making. Under this process, one can figure out the coverage of Android app process path reached during the analysis is critical for data collection and final decision making. Malware, for example, in some cases are repackage apps. Some malicious developers make their malware from repackaged normal apks meanwhile inserting malicious code into it (Hu et al., 2014). Those apps, in this way, will spread faster by taking the fame of original app. Real world example such as malicious Instagram (Zhang, Niu, Wu, Wang, & Xue, 2013) has reached over 30 million people. In this case, there is only few code part involves malicious behavior. Consequently, we need more precise and complete UI trigger to extract its behavior.

SYSTEM DESIGN

To fulfill the task of automatic accessing UI element on Android platform, we choose to generate scripts that can be run on existing Android debug tool. Using such existing tool, our approach can be performed across every version of Android OS and different environments.

Our proposed system, AndroAutoScript, extracts information from Apk files and generates automatic UI script of monkeyrunner to help dynamic analysis automation. Monkeyrunner is a tool coming with Android SDK which provides a programmable interface to run custom scripts that operate with Android applications (Android monkeyrunner, n.d.). We can write a Python program as input to describe a function sequence that operates Android device. Functions such as Install package, touch certain UI component, take screenshots are included. As monkeyrunner possesses a well-defined toolset, we can realize our purpose by using it. The overall framework of AndroAutoScript is shown in **Figure 5**. This framework includes a script generator on the left which deploys Apk analysis methods to build up an UI automation script according to the program structure of input application. In the center is our main output, the UI automation script, which possesses three major functions that is specifically designed for this work. As we familiar with, the script will be the input of a script runner that performs as automatic UI event triggering in a dynamic analysis environment. In following sections, we will go through the idea of our system design.

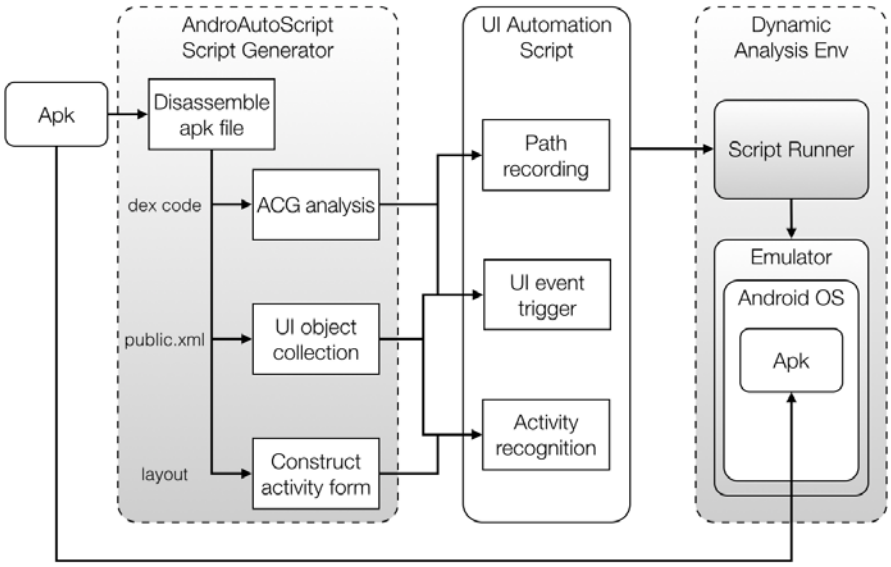

Figure 5. Framework of AndroAutoScript

Script Generator

Apk disassembling

To build up an UI interaction script that can precisely trigger UI components in desired order, we have to, firstly, extract information from target Apk file. By disassembling Apk, we can obtain all information about the

Android application. Smali code contains the program structure. For readable reason, we further decompile smali code into Java. Then we can extract information by using Java parser.

One important information we can learn from disassembled code is the Activity calling structure. An Activity instance, in Android framework, controls a whole hierarchical content within a showing screen. In other words, all UI objects are controlled by Activity class instances. As Android applications and OS itself are UI-based program, Activity Call Graph (ACG) plays no less role than Function Call Graph (FCG) in program control flow analysis. In Android framework, calling of next Activity is completed through Android API. There is function call like *startActivity* or *startActivityForResult* can help the process. In this stage, we further collect the information of UI objects and Activity content by disassembled XML files.

```
Lbedminton/edu/nthu/CompareVideoActivity$1; onClick (Landroid/view/View;)V
    public void onClick(android.view.View p5)
    {
        v1 = new android.content.Intent();
        v1.setClass(this.this$0, bedminton.edu.nthu.CompareListActivity);
        v0 = new android.os.Bundle();
        v0.putString("acquire_up", );
        v1.putExtras(v0);
        this.this$0.startActivityForResult(v1, 0);
        return;
    }
```

Figure 6. Example code - onClick method

Activity call graph analysis

In this phase, we parse the decompiled code and extract the Activity call relationship then we build an Activity Call Graph. ACG provides a structured data that is used to traverse paths and record visited components while running the automatic script. Android OS employ the Intent object to help Activity switches. There is an example code which is shown in **Figure 6**. An Intent instance declares the recipient and transfers data, if needed, and be used as a message for a new started component. When an application is going to start a new Activity, it will create an Intent then invoke *startActivity* or *startActivityForResult* method. Most of the Activity switch are bound with UI event listeners. Common application usages like use a tab to switch between different screen holder, or tap a button to go to next page. To construct the ACG, we can focus on these certain code parts. Then we can record both the calling relationship and UI object at this stage.

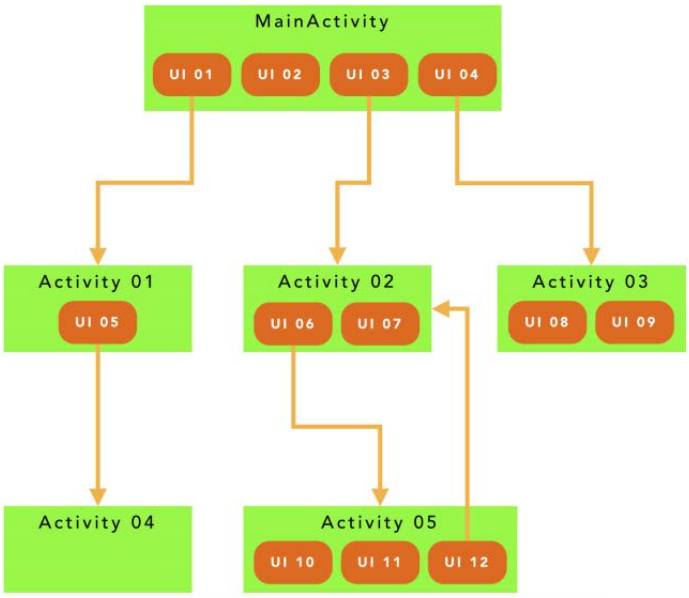

Figure 7. Activity call graph diagram

UI object list

After disassembling, we can obtain additional data sources which are the inner data of the application. Android application uses XML files as storage or register data to help the data access, such as AndroidManifest, Activity layout, and UI strings. These files are archived and encoded within an Apk file. If we use apktool (Apktool, 2017), an open disassembling tool for Apk file, to decode an Apk, there would be a XML file named "public.xml" under the path "/res/values/". Under /res/, an Apk keeps its resource data files. The file is used to store UI objects' type, id, and name mapping. Some of these saved objects are interact-able, others are not. We can find those interaction UI object by matching public.xml with the decompiled code of the application. While, in an Android application, programmers set up a UI interaction component, they call "set some interaction listener" method such as *buttonA.onClickListener*. Following the clued, we can trace back that *buttonA* has been set up an id before. That id will be found corresponding to public.xml and it is how this job be done. In this function, we collect all UI Objects those are matched by above approach and record their higher level class to pass to script as reference.

Table 1. Example of public.xml content

```
<public type="id" name="button_cp_up_play" id="0x7f050000" />
<public type="id" name="button_cp_up_pause" id="0x7f050001" />
<public type="id" name="button_cp_up_stop" id="0x7f050002" />
<public type="id" name="button_cp_up_choose" id="0x7f050003" />
<public type="id" name="surfaceView_cp_up_demo" id="0x7f050004" />
<public type="id" name="surfaceView_cp_down_student" id="0x7f050005" />
<public type="id" name="button_cp_down_play" id="0x7f050006" />
<public type="id" name="button_cp_down_pause" id="0x7f050007" />
<public type="id" name="button_cp_down_stop" id="0x7f050008" />
<public type="id" name="button_cp_down_choose" id="0x7f050009" />
<public type="id" name="imageView1" id="0x7f05000a" />
<public type="id" name="textView1" id="0x7f05000b" />
<public type="id" name="ButtonActiveDemo" id="0x7f05000c" />
<public type="id" name="ButtonActiveRecord" id="0x7f05000d" />
<public type="id" name="ButtonActivePlay" id="0x7f05000e" />
<public type="id" name="ButtonActiveCompare" id="0x7f05000f" />
```

```
Lbedminton/edu/nthu/MenuActivity; onCreate (Landroid/os/Bundle;)V
    public void onCreate(android.os.Bundle p3)
    {
        super.onCreate(p3);
        this.setContentView("0x7f030001");
        this.setRequestedOrientation(1);
        this.ButtonActiveDemo = this.findViewById("0x7f05000c");
        this.ButtonActivePlay = this.findViewById("0x7f05000e");
        this.ButtonActiveRecord = this.findViewById("0x7f05000d");
        this.ButtonActiveCompare = this.findViewById("0x7f05000f");
        this.ButtonActiveDemo.setOnClickListener(new bedminton.edu.nthu.MenuActivity$1(this));
        this.ButtonActivePlay.setOnClickListener(new bedminton.edu.nthu.MenuActivity$2(this));
        this.ButtonActiveRecord.setOnClickListener(new bedminton.edu.nthu.MenuActivity$3(this));
        this.ButtonActiveCompare.setOnClickListener(new bedminton.edu.nthu.MenuActivity$4(this));
        return;
    }
```

Figure 8. Example code - UI listener set up

Activity form construction

Besides public.xml, our approach utilizes other resource data under /res/. We can find layout files of each screen holder, or *Activity* instance, under /res/layout/. A layout file describes how each *Activity* deploys and displays elements on mobile screen. There are several *Activity* class, generally, in an Android application to control different screen. According to these layout files, we can mark up calling edges of ACG which represent *Activity* call trigger as well as an UI interaction object. These data provide more complete information of one *Activity* node on ACG.

Another reason we retrieve layout files is to provide our script the ability to recognize current screen. In this stage, we divide UI object collection we obtain before into different *Activity*. Before this process, a script program has only one set of UI elements that cannot help it to determine which screen is showing. Because we want to provide more flexibility to our script, we build up auto-recognition mechanism in it so the script can choose next step by itself if the path went out of our expectation. With Activity form store in script, it can then record which component has been triggered or not. The Activity form plays a major role in path automation.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout android:gravity="center" android:orientation="horizontal" android:background="#ff42b7ff" android:layout_width="fill_parent" android:
layout_height="fill_parent"
  xmlns:android="http://schemas.android.com/apk/res/android">
    <LinearLayout android:gravity="center_horizontal" android:orientation="vertical" android:layout_width="wrap_content" android:layout_height="fill_parent
    " android:layout_weight="0.47">
        <Space android:layout_width="fill_parent" android:layout_height="150.0dip" />
        <ImageView android:id="@id/imageView1" android:layout_width="242.0dip" android:layout_height="253.0dip" android:src="@drawable/load_material_t2" />
        <TextView android:textAppearance="?android:textAppearanceLarge" android:textSize="50.0dip" android:id="@id/textView1" android:layout_width="
        wrap_content" android:layout_height="wrap_content" android:text="@string/text_system" />
        <LinearLayout android:gravity="center" android:layout_width="fill_parent" android:layout_height="250.0dip">
            <Button android:id="@id/ButtonActiveDemo" android:background="@drawable/ic_button1" android:layout_width="242.0dip" android:layout_height="150.
            0dip" />
            <Space android:layout_width="40.0dip" android:layout_height="fill_parent" />
            <Button android:id="@id/ButtonActiveRecord" android:background="@drawable/ic_button2" android:layout_width="242.0dip" android:layout_height="
            150.0dip" />
        </LinearLayout>
        <LinearLayout android:gravity="center" android:layout_width="fill_parent" android:layout_height="250.0dip">
            <Button android:id="@id/ButtonActivePlay" android:background="@drawable/ic_button3" android:layout_width="242.0dip" android:layout_height="150.
            0dip" />
            <Space android:layout_width="40.0dip" android:layout_height="fill_parent" />
            <Button android:id="@id/ButtonActiveCompare" android:background="@drawable/ic_button4" android:layout_width="242.0dip" android:layout_height="
            150.0dip" />
        </LinearLayout>
    </LinearLayout>
</LinearLayout>
```

Figure 9. Example of layout XML file

UI Automation Script

The final output of our purposed system, AndroAutoScript, is an UI automation script build up specifically according to each input Apk analysis result. The goal of the script program, as what we mentioned before, is to run automatic UI manipulation for an Apk and we want it to trigger as more UI component as possible. To visit every *Activity*, so, is necessary. We design the output of script and make the program be able to make a decision about next step. Considering there are plenty of event-driven code in Android applications, we may easily lose some connection when rebuilding its' control flow. It may cause unexpected steps when running an UI interaction script. If a straightforward script program took a wrong step that brings up unexpected Activity, it may then want to trigger next UI which does not even exist on the screen. For this reason, except automatic UI event triggering, we deploy features of Activity recognition and path recording in our output scripts.

The implementation of UI event trigger can be easily completed through monkeyrunner API. Digging in Apk file code and its resource files, we can get the set of UI objects and separate those are set interaction from static UI. There are objects' ids in hex format stated in the code. We then map these ids to its' Android declaration format, so they can be used in monkeyrunner API. Having ACG in our system, we set up UI event triggering with a pre-ordered sequence then the script program mainly follows the sequence and alters some steps if an exception occurred.

To deal with exception steps, we combine data of UI object collection and Activity form in AndroAutoScript to compose recognition list for each Activity. We use python dictionary to implement the recognition lists. With the lists, whenever the script program is going to trigger next UI object, it can determine if the component exists or not. For negative case, the program can match current screen using the recognition lists and choose another UI interaction object on the screen. By doing this we can ensure the program will not stop while exception.

Table 2. Implementation environment

AndroAutoScript	Dynamic Analysis Env
Ubuntu 14.04	Android SDK r24.1.2
Python 2.7.6	monkeyrunner
Androguard 1.9	Genymotion Emulator

Efficiency is another issue we care about. We want to provide an UI access method that can prevent repeat event especially when an exception happens. The recognition lists are used to record visited path as well. Once an interaction UI is triggered, the script program set a visit flag on its item in recognition list. Next time it comes to the same screen it will take some event has not been visited before. We state the implementation details in Section 4.

IMPLEMENTATION

In this section, we describe how we implemented AndroAutoScript in detail. Our goal is to build up an automatic script for Android application access. When the script program is run, the script runner should retrieve Activity nodes and trigger UI events as much as possible. Also, the program should possess the ability to make its own decision while encounters some unexpected steps.

Environment and Tools

Script generator

AndroAutoScript: AndroAutoScript is implemented in a mixture way with a shell script and python programs. We write a shell script in Ubuntu 14.04 operating system. The shell script automatically takes the apk file under the selected path to run the process. The script generator is a python program, and our python version is 2.7.6. We utilize several classes from an open-source tool, Androguard (version 1.9), and modify some of them to fit our system goal.

Script runner and emulator

We choose monkeyrunner, a tool belongs to Android SDK r24.1.2 for Linux, as our target script executor so that we can take advantages of the compatibility of monkeyrunner with Android framework and its well defined APIs. To run Android OS, Genymotion (Genymotion emulators, n.d.) emulator (version 2.4.0) is put up for verification purpose. Genymotion provides a fast virtual machine accomplished on the base of Oracle VirtualBox.

AndroAutoScript

As **Figure 10** shows, once AndroAutoScipt receives an Apk input, it will, firstly, extract this Apk's data of disassembling code and XML resources files. We import python objects that decode Apk files and analyses Dalvik VMs from Androguard (Androguard, n.d.) to help the process. With these informations, we can go on analyzing ACG.

Graph analysis

For dealing control flow graph with Android applications, we can apply Function Call Graph (FCG) approach at the beginning. We go through the code and make each function/method a node, function call as directed edge to build up its FCG. Therefore, we get a directed graph which contains several distributed subgraph like **Figure 11**.

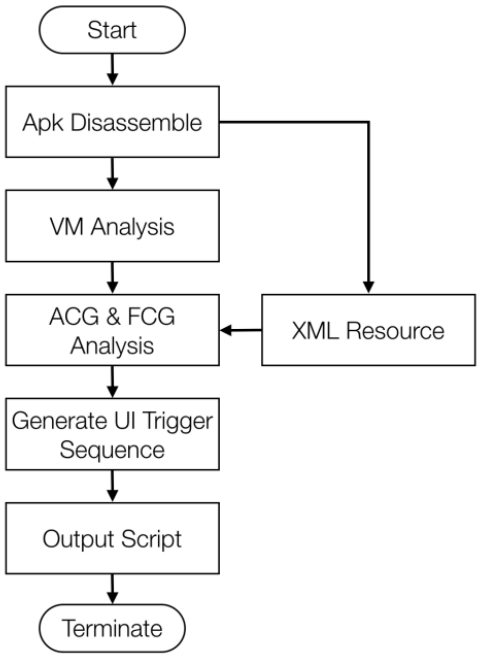

Figure 10. AndroAutoScript Work Flow

Green nodes indicate Activities' onCreate() methods and Brown ones are UI listener methods. We arrange the graph manually to show the direction of control flow in a top-to-bottom fashion. It can easily be found that there are separations before every UI event listener. Although we align UI listeners with their own object initializing methods, these connections cannot be traced through a function call. Because once an UI event listener is set, Android will take care its trigger calls. Bringing up *Activities* is another implicit call in Android. One may use

startActivity() or *startActivityForResult()*method to start a new activity in Android application program. We need to build up these connections before we use utilize control flow to establish a script generator.

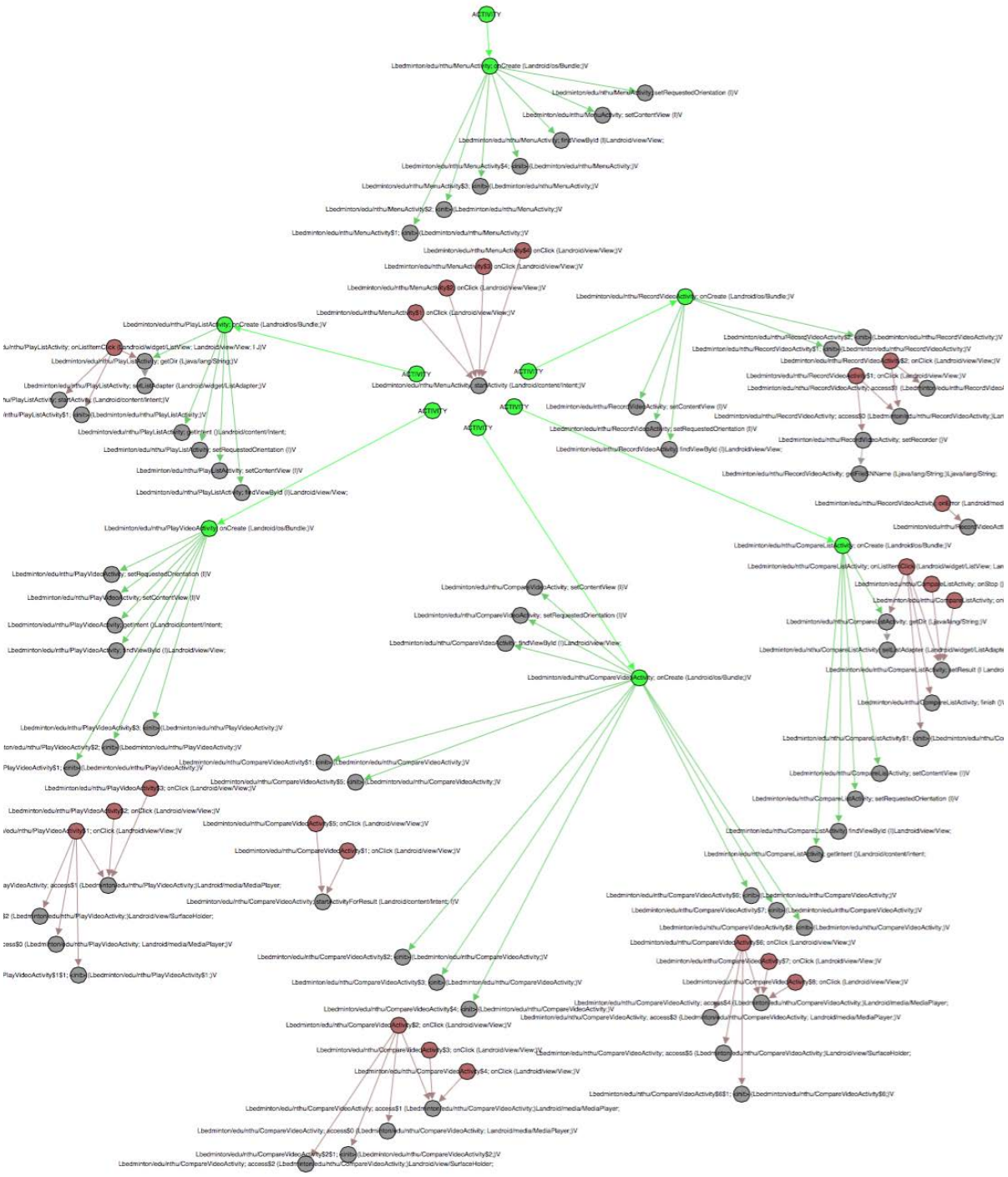

Figure 11. Android application function call graph

Digging into Apk code, we can find our desired information. We focus on code where declares UI listeners and function of on-event-triggered part. UI listener declaration contains instance name of the new-declared listener object. In our sample, such declaration could be this.ButtonActiveDemo.setOnClickListener(new bedminton.edu.nthu.MenuActivity$1(this));. Here UI listener instance name is "bedminton.edu.nthu.MenuActivity$1" and it is named by Android compiler's rule. The rule is that every listener declaration will be named with prefix of its caller class package name. That means this listener, which is a button

listener, is declared in an Activity named MenuActivity. Having the object name, we can bind up the initial node with on-event-triggered node.

```
390     def trace_node_by_UI_code(self, method, sourceCode):      # Steve
391         actvyList = []
392         if "onClick" in method.get_name():
393             self.append_trace_list(sourceCode, actvyList)
394         if "onListItemClick" in method.get_name():
395             self.append_trace_list(sourceCode, actvyList)
396 
397         if len(actvyList) > 0:
398             # print len(actvyList)
399             return actvyList
400         else:
401             return None
402 
403     def append_trace_list(self, sourceCode, actvyList):
404         if "startActivity" in sourceCode:
405                 # print sourceCode
406                 varNames = self.find_variable_of_startActivity(sourceCode)
407                 # print varNames
408                 for varName in varNames:
409                     classNames = self.find_className_by_variableName(varName, sourceCode)
410                     for className in classNames:
411                         classInfo = className, "onCreate", "(Landroid/os/Bundle;)V"
412                         actvyList.append(classInfo)
413 
414     def find_variable_of_startActivity(self, sourceCode):   # Steve
415         variableNames = []
416         for line in sourceCode.split('\n'):
417             if "startActivity(" in line:
418                 params = self.extract_params_from_method("startActivity(", line)
419                 variableName = params[0]
420                 if (variableName != None) & (variableName not in variableNames):
421                     # print variableName
422                     variableNames.append(variableName)
423             if "startActivityForResult(" in line:
424                 # print line
425                 params = self.extract_params_from_method("startActivityForResult(", line)
426                 variableName = params[0]
427                 if (variableName != None) & (variableName not in variableNames):
428                     # print variableName
429                     variableNames.append(variableName)
430         # print variableNames
431         return variableNames
```

Figure 12. Function that deal with startActivity()

There is some additional information we can get from this part of process. We can find id of those UI variable in code before setting up listeners. For the same example, line of "this.ButtonActiveDemo = this.findViewById("0x7f05000c");" is found stated in front of the previous code. A hexadecimal id is assigned to this button and registered in the application. This id will be used later while our script has to figure out UI components form screen.

```
323
324     def match_variables(self, sourceCode):
325         tempMatches = dict()
326         # setOnItemClickListener usually don't appear with findViewByID
327         matches = dict()
328         for line in sourceCode.split('\n'):
329             if "findViewById" in line:
330                 params = self.extract_params_single_const("findViewById(", line)
331                 idStr = params[0]
332                 # print "findViewById: ", idStr, line   # debug
333                 idint = 0
334                 try:
335                     idint = int(idStr,16)
336                 except ValueError:
337                     print "Value error: ", idStr
338                     continue
339                 arsc = self.apk.get_android_resources()
340                 idAlian = '/'.join(arsc.get_id(self.apk.get_package(),idint)[0:2])
341                 # print idAlian # debug
342                 var = self.strip_variable_from_top(line, " = ")
343                 if idAlian not in tempMatches.values():
344                     tempMatches[var] = idAlian
345             if "setOnClickListener" in line:
346                 params = self.extract_params_from_method("setOnClickListener(", line)
347                 param = params[0]
348
349                 # ;;; if occurs "setOnClickListener(this)" it should return self class in match
350                 className = self.strip_prefix_and_suffix(param, "new ", "(this)")
351                 LclassName = 'L'+'/'.join(className.split('.'))+';'
352                 # print "setOnClickListener class: ", LclassName, line # debug
353                 var = self.strip_variable_from_top(line, ".setOnClickListener")
354                 if var in tempMatches.keys():
355                     matches[LclassName] = tempMatches[var]
356             if "setOnItemClickListener" in line:
357                 params = self.extract_params_from_method("setOnItemClickListener(", line)
358                 param = params[0]
359                 className = self.strip_prefix_and_suffix(param, "new ", "(this)")
360                 LclassName = 'L'+'/'.join(className.split('.'))+';'
361                 # print "setOnItemClickListener class: ", LclassName, line # debug
362                 var = self.strip_variable_from_top(line, ".setOnItemClickListener")
363                 if var in tempMatches.keys():
364                     matches[LclassName] = tempMatches[var]
365         return matches
```

Figure 13. Function that extracts id

The other connection we need to rebuild is Activity call. If one on-event-triggered method, for instance *onClick()*, will lead the application to another screen, it would call *startActivity()* to do so. A called Activity name will not show directly in the statement of startActivity() method, instead, one should use an *Intent* object to carry Activity name into use. Intent is a class used to bring detail information of system action, like a message pack in Android framework. We can still retrieve called Activity name in line of Intent declaration and then build up the connection between UI object a target Activity. As what we stated above, we modify FCG approach to construct a complete graph that is shown in **Figure 14**. Note that not every UI event listener involves Activity switch. Some of the UI trigger only computational function, and we also get those behavior in the graph. Our output script is provided with complete information of the UI objects.

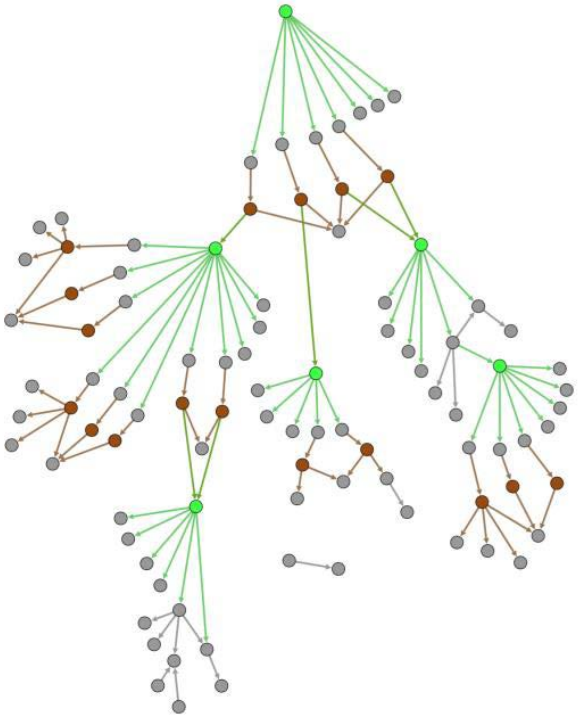

Figure 14. Modification of function call graph on Android application

Now we have finished the Activity call graph. This graph is utilized to form a UI automation sequence in output script. Using deep first traversal algorithm, AndroAutoScript makes an UI-trigger sequence that starts from mainActivity, the beginning of every Android application, and then visits every UI components in ACG.

```
def custom_process_01(self):
    idClickMatches = dict()
    onClickInfo = []
    for method in self.vm.get_methods():
        mx = self.vmx.get_method(method)     # Does it possible get classes? vm.get_classes()?

        if method.get_code() == None:
            continue

        ms = decompile.DvMethod(mx)
        ms.process()
        methodSourceCode = ms.get_source()

        matches = self.match_id_with_onClickListener(method, methodSourceCode)
        # print matches
        if len(matches) > 0:
            idClickMatches.update(matches)

        parentClassName, parentMethod, parentDescriptor = self.extract_ui_parent_method(method)
        if parentMethod != None:
            # print parentClassName, parentMethod, parentDescriptor
            info = method.get_class_name(), method.get_name(), method.get_descriptor()
            onClickInfo.append(info)

            n1 = self._get_exist_node(parentClassName, parentMethod, parentDescriptor)
            n2 = self._get_exist_node(method.get_class_name(), method.get_name(), method.get_descriptor())
            if (n1 != None) & (n2 != None):
                self.G.add_edge(n1.id, n2.id)

        actvyList = self.trace_node_by_UI_code(method,methodSourceCode)
        if actvyList != None:
            # self.print_paths_method(method)
            for actvyClassName, actvyMethod, actvyDescriptor in actvyList:
                n1 = self._get_exist_node(method.get_class_name(), method.get_name(), method.get_descriptor())
                n2 = self._get_exist_node(actvyClassName, actvyMethod, actvyDescriptor)


                if (n1 != None) & (n2 != None):
                    self.G.add_edge(n1.id, n2.id)
                if n1 == None:
                    print method.get_class_name(), method.get_name(), method.get_descriptor(), "[[not found]]"
                    print actvyClassName, actvyMethod, actvyDescriptor
                if n2 == None:
                    print method.get_class_name(), method.get_name(), method.get_descriptor()
```

Figure 15. Function rebuild the connection

XML resources file

Inside XML resources file in Apk, we can extract string id for each UI objects. After be packaged by Android IDE tools, ids in Apk code are transformed into hexadecimal as we stated in Section 4.2.1. We have to convert these ids into string form so they can be used in a monkeyrunner script. All XML resources file are archived in one ARSC format file inside an Apk. We then access the file by an ARSC parser. **Table 1** shows an example of public.xml file. It contains the mapping of string and hexadecimal version of UI object id. We implement the transform in 394 AndroAutoScript according to the file.

Android apps also store layout of each Activity in XML files. Layout files describe hierarchical structure of each screen that has been used. AndroAutoScript read these file to divide UI objects from different Activities. Output scripts will have a set of UI list sorted by Activities. This information is written into python dictionary form in scripts.

```
activityUIdic = {\
"MenuActivity":{"id/ButtonActiveRecord":None, "id/ButtonActiveCompare":None, "id/ButtonActiveDemo":None},\
"PlayVideoActivity":{"id/buttonPlay":None,"id/buttonPause":None,"id/buttonStop":None},\
"RecordVideoActivity":{"id/record_start":None,"id/record_stop":None},\
"CompareVideoActivity":{"id/button_cp_up_play":None,"id/button_cp_up_pause":None, "id/button_cp_up_stop":None,
"PlayListActivity":{"@android:id/list":None},\
"CompareListActivity":{"@android:id/list":None},\
}
```

Figure 16. Activity form in script

Script Program

The final output of AndroAutoScript is a python code file which is an executable monkeyrunner script program. Considering unexpected events may occur, we design the script with a more flexible process. Beside a UI automation sequence that we obtain through ACG analysis in AndroAutoScript, the script has alter steps if something goes wrong. Following the work flow of our script program, **Figure 17**, we explain the details. When the script program start, first, it loads data that AndroAutoScript gives. The data contains a UI trigger sequence which gives basic execution steps for application automation. We store the sequence with python dictionary (**Figure 18**) and put additional attributes such destination and source Activity or visited flag for recording visited path. The script read steps from sequence data in a loop. Every time it gets an UI object id, script program checks if the object is visible on the screen. Monkeyrunner provide a class, EasyMonkeyDevice, from com.android.monkeyrunner.easy package helps this job. It has easy device.visible() method can check there is the certain UI object or not. If the object exists, we can trigger the UI event immediately and set visited flag in sequence.

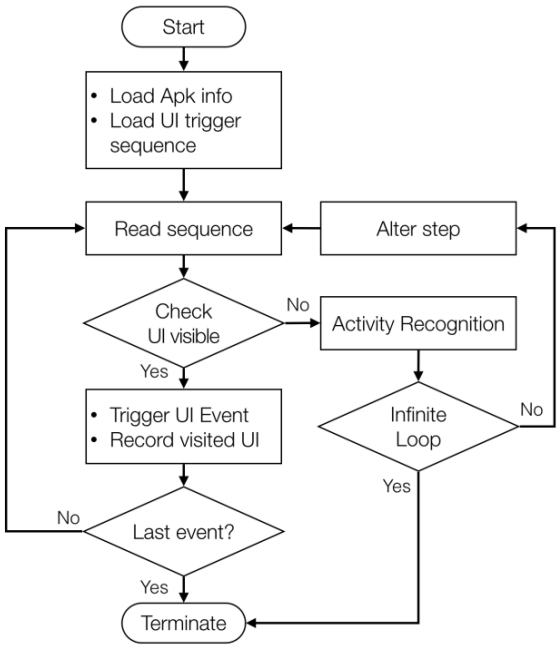

Figure 17. Script Program Work Flow

```
triggerSequence = {\
1 : {"type":"UI", "strID":"id/ButtonActiveRecord", "currentActivity":"MenuActivity", "nextActivity":"RecordVideoActivity", "visited":None},\
2 : {"type":"UI", "strID":"id/record_start", "currentActivity":"RecordVideoActivity", "nextActivity":None, "visited":None},\
3 : {"type":"UI", "strID":"id/record_stop", "currentActivity":"RecordVideoActivity", "nextActivity":None, "visited":None},\
4 : {"type":"dev", "devKey":"KEYCODE_BACK", "currentActivity":"RecordVideoActivity", "nextActivity":"MenuActivity", "visited":None},\
5 : {"type":"UI", "strID":"id/ButtonActiveCompare", "currentActivity":"MenuActivity", "nextActivity":"CompareVideoActivity", "visited":None},\
6 : {"type":"UI", "strID":"id/button_cp_down_choose", "currentActivity":"CompareVideoActivity", "nextActivity":"CompareListActivity", "visited":None},\
7 : {"type":"dev", "devKey":"KEYCODE_BACK", "currentActivity":"CompareListActivity", "nextActivity":"CompareVideoActivity", "visited":None},\
8 : {"type":"UI", "strID":"id/button_cp_down_stop", "currentActivity":"CompareVideoActivity", "nextActivity":None, "visited":None},\
9 : {"type":"UI", "strID":"id/button_cp_down_pause", "currentActivity":"CompareVideoActivity", "nextActivity":None, "visited":None},\
10 : {"type":"UI", "strID":"id/button_cp_up_choose", "currentActivity":"CompareVideoActivity", "nextActivity":"CompareListActivity", "visited":None},\
11 : {"type":"dev", "devKey":"KEYCODE_BACK", "currentActivity":"CompareListActivity", "nextActivity":"CompareVideoActivity", "visited":None},\
12 : {"type":"UI", "strID":"id/button_cp_up_pause", "currentActivity":"CompareVideoActivity", "nextActivity":None, "visited":None},\
13 : {"type":"UI", "strID":"id/button_cp_down_play", "currentActivity":"CompareVideoActivity", "nextActivity":None, "visited":None},\
14 : {"type":"UI", "strID":"id/button_cp_up_stop", "currentActivity":"CompareVideoActivity", "nextActivity":None, "visited":None},\
15 : {"type":"UI", "strID":"id/button_cp_up_play", "currentActivity":"CompareVideoActivity", "nextActivity":None, "visited":None},\
16 : {"type":"dev", "devKey":"KEYCODE_BACK", "currentActivity":"CompareVideoActivity", "nextActivity":"MenuActivity", "visited":None},\
17 : {"type":"UI", "strID":"id/ButtonActivePlay", "currentActivity":"MenuActivity", "nextActivity":"PlayListActivity", "visited":None},\
18 : {"type":"dev", "devKey":"KEYCODE_BACK", "currentActivity":"PlayListActivity", "nextActivity":"MenuActivity", "visited":None},\
19 : {"type":"UI", "strID":"id/ButtonActiveDemo", "currentActivity":"MenuActivity", "nextActivity":"PlayListActivity", "visited":None},\
}
```

Figure 18. Example of UI automation sequence

While false condition, the script will go to alter step branch. It can be a situation that, in our example (**Figure 19**), our script brings up an unexpected Activity or Android application occurs error so comes up with an alert dialog. The script then starts the Activity recognition feature to determine current screen. Then program chooses a step inside the sequence that is not executed and belongs to current Activity to continue the script procedure. In system alert dialog case, we can learn the "ok" or "cancel" button in default UI list. Default UI object normally comes with simple string id such as id/button1 to id/button3. In case the script program keep comes in alter step at same point, we set a threshold execution count. Once the counter goes over the threshold in taking alter step, we terminate the script.

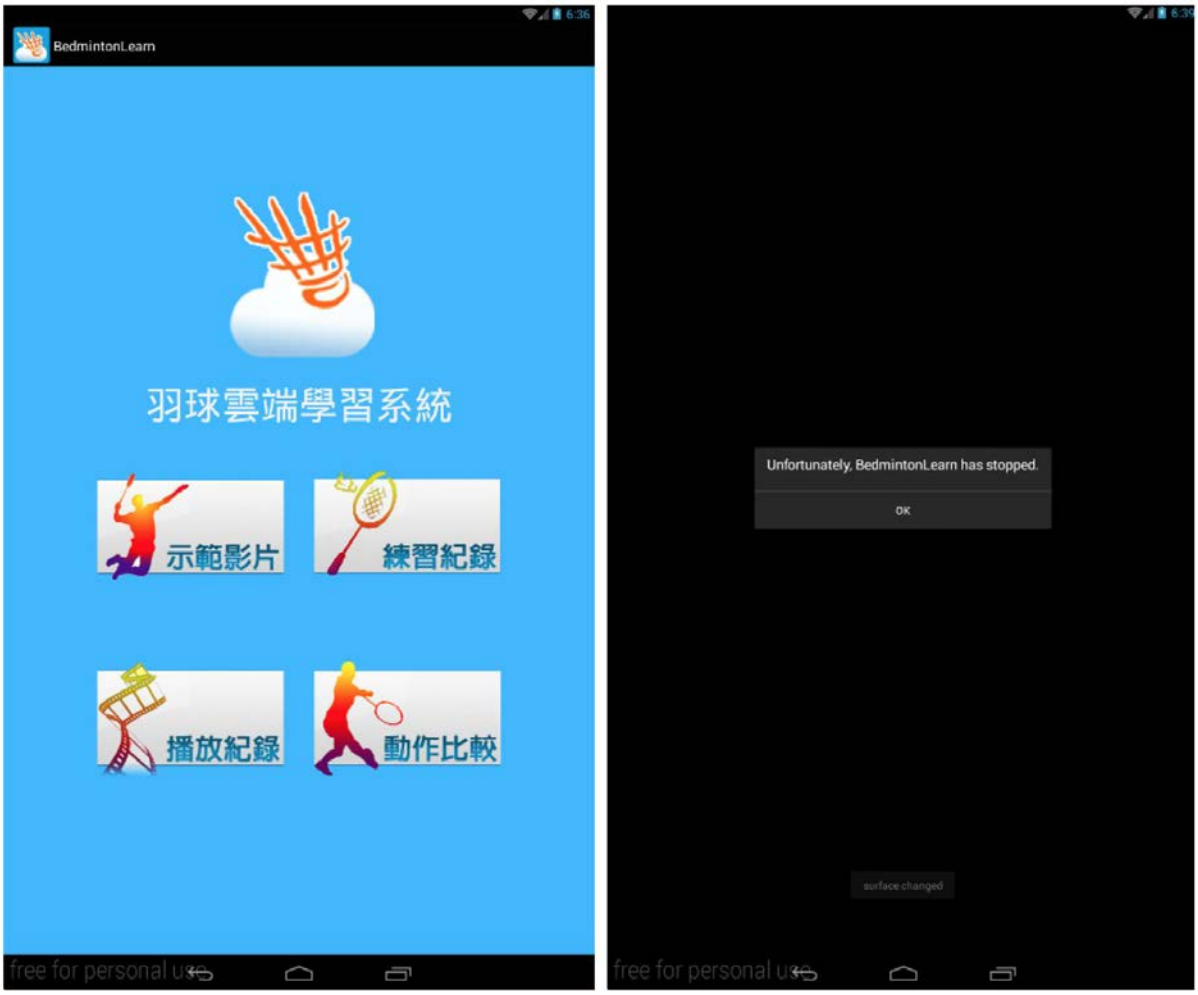

Figure 19. (a) Example application (b) On alert dialog

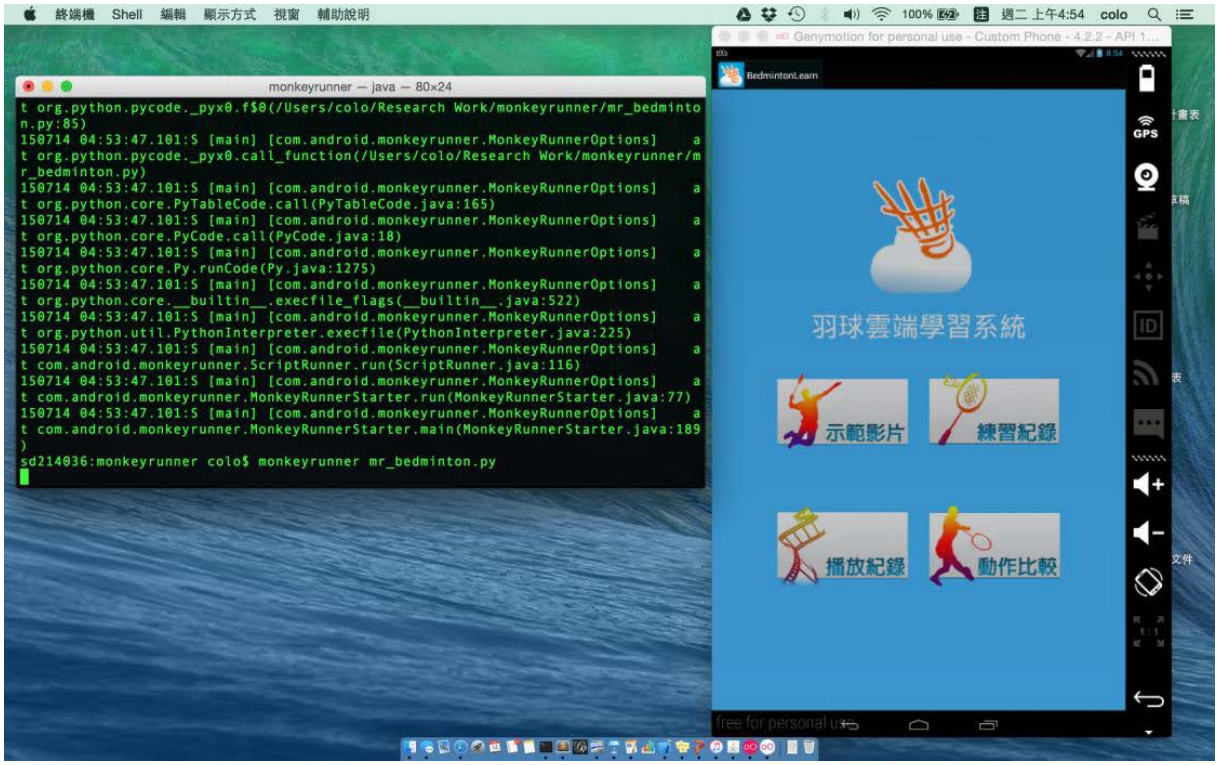

Figure 20. Screenshot of execution (A)

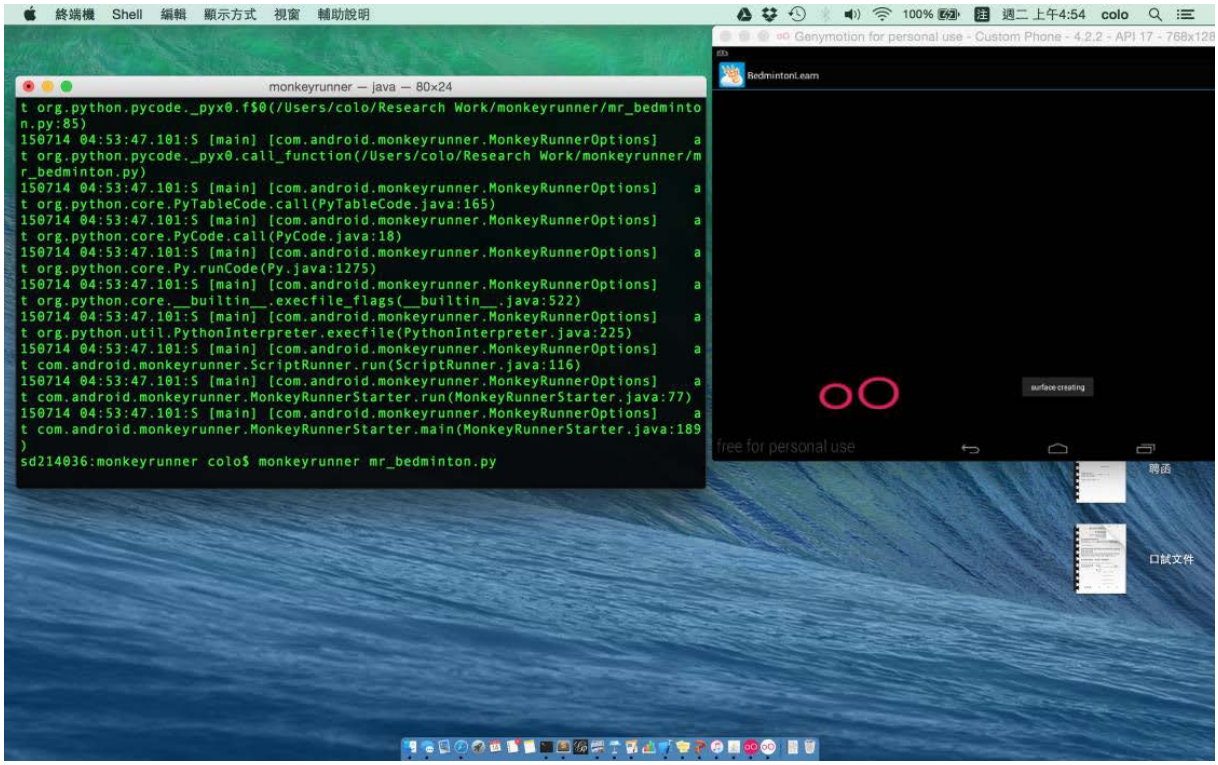

Figure 21. Screenshot of execution (B)

ANALYSIS

In this section, we use an experiment to evaluate our proposed system - AndroAutoScript. Monkey is the most generally used automatic tool that has been deployed in dynamic analysis environment. Our goal is to exam the efficiency of the automatic systems. We use the index of application coverage and the execution time. The example applications have multiple Activities and some of them have complex Activity call structure that can confuse the automation mechanism if the system has no structure information. Because our approach covers limited UI objects, for those UI we cannot deal with we use substitute UI. Most of the Android apps are developed following certain structure in feature of procedure way, we choose ten apps and use substitute UI in new apps to simulate their structure. And we slightly refine our script by each sample for smooth the procedure.

Table 3. Experimental result

Approach/Apps	Monkey		AndroAutoScript	
Coverage	Time		Coverage	Time
Tree-type I	100%	1m 53.192s	100%	5.118s
Tree-type II	100%	1m 43.720s	100%	9.428s
Master-detail-type I	100%	2m 46.668s	100%	5.916s
Master-detail-type II	100%	1m 16.392s	100%	11.968s
Master-detail-type III	89.5%	3m 43.536s	100%	24.644s
Tab-type I	100%	8m 04.460s	100%	14.268s
Tab-type II	77.8%	6m 20.340s	100%	17.856s
Tab-type III	100%	9m 28.752s	100%	31.756s
Loop-type I	76.9%	9m 51.588s	100%	14.214s
Loop-type II	73.2%	22m 43.952s	100%	2m 1.368s

Table 4. Monkey command

adb shell monkey --ignore-crashes --ignore-security-exceptions --ignore-timeouts --pct-touch 70 --pct-syskeys 0 --pct-anyevent 0 --pct-motion 0 --pct-appswitch 0 -v -v -v --throttle 30 -s 2000 –p com.example.test.appalph 50000

We catalog these apps into different types. Tree-type means the app has simple tree process path. Master-detail type apps have a main list on starting interface. Clicking on each list item, the app will show its detail. Tab type obtains a row of tab icons and each one represents its own page while touched. Some apps have process structure too complicated that has loops in its process, we put them into loop type apps. We then compare our generated script executing monkeyrunner with monkey's random stream, the result shows in **Table 3**. Coverage stands the percentage of numbers of Activity that have been displayed during the experiment.

Result

We compare AndroAutoScript with Monkey the Android debug bridge tool which is set to trigger app UI fifty thousand times. AndroAutoScript runs an app and ends once it goes through all nodes in the graph. On the other way, because fifty thousand triggers of Monkey can repeat some actions several times, we record its process time by getting rid of redundant steps.

The result shows our approach can precisely reach each Activity it knows from the app's information and uses a trigger-save way to finish it. While Monkey did fifty thousand triggers for each app, it can be trapped by some sophisticated structure like a loop, and takes longer to reach full visiting.

Case Study

As what we mentioned before, the automatic mechanism on dynamic analysis needs to efficiently reach every corner of an application. We choose one of the loop type apps to show both behavior of Monkey and AndroAutoScript on Activity visiting. Our experiment runs both automation of monkey and AndroAutoScript on the sample app, and record the Activity visit count during execution. The Activity visit count can be on behalf of execution coverage of the app. The more activity we visit the more procedure of program we reach. Our result shows in **Figure 22** and **Figure 23**.

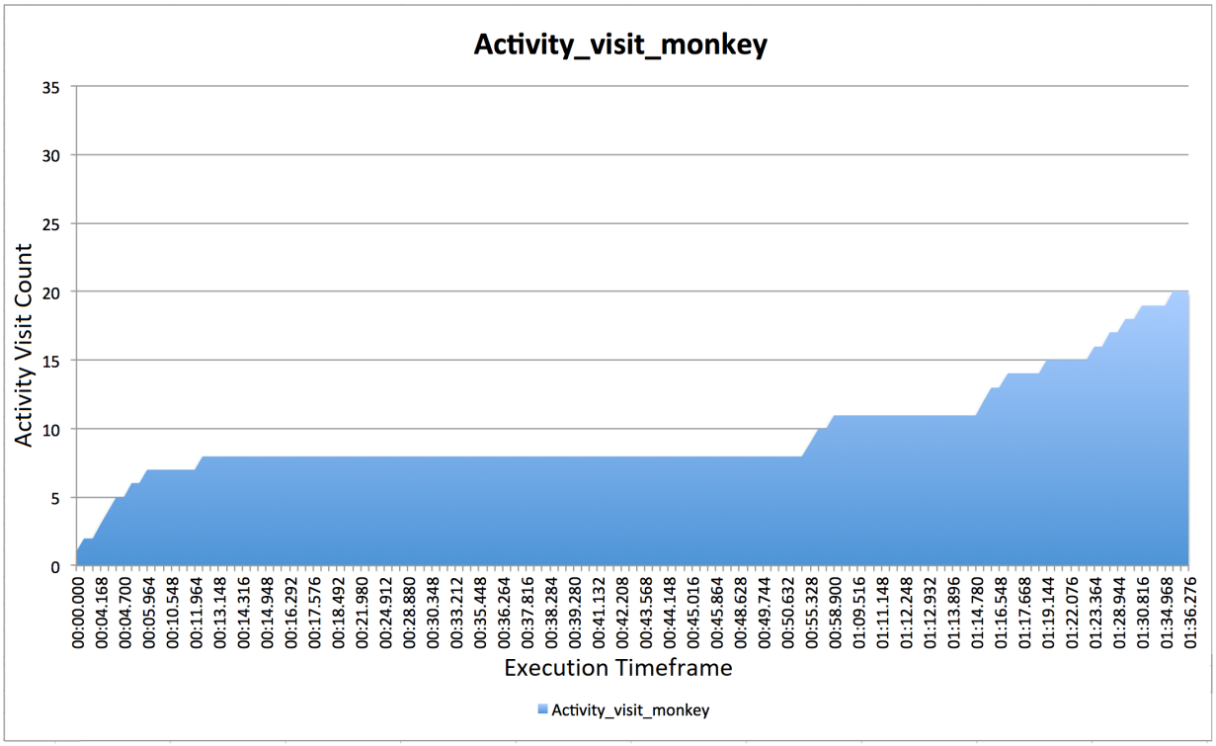

Figure 22. monkey performance on visit Activity

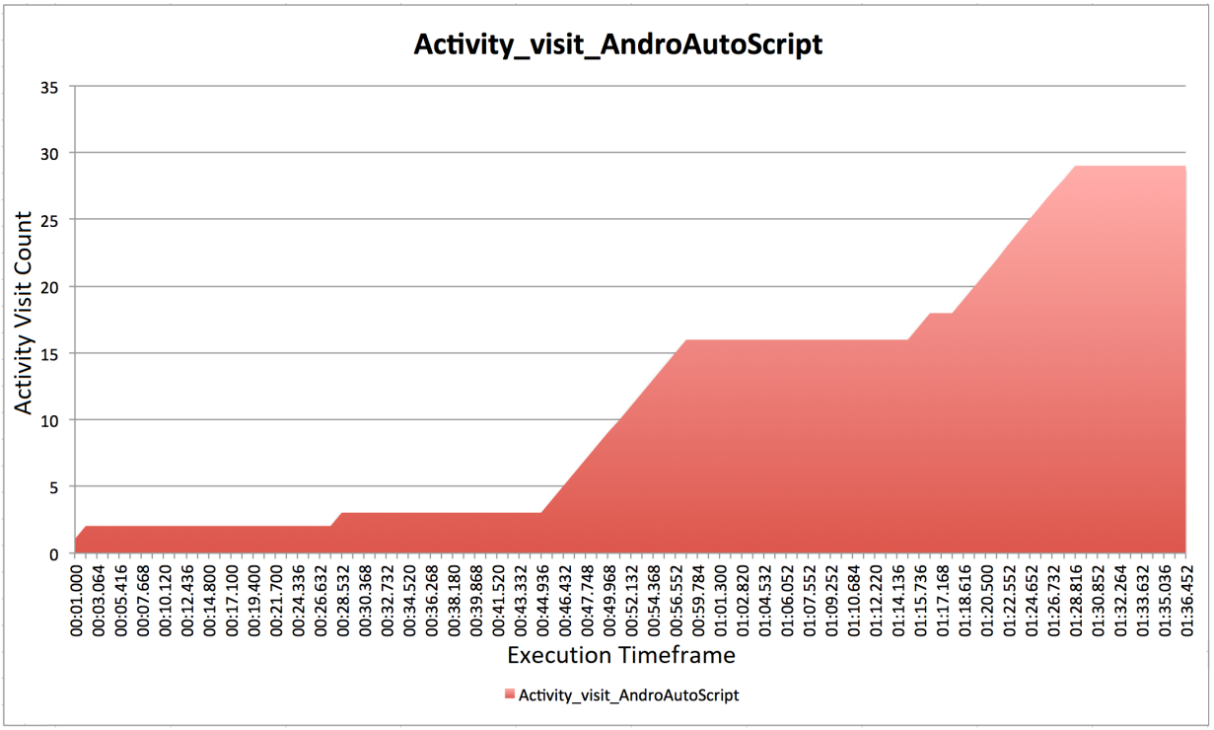

Figure 23. AndroAutoScript performance on visit Activity

Discuss

In our result of the experiment, we can see the Activity visit count on both approaches in a period of ninety seconds. Monkey executes very fast UI triggering that increase the visit count rapidly in the first ten seconds. In this period Monkey visits almost three times Activities as our system. The script our system generated follows the structure of UI and Activity call in the application. It triggers each UI component step by step. That cause our visit

rate seems to stay low at the start. About forty-five second, AndroAutoScript enters another part of the program which concentrated on Activity usage. The visit count, then, increases steadily when triggering nearby new Activities one-by-one.

At the same time, monkey encounter the bottleneck and visit rate stops growing. Monkey generates random stream that may be trapped inside a loop calling structure and hardly trigger each UI in a multiple-component interface. Although we set command to reduce the system-key-event in monkey, it would sometimes distracted by *BACK* key or other system event. In other words, we can say random trigger stream calls Activities by fortune. On the contrary, AndroAutoScript follows the calling structure and it can steadily and sequentially go through every part of a program.

According to our experiment, our approach can efficiently reach most parts of an application. Monkey, however, would take a lot of time trapped in some calling structure that may slow down the automation progression.

CONCLUSION

In this work, we propose an approach to Android application UI automation. Our motivation is to reduce the human resource cost while execution dynamic analysis on Android applications. Our method combines the information of disassembled code and XML resources file in Apk to build up an automation script. And we choose Android SDK tool, monkeyrunner, as our script executor.

There are other excellent approaches to Android UI automation such as DroidTrace (Zhen et al., 2014) deploy a method called "forward execution". They utilize Apk repackaging technique, insert UI event trigger code after every UI listener setting to fulfill automation. SmartDroid (Zheng et al., 2012) take a mixed approach with both static and dynamic method. Unless analyze ACG with static Apk data, they further make modification on Android emulator to catch implicit call inside Android OS.

The advantage of our approach is that it takes low effort in implementation. We do not need to modify Apk and repackage it neither need to modify Android emulator. Also, we include monkeyrunner in our framework, so this approach can easily transmit in between different version. Our approach takes exception in UI automation into consideration. The alter step setting can increase the integrity of script execution.

Author Contributions: Yining Liu and Hung-Min Sun conceived and designed the experiments; Shih-Chi Wang performed the experiments; Yang Yang analyzed the data; Yeh-Cheng Chen contributed reagents/materials/analysis tools; Shih-Chi Wang wrote the paper; Yining Liu revised the paper.

Conflicts of Interest: The authors declare no conflict of interest. The founding sponsors had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, and in the decision to publish the results.

REFERENCES

Androguard. (n.d.). Retrieved from https://code.google.com/p/androguard/

Android – Wikipedia. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Android (operating system)

Android monkey. (n.d.). Retrieved from https://developer.android.com/studio/test/monkey.html

Android monkeyrunner. (n.d.). Retrieved from https://developer.android.com/studio/test/monkeyrunner/index.html

Apktool. (2017). *A tool for reverse engineering Android apk files*. Retrieved from https://ibotpeaches.github.io/Apktool/

Baskaran, B., & Ralescu, A. (2016). *A Study of Android Malware Detection Techniques and Machine Learning*. In *Master of Arts in Intercultural Studies (MAICS)*, pp. 15-23.

Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M. S., Conti, M., & Rajarajan, M. (2014). Android security: A survey of issues, malware penetration and defences. *Communications Surveys & Tutorials, IEEE, 17*(2), 998 – 1022. https://doi.org/10.1109/COMST.2014.2386139

Genymotion emulators. (n.d.). Retrieved from https://www.genymotion.com

Google Play: number of available apps 2009-2017. (2017). Retrieved from https://www.statista.com/statistics/266210/number-of-527-available-applications-in-the-google-play-store/528

Hou, O. (2012). *A look at google bouncer*. Retrieved from http://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/

Hu, W., Tao, J., Ma, X., Zhou, W., Zhao, S., & Han, T. (2014). Migdroid: Detecting app-repackaging android malware via method invocation graph. In *23rd International Conference on Computer Communication and Networks (ICCCN)*, pp. 1–7. https://doi.org/10.1109/ICCCN.2014.6911805

Smartphone OS market share. (2017). *IDC, Q1*. Retrieved from https://www.idc.com/promo/smartphone-market-share/os526

Smieh. (2012). *Anatomy physiology of an android*. Retrieved from https://commons.wikimedia.org/wiki/File:Android-System-Architecture.svg

Zhang, L., Niu, Y., Wu, X., Wang, Z., & Xue, Y. (2013). Automatic analysis of android malware. In *International Workshop on Cloud Computing and Information Security (CCIS)*, pp. 89-93.

Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., & Zou, W. (2012). Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *SPSM'12,* pp. 93-104. https://doi.org/10.1145/2381934.2381950

Zheng, M., Sun, M., & Lui, J. C. (2014). Droidtrace: A ptrace based android dynamic analysis system with forward execution capability. In *International Wireless Communications and Mobile Computing Conference (IWCMC)*, pp. 128–133. https://doi.org/10.1109/IWCMC.2014.6906344