SymDroid: A Symbolic Executor to Identify Activity Permission in Android Application

Qianwen Li
University of Maryland, College Park
qianwenliq@gmail.com

ABSTRACT

Mobile development is expanding widely over the past few years. Amongst the top operating platforms for Smartphones, Google Android platform has been discovered with known privilege escalation attacks. Many of the known privilege escalation attacks are related to the permission system used by the Android platform. This paper will introduce SymDroid, a tool using symbolic execution to explore permissions used in Android applications.

1 INTRODUCTION

The Smartphone market has become one of the fastest growing areas in the telecommunication industry in recent years and will soon dominate the mobile devices market. A report by "prnewswire.com" states that [14]: As of December 2nd, 2011, a total of 989,476 mobile applications have been downloaded, and on average, approximately 2,000 new apps enter the Smartphone applications marketplace each day. Out of all the applications being downloaded, the Google Android[1] application, with 320,315 applications that have been downloaded to date, is the second most downloaded applications of all times. This number is closely behind the Apple iOS application, which has a market share of 32.54 percent of the overall Smartphone application market and represents an increase of more 100 percent over the last 12 months. Smartphone applications range from a basic calculator to powerful analytical tools that even have the capability to remotely control the lighting system in a person's house. In the dynamic environment where better and newer features are constantly being added to the mobile applications, security issues become increasingly critical; especially since most applications are developed by third parties. In recent years, numerous research papers and tools have been developed targeted at discovering security vulnerabilities in mobile applications. Some examples include XmanDroid[3] and SCanDroid[11]. The purpose of this paper is to introduce a way of auditing permissions used and permissions checked during application launching. We describe SymDroid, a symbolic executor for identifying permission checking on Android applications. The underlying concept of SymDroid is to use symbolic execution to branch through all the possible paths of an application and perform permission checking on each path. This is a continuation of a project done by Jinseong[12], who is a current graduate computer science student at the University of Maryland, College Park.

The remainder of this paper is organized as follows: Section 2 introduces the background of Android permission system and symbolic execution; Section 3 explains the architecture of each part of the SymDroid in detail as well as discusses the test results; and Section 5 is the conclusion.

2 BACKGROUND

2.1 ANDROID

Android is a software stack for mobile devices that includes an operating system, middleware and key applications[7]. Android applications are Java-based, but there exists several differences between the Andorid application and the Java program. In contrast to the Java virtual machine, Android uses the Dalvik Virtual Machine, which runs on a slow CPU, with relatively little RAM while powered by a battery and it emphasize on memory efficiency[2]. Additionally, the Android application generates a binary file called the .dex file, or the Dalvik Executable file, in additional to the .class file. In other words, the Android application will first compile to .class files just like Java, and then it will convert all the .class file into one single .dex file[9]. The primary difference between .dex format and .class format is the elimination of redundancy of the five constant pools (string_ids, type_ids, proto_ids, field_ids, method_ids) under the header that are in each file by moving them to the top of the .dex file.

[2]. Figure 1 below demonstrates the difference between the regular Java format architecture and the Android .dex format architecture.

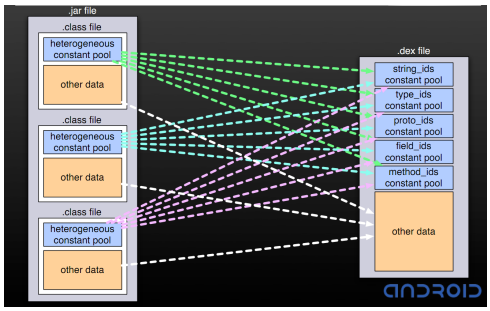

Figure 1: comparsion of .dex file and .class file[2]

Android applications typically start with an activity. The activity provides a user interface for a single screen in your application. [8]. An activity can be in one of four stages: created, stopped, resumed or destroyed. Figure 2 shows the activity's lifecycle. New activities can be invoked from any existing activity, and when a new activity is being created, it will be pushed onto the activity stack with the caller application underneath it. Besides managing the lifecycles of individual components, Android's application framework is also responsible for calling components from other components and for passing messages between components[11]. Interprocess communication (IPC) on Android allows intents to invoke other intents or pass information to other intents. In IPC, intent is passed as parameters for launching activities[11].

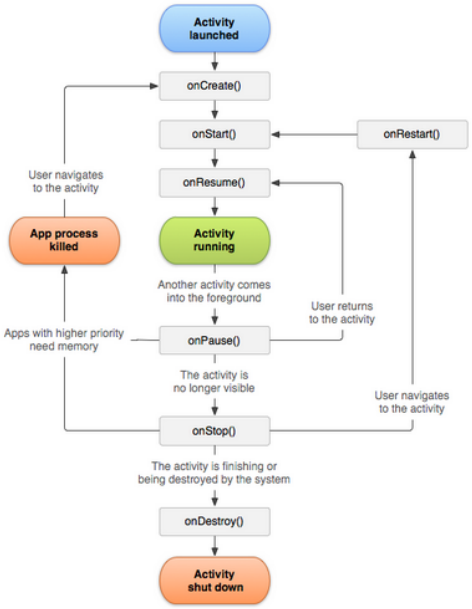

Figure 2: The activity lifecycle[8]

Android deploys application sandboxing and a permission framework is implemented as a reference monitor at the middleware layer to control access to system resources and mediate application communication[4]. More specifically, Android application uses "system-centric" security model, where applications statically identify the permissions that govern the rights to their data and interfaces at installation time. Each permission label is a unique string variable declared in the Android Manifest.xml file by the developer to legally grant access to certain features such as Internet usage and the ability to read / write content. However, such technique has been identified with several security weaknesses because the permissions given are mostly coarse grained on what the application is granted or how the application will be using it[15]. For instance, an information sharing application such as Bump[5], which is an application that allows two devices to share photos, contacts and applications by bumping into each other, could be sending more information than authorized. To prevent such security vulnerability, software testing is commonly used.

A project called the Stowaway project[10] analyzed the Android permission system. Stowaway declares that permission checking could be performed in three ways: interacting with system API calls, database and message-passing system. The API functions may be protected by permissions. Operations on user data, which is stored in Content Provider, may require permissions. Applications may also need permissions to receive Intents (i.e., messages) from the operating system.

2.2 SYMBOLIC EXECUTION

Symbolic execution was first introduced around the 1975s and has been explored for many usages since then. Recently, various papers have been published advocating for the use of symbolic execution for program analysis and bug exploration. The main idea of symbolic execution is to treat values as symbolic expressions that can assume any value. The execution will proceed under normal execution except when computation of expression or conditional branching occurs. In which case, the computation of expression will be "delayed" or generalized by the appropriate algebraic formula manipulations and conditional branching are being forked on all possible paths[6]. The path conditions will accumulate all the constraints that the symbolic value must satisfy in order to proceed to the branch. As a result, the output values computed by a program are expressed as a function of the input symbolic values[13]. Constraint solver is used to test branching condition and the final assertion, if there is any. The main advantage of using symbolic execution for program testing is that it only needs to execute

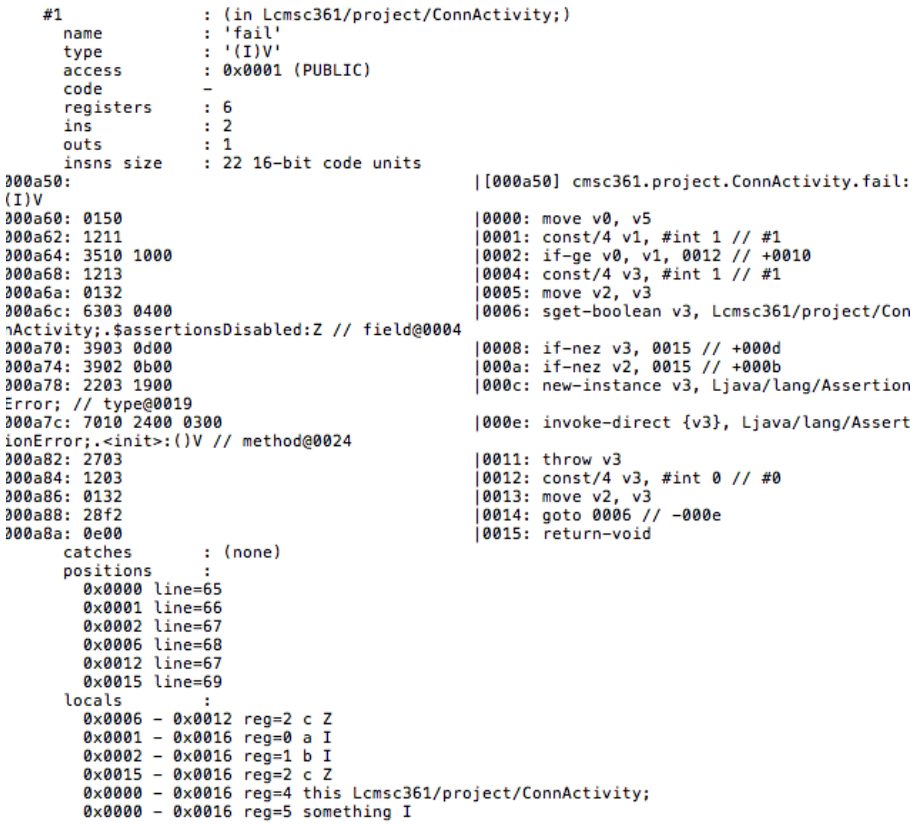

Figure 3: sample .dex instruction

on the program once and the ability to generate counterexamples that can be used to recreate the same behavior.

3 ARCHITECTURE AND IMPLEMENTATION

3.1 OVERVIEW

As introduced, Symdroid is used to detect confused deputy attacks on Android framework. The system will run on application .apk or .dex file, symbolically execute through all possible paths to identify permission checks. The entire system consists of three main parts: parser, executor and permission checker. The parser will read in the target application or the .dex file, parse the instructions and stored it in an inner structure that is used by the executor. The executor includes a virtual machine and a symbolic executor. The virtual machine controls a list of statements and a list of functions to be executed in order. The symbolic executor will execute each statement and branch all possible paths. Permission checking is done on each Android API function call using the Stowaway project, which will be introduced in detail in Section 3.4.

3.2 PARSER

A parser was created for SymDroid so that the .dex file will be directly parsed and stored in the internal structure called dex, similar to the original binary format. The dex structure has fields such as dex_header, d_string_ids and d_data to represent objects in the .dex file. The symbolic executor will then use the dex structure directly for execution. Similarly, all Android opcodes will be mapped to the corresponding opcode structure. The operand structure can be one of four types: const of type int64, register of type int, register of type int, index of type int, and offset of type int32. The instruction is declared to consist of an opcode and a list of corresponded number of operands. Figure 3 illustrates a sample .dex file generated by the dex disassembler provided by Android, called the *dexdump*. The internal structure for SymDroid is much similar to the figure structure. The parsed data, which contains a list of Dalvik instructions, is then transferred to the executor.

3.3 EXECUTOR

As mentioned, the executor consists of a virtual machine and a symbolic executor. Virtual machine

controls the overall flow of the program while symbolic executor will execute each statement and return all possible paths after the current statement is executed.

Algorithm 1 sample execution queue

```
main.onCreate();
main.onStart();
main.onResume();
l = ListOfOtherIntents();
i = 0;
while i < l.length do
  l[i].onCreate();
  l[i].onStart();
  l[i].onResume();
  l[i].onPause();
  l[i].onStop();
  l[i].onDestroy();
end while
main.onPause();
main.onStop();
main.onDestroy();
```

3.3.1 VIRTUAL MACHINE

The virtual machine for SymDroid is simply two while loops. The outer while loop, which is added in order to simulate Android framework, iterates through a queue of functions to be executed and the inner loop will loop through each statement of the current function. The Android framework is simulated by calling activity API functions sequentially according to the order of activity lifecycle. More specifically, the list of intents is filtered to create the queue of functions to be executed. The main activity will first push in its own list of functions (onCreate, onStart, onResume, onPause, onStop, onDestory), then the same list of functions will be pushed in the queue between the onResume and onPause function of the main activity for each activity that is not the main activity. Algorithm 1 shows a sample code that generates the execution queue. When a new activity is being created and started within the app itself, the onCreate function of the new activity is called and the execution queue will be modified to include the list of functions for the new activity. This behavior is the same as calling startActivityForResult from a driver application. Once the virtual machine receives the list of Dalvik instructions from the parser, it will first map the data to a structure called micro instruction, a structure created to simulate Dalvik bytecode instruction. Figure 4 shows the semantics for the micro instruction. The list of micro instructions is then passed to the symbolic executor.

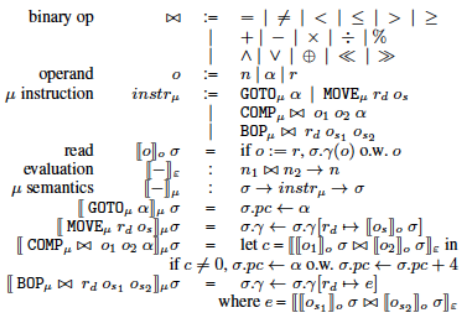

Figure 4: semantics for micro instructions[12]

3.3.2 SYMBOLIC EXECUTOR

The symbolic variables for SymDroid are primarily function parameters. The starting function onCreate takes in a parameter of type Bundle, which is basically a list of hashes of key value pairs. An empty mapping is passed onto each onCreate function and whenever a value is retrieved from the bundle, a symbolic variable will be created. Each statement is processed just like normal machine language instructions using the internal dex structure, where corresponding register and heap memory get updated with each statement. Symbolic variables are also updated by adding constraints. If conditional branching includes a symbolic variable, all possible paths will be explored by solving constraints for the symbolic variable.

In order to solve constraints for symbolic value, SymDroid binds a constraint solve called stpvc. The stpvc is a higher level interface to Libstp, which is a library for Ocaml STP constraints solver. An instant of validity checker is being created in Dalvik.ml to serve as the global validity checker. When branching is needed for computation, simply pass in a query with the global validity checker to the stpvc and it will return a boolean value indicating if the path is reachable.

3.4 PERMISSION CHECKING

One of the key features I added to SymDroid is the permission checking. The Stowaway project is used to handle permission checking. The Stowaway project is a tool that detects over privilege in compiled Android applications[10]. More specifically, it determines the set of API function calls that an application uses and maps those function calls to permissions[10]. For instance, a mapping from the function WebView.loadurl() to android.permission.INTERNET implies that whenever the function WebView.loadurl() is being called, it will eventually call the function checkPermission(android.permission.INTERNET).

Application Name	Declared	Check Hit	Unique Hit	Caller Check
BackupRestore	0	0	0	0
BluetoothChat	2	1	1	0
BluetoothHDP	1	1	1	0
ContactManager	3	1	1	0
Home	6	1	3	0
RandomMusic Player	2	1	1	0
SipDemo	6	1	1	0

Table 1: Result for Sample Applications

SymDroid will load the Stowaway mapping onto a map structure stored in SymDroid prior to execution. When an Android API function is called, the system simply maps the function name and output any permission required if there is any. If the function call is checkCallingPermission or other similar methods, the system will print out the permission granting permission to the caller app.

3.5 EVALUATION

A list of applications has been selected from the sample Android applications came along with the SDK for testing. Tables below show the result of running the applications on SymDroid as well as a short description of intents in each application. For Table 1, the first column has the application; the second column shows the number of permissions declared in the Manifest.xml file; the third column is the number of permission checks being hit by SymDroid; the fourth column indicates the number of unique permission checks being hit; and the last column presents the number of unique permission checks being hit which also have the caller application permission check beforehand. Table 2 shows the list of API calls and the corresponding permissions for each app. Most applications hit at least one permission check but none of them have any permission checks for caller application. Table 3 lists the description of activities for each app. For example, figure 5 shows a piece of code from the Home application. When executes the onCreate() function for the Home activity, SymDroid steps into the function setDefaulWallpaper() and hits the API function call com.example.android.home.Home.clearWallpaper() after a branching condition on the variable "wallpaper". In this case the variable "wallpaper" has the contraint that it does not equal to NULL. Next, the permission checking maps the API call (in this case is the call android.app.Activity.clearWallpaper()) to the Stowaway map and outputs the permission android.permission.SET_WALLPAPER. Note that the number of unique permission check hit is most of the time less than the number of permission declared in the Manifest.xml file. This is because permission may be required in three ways: interacting with system API, database, and the message-passing system[10]. Symdroid only handles permission checking on system API for now.

4 CONCLUSION

Recent discoveries of several security vulnerabilities raise the concern of permission safety policy used by the Android security framework. In this paper we present the concept and architecture of SymDroid, which is a system for examining permission checking against Android framework. SymDroid uses symbolic execution to explore all possible paths and perform permission checks on each Android API function calls. This study tested sample Android applications and results suggest that none of the applications perform caller permission checks.

Going forward, there are many opportunities of improvements for SymDroid. One area of improvement is to include permission checking on content provider and intent. Hence all permission checks will be discovered.

4.1 RELATED WORK

Many Android security enforcement techniques has been discussed and published recently: Saint and Taintdroid monitor inter-application flow by modifying the Dalvik VM and libraries to track the flow of sensitive data between different applications[16][17]. XManDroid is taking the approach of analyzing the communication links among application and apply policy on such communication[3]. SCanDroid, which is declared to be the first program analysis target on the Android platform, is trying to extract security specification from manifests and using control flow to analyze[11].

REFERENCES

[1] android. www.android.com.

[2] D. Bornstein. Dalvik vm internals. In *Google I/O Developer Conference*, volume 23, pages 17–30, 2008.

App Name	API Function Call	Permission
BackupRestore	android/app/backup/BackupManager/dataChanged	android.permission.BACKUP or NONE
BluetoothChat	android/bluetooth/BluetoothAdapter/isEnabled android/bluetooth/BluetoothAdapter/getBondedDevices	android.permission.BLUETOOTH android.permission.BLUETOOTH
BluetoothHDP	android/bluetooth/BluetoothAdapter/isEnabled	android.permission.BLUETOOTH
ContactManager	com/example/android/contactmanager/ContactManager/setContentView	android.permission.INTERNET or NONE
Home	com/example/android/home/Home/clearWallpaper com/example/android/home/Home/setContentView android/app/ActivityManager/getRecentTasks	android.permission.SET_WALLPAPER android.permission.INTERNET or NONE android.permission.GET_TASKS
RandomMusicPlayer	com/example/android/musicplayer/MainActivity/setContentView	android.permission.INTERNET or NONE
SipDemo	com/example/android/sip/WalkieTalkieActivity/setContentView	android.permission.INTERNET or NONE

Table 2: Permission Checking List

App Name	Num of Activity	Activity Description
BackupRestore	1	BackupRestoreActivity: main activty to backup data
BluetoothChat	2	BluetoothChat : main activity that display current chat session DeviceListActivity: a dialog that list all paired devices and detected devices
BluetoothHDP	1	BluetoothHDPActivity: activity passes messages to and from service
ContactManager	2	ContactManger: starting activity, display the list of contact ContactAdder: activity to add new contact
Home	2	Home: simulate the Home that users use to launch applications Wallpaper: Wallpaper picker for the Home application. User can choose from a gallery of stock photos
RandomMusicPlayer	1	MainActivity: shows the media player buttons and lets the user click them
SipDemo	2	SipSettings: Handles SIP authentication settings for the Walkie Talkie app WalkieTalkieActivity: Handles all calling, receiving calls, and UI interaction in the WalkieTalkie app

Table 3: List of Intents

```
@Override
public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setDefaultKeyMode(DEFAULT_KEYS_SEARCH_LOCAL);
    setContentView(R.layout.home);
    registerIntentReceivers();
    setDefaultWallpaper();

    loadApplications(true);

    bindApplications();
    bindFavorites(true);
    bindRecents();
    bindButtons();

    mGridEntry = AnimationUtils.loadAnimation(this, R.anim.grid_entry);
    mGridExit = AnimationUtils.loadAnimation(this, R.anim.grid_exit);
}

/**
 * When no wallpaper was manually set, a default wallpaper is used instead.
 */
private void setDefaultWallpaper() {
    if (!mWallpaperChecked) {
        Drawable wallpaper = peekWallpaper();
        if (wallpaper == null) {
            try {
                clearWallpaper();
            } catch (IOException e) {
                Log.e(LOG_TAG, "Failed to clear wallpaper " + e);
            }
        } else {
            getWindow().setBackgroundDrawable(new ClippedDrawable(wallpaper));
        }
        mWallpaperChecked = true;
    }
```

Figure 5: Sample code for the Home application

[3] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Security*, 2011.

[4] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *Proc. of the 19th Network and Distributed System Security Symposium (NDSS 2012), San Diego, CA*, 2012.

[5] Inc Bump Technologies. Bump.

[6] J.A. Darringer and J.C. King. Applications of symbolic execution to program testing. *Computer*, 11(4):51–60, 1978.

[7] A. Developers. What is android? *Internet: http://developer. android. com/guide/basics/what-is-android. html [Sept, 5 2011]*, 2010.

[8] Android Developers. http://developer.android.com/.

[9] D. Ehringer. The dalvik virtual machine architecture, 2010.

[10] A.P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[11] A.P. Fuchs, A. Chaudhuri, and J.S. Foster. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa*, 2009.

[12] Youngil Kim Jinseong Jeon. Symdroid: Symbolic execution for dalvik virtual machine.

[13] S. Khurshid, C. PĂsĂreanu, and W. Visser. Generalized symbolic execution for model checking and testing. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, 2003.

[14] Mobilewalla. Mobile apps approaching major milestone of 1 million apps in marketplace: Mobilewalla.

[15] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 340–349. IEEE, 2009.

[16] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 340–349. Ieee, 2009.

[17] E.J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 317–331. IEEE, 2010.