Forced-Path Execution for Android Applications on x86 Platforms

Ryan Johnson, and Angelos Stavrou
Department of Computer Science
George Mason University
Fairfax, Virginia 22030
Email: {rjohnso8, astavrou}@gmu.edu

Abstract—We present a code analysis framework that performs scalable forced-path execution of Android applications in commodity hardware. Our goal is to reveal the full application functional behavior for large commercial applications without access to source code. We do so by identifying code blocks and API calls that are deemed sensitive and provide a security report to an analyst regarding the functionality of the Android application that is under inspection.

We show that our approach is scalable by allowing for the execution of each software component by numerous instances of execution modules. Each execution instance exercises a different code path through the application call-graph leading to full code and state space coverage and exposing any hidden or unwanted functionality. The output is a list of API calls, parameter values, component call graphs, and control flow graphs. We show how this can be leveraged for automated policy enforcement of run-time functionality.

Index Terms—Android OS; Application Analysis; Emulation;

I. Introduction

The corresponding capabilities resulting from the requested permissions of an Android application can be difficult to comprehend for general users [1], [2]. In addition, there is generally little transparency into how the permissions will be used. Numerous Android applications request the android.permission.INTERNET permission. An installed application with this permission will be able to open network sockets and communicate with remote hosts. The intent of the network access can range from pulling down benign ads via a third-party advertising kit to exfiltration of personal data. The exfiltration of personal data can be unknowingly sanctioned by the user due to their decision to install an application with permissions that allows the application to perform the exfiltration. The pertinent information for network communication is the endpoint of the communication and the data that is transferred. In addition, some applications that have the android.permission.SEND_SMS will simply send a text message to the user's phone thanking them for downloading the application, whereas certain trojanized applications will send text messages to premium numbers [30]. Obtaining granular information for an Android Application Programming Interface (API) call can supply context and reveal intent. Android permissions declared in an application's AndroidManifest.xml file allows an application to utilize the functionality that the permissions yield. Application developers tend to assign unnecessary permissions to Android applications [18], [19]. Therefore, some of an application's permissions and their corresponding functionality may not be used which can mislead the user.

Our framework can extract low-level information about the behavior of an application. This can reveal the true intent of why an application developer requests powerful permissions which can affect the user monetarily or result in a compromise of personal information. The framework provides an automated method for obtaining the behavior of an application while maximizing the number of execution paths taken through the application in order to log the values of parameters to sensitive Android API calls. Some malware will immediately execute malicious actions while other malware may wait for certain conditions to occur [5]. Our framework is applicable to both approaches since the framework utilizes forced-path execution to enter branches that may be difficult to reach under normal circumstances. Forced path execution has been used in other frameworks to increase execution coverage of a program [6], [7], [10]. Stressing the application code can reveal hidden functionality which can contain specific conditions that need to be met for malicious code to be executed. In addition, an application may have defenses [22] in place by making an attempt to detect its execution environment and consequently altering its behavior [14]. Countermeasures [15] also exist for when a program attempts to detect its execution environment.

II. Analysis Framework Implementation

The structure of the framework was previously described in [9], [8]. The analysis framework only requires the Android Package (APK) file of an Android application. We use open-source software called apktool [3] to generate a human-readable representation of the Dalvik bytecode for an application. This format is called smali [4] and each smali file generally corresponds to a Java class. Nested Java classes will have their own smali file. The analysis framework reads, parses, and executes the instructions from the smali files. The framework contains a Java implementation for executing each of the 226 Dalvik instructions as they are represented in the smali files [32]. The Dalvik Virtual Machine (VM) uses registers to represent primitive data types and refer to objects. We created a custom Java data type to represent the registers in our framework, so they can be used in

```
Process p = Runtime.getRuntime().exec("rm -r /mnt/sdcard/DCIM");


invoke-static , Ljava/lang/Runtime;->getRuntime()Ljava/lang/Runtime;
move-result-object v9
const-string v10, "rm -r /mnt/sdcard/DCIM"
invoke-virtual          v9,          v10,          Ljava/lang/Runtime;-
>exec(Ljava/lang/String;)Ljava/lang/Process;
move-result-object v6
```

Fig. 1. *Equivalent code for Java and Smali.*

the Dalvik instructions. Each Dalvik instruction has its own implementation that utilizes our custom class implementing registers and the framework's internal data structures. A single line of Java code can translate into multiple lines of smali code since it is at the bytecode level. The registers in Dalvik bytecode are designated by a lowercase letter v or p followed by a decimal number. The translation of a Java statement for deleting all pictures from a rooted Android device into equivalent lines of code in the smali format is shown in Figure 1. An Android application would only need the android.permission.WRITE_EXTERNAL_STORAGE permission to perform such an action.

The AndroidManifest.xml is examined to extract the starting class associated with each application component in an application. In addition, the content within certain tags (e.g., IntentFilter, data, meta-data, etc.) are also logged. Each application component, except for Content Resolvers, declared in the application's AndroidManifest.xml file is executed as well as any dynamically registered BroadcastReceivers. Initially, the application component(s) with the category of android.intent.category.LAUNCHER are executed first. Any Intent objects that are sent to another application component within the application will be stored in a queue for the application component to access when it is executed. There are certain instances where application components will be created instantaneously to interact with the currently running application component. If the application component calls a Service application component, sends a non-system broadcast to a BroadcastReceiver application component, or executes API calls like android.app.Activity.startActivityForResult(Intent, int), etc.

Coverage of each application component is modeled using a binary tree where each node represents a conditional statement, a switch case, or a helper node that indicates that a certain branch from it's parent node should not be taken. This can be the result of the execution path completing when a branch from the previous conditional statement or switch case is taken, a hard-coded exception being thrown within the application, an infinite loop being detected, encountering an API call which will terminate the VM, or a loop hitting the maximum allowable number of loop iterations. Each leaf node in the binary tree will be a node that indicates that one of the aforementioned cases arose, assuming the framework has been given ample time to complete the execution of the application component. The framework contains mechanisms to limit the execution time by allowing for an upper limit on the allowable iterations through a loop, limiting recursion to a

certain number of recursive calls, and allowing a time limit to be set so that the execution of each application component will be bounded by a maximum execution time. These approaches also aid in avoiding infinite loops.

The main components of the framework are a single controller module and a plurality of execution modules. The controller module coordinates the operation of the execution modules and maintains shared data-structures (e.g., binary tree, results, etc.). The execution modules operate independently and concurrently take different execution paths through an application component. An execution module exists for a single unique execution path through an application component. Upon completion of a path, the controller module reinitializes the execution module so it can take an new path through the application component, assuming all paths have not been taken and that the time limit, if one exists, has not elapsed. The output from processing each application component is: (1) a method call graph, (2) control flow graph, (3) the output of an in-order traversal of the binary tree, (4) a list of the jump values taken for each execution through the application component, and (5) a list of relevant behaviors of the application component.

Some recent additions to the framework have been made to increase efficiency. We have introduced multi-threading into the program to increase the coverage of an application. A user-selected number of execution modules can concurrently execute through an application component. Each execution module contains specific data (e.g., call stack, state of variables, network connections, files, the state of static variables, etc.) and operates independently of the other execution modules except for sharing the binary tree that is modeling taken and untaken execution paths and a few other data structures. Each execution module uses a pseudo-random number generator to determine which path from a conditional statement should be taken, assuming both branches are open and have not been completed. In certain instances, only a single branch from the conditional statement will be open, so that branch will be taken. The same is also true for determining which switch case to take when a choice is available. The pseudo-random number generator introduces entropy in the execution so that coverage will likely differ during subsequent executions of the analysis framework, assuming the analysis does not complete within the user-set time limit.

The framework has also allowed some flexibility in allowing a loop to iterate a greater number of times than the user set upper bound. This generally occurs when the application is performing file or network I/O. In these instances, the program will allow the a loop to execute as the actual conditions dictate. This occurs by examining the data source attribute of a RegisterEntry object, our custom data type for registers, which is being used as an operand to an if conditional statement. The data source is first examined to see if the RegisterEntry is a product of an API call that is allowed to bypass forced execution. In the case of performing network I/O, this is generally done by java.io.InputStream or one of its subclasses. This class, and its subclasses, contain methods to read from a
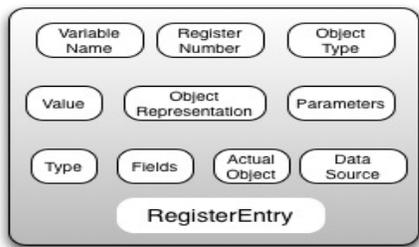
Fig. 2. *Graphical representation of a RegisterEntry object.*

stream, write to a buffer, and return the number of bytes read. Programmed within the handling of these calls is to tag the integer that will be returned with the number of bytes read from the stream. A check for this and other tags is coded into the handling of if conditional statements. When the source of one of these calls is determined to be file or network I/O, then the conditional statement will be evaluated based on the actual values and not rely on forced path execution.

A. Representing objects and primitive data types

The Dalvik VM uses registers to reference objects and primitive data types. The register simply contains a reference to an object or the bit value of a primitive data type to be used in an instruction. The register-based approach differs from the stack-based approach commonly used in non-mobile platforms. The framework uses a a custom Java data type called a RegisterEntry which represents a register and the underlying object or primitive data type to which it refers. The data type also contains ancillary data used internally by the framework. Figure 2 contains a graphical representation of the RegisterEntry data type and its fields.

The Type field indicates whether the RegisterEntry object represents a primitive data type or an object. The Register Number field denotes the register number which is a unique identifier for each register within a method. The Value field is a String object which is used to represent the value of primitive data types. The String will be converted to the underlying primitive data type and back when needed. This field may also be used to contain a String value for simple objects. The Object type field is used to denote the actual object type. This is necessary for finding the correct method to call when the object is cast as a supertype or as an interface. The Fields field is a data structure to contain the fields, represented as RegisterEntry objects, of an object. The Variable name field can contain a variable name which may or may not be present depending on how the application was compiled. The presence of a variable name is useful for correlating the variable in the bytecode with the variable in the decompiled source code. To get an approximation of the source code for an APK, a commonly used technique is to use dex2jar [23] to obtain a jar file and then examine it using a Java decompiler. The Parameters field is a data structure that contains all RegisterEntry objects that were used as parameters to the constructor call for an object. The Actual object field is an Object reference that can contain the actual object that the RegisterEntry object represents. The Object representation field is an Object reference that contains a custom data type that represents the object. The Data source field indicates the method call or Android API call for which the RegisterEntry is a product. This attribution of source helps for various functions such as determining how a key is generated, tagging data for generalization purposes, and helping to determine when the strict loop execution limit can be bypassed (e.g., reading from a file).

B. Handling Non-Resident Code

Our framework emulates the Android Operating System (OS) which normally runs on a Android-enabled device or an emulator provided in the Android Software Development Kit (SDK). We developed an execution framework that executes within a Java VM and contains a Java implementation to execute the Dalvik opcodes as they appear in smali files. Groupings of similar Android API calls have their parameter values logged as they occur during execution. The high-level categories, with each containing related API calls, are commands executed, Java reflection, libraries loaded, network I/O, file I/O, native method calls, Intents sent, ContentResolver events, Bluetooth events, NFC events, location events, BroadcastReceiver events, telephony events, ClassLoader events, Cryptography events, and Media events. Some of the API calls in these groups will require a permission, whereas others will not. Due to the extensive data that can occur with file and network I/O, the actual network I/O are written to files and the file I/O are actually implemented by mirroring the directory structure of the phone within a special directory.

The code for the Android API is absent from the output of apktool since this code runs on the device according to API version of the device. This absence presents challenges, so we take a few different approaches in handling the Android API calls contained within an application. Each API call must specifically be handled by our program, or a stub object with no state will be created and returned. The abstraction of returning a hollow stub object with no state creates some efficiency by not executing an implementation for the API call, although it reduces precision of the analysis since the object is not actually present to work with. We focus on classes that have sensitive functionality, implement data structures, correspond to the boxed primitive data types, and have functionality that is integral to the operation of Android applications and the OS.

Generally, we use the actual object for classes residing in the Java API and use an alternate representation of an object for the classes in the Android API. The Android API is extensive and contains a subset of the Java API. Since our framework was developed in Java, we can directly execute any Java API call by creating the object with its initial parameters when a constructor is encountered and calling the object's methods directly as they are encountered in the smali files. Figure 3 shows the implementation for the java.lang.String.toCharArray() method call. This illustrates a rather terse case for handling a call from the Java API. We also take advantage of inheritance so that a method call can

```
else if (methodCall.equals("java.lang.String.toCharArray()"))
{
    String stringObj = regNumbers.get(0);
    RegisterEntry stringObj = this.getRegEntry(stringObj);
    String strValue = (String) stringObj.actualObject;
    String reg = this.getMoveResultRegister(methodLinesPart);
    if (!reg.equals("")) {
      RegisterEntry charArray = new RegisterEntry("object",
reg, strValue, "char[]", "");          charArray.actualObject =
strValue.toCharArray();
        this.updateMapping(charArray);
    }
    return "true";
}
```

Fig. 3. *Implementation for the java.lang.String.toCharArray() method call.*

be coded for the highest class in the hierarchy and be valid for all the subtypes for that method. This is possible since the actual object can be cast as a supertype for the purpose of executing the method call and be valid for the subtypes.

Our alternate representation is a custom Java data type that contains fields corresponding to the fields that the modeled class from the Android API contains. We have also coded methods that operate on these fields according to the behavior of particular API calls. For example, the android.os.Bundle class is commonly used as a data structure to encapsulate other objects and primitive data types. We examinied the source code for the Bundle class and discovered the internal representation of the Bundle class is a java.util.HashMap object. The Bundle class reveals that the Bundle class has various methods which are simply wrapped calls to the equivalent method of the HashMap object. Our framework operates in a similar way for certain classes by handling calls to a Bundle object and having them act as a wrapper to the calls to the underlying HashMap object which our program uses as an alternate representation for Bundle objects. The Intent class is important for intra- and inter-process communication, so we have created an custom data type to encapsulate the relevant fields for Intent objects. The methods for getting and setting these fields have been manually coded to get and set the fields within the custom data type that comprises the Intent object's alternate representation. We have also hard-coded the values of certain static variables from classes in the Android API. Whenever the framework encounters an sget instruction, the value is checked to see it is an Android API static variable that has been hard-coded in the framework.

Some API calls will not require the use of an actual object or an alternate representation. The android.app.Activity.getText(int) API call does not require any specialized representation since smali file(s) for the class extending android.app.Activity exists. The inherited API calls from the android.app.Activity class need an implementation so they can be handled. The android.app.Activity.getString(int) API call is handled in the framework by internally accessing the public.xml and strings.xml files to obtain a

String value. For certain types of API calls, a hard-coded value will be returned. For example, for the android.location.Address.getLatitude() API call, the framework will always return a double value of 892754.563920. The framework does not have GPS capability, so returning a hard-code value can help to track this value by utilizing the dataSource field of the RegisterEntry object that contains it. Certain API calls such as those to send text messages from the android.telephony.SmsManager and android.telephony.gsm.SmsManager classes will just log the parameter values to the calls and not actually exercise the functionality of sending the text message.

We are considering using a rooted phone to obtain certain class files form an Android phone, creating a jar, and adding them to the build path to access these classes the same way the Java API is being accessed. This will only be possible for certain classes depending on the actual dependencies of an individual class. We are still investigating which classes would be applicable for this approach. The dependency chain for a single class could become untenable. Classes that use native code to access phone hardware pose additional challenges. Certain classes could be stripped of native code in the source code files and possibly references to other Android classes, then compiled and added to the build path of the framework to give access to the actual object for these calls. This approach requires additional consideration to determine its feasibility. We are also planning to determine the viability of hooking up a phone to the server running the analysis program to offload Android API calls to be handled on the phone by using Java Reflection to create the objects on the device and call their methods, so that the object and its results could be used as the actual object field in a RegisterEntry. This approach has the ability to expand coverage and precision, but should also add significant performance overhead.

C. Forced-path Execution

Since our framework forces through conditional statements independent of the actual values being evaluated, it will be able to obviate run time checks to determine if the application is running inside an emulator. A malware author may include code to determine if the application is being run with a debugger, virtual machine, or emulator. If any of these undesired run time environments are detected, the application may not exhibit its malicious activity in an attempt to avoid analysis of its true behavior [14]. Since execution is forced we do not have to rely on other methods such as fuzz testing [12], symbolic execution [13], or automatic test generation [20]. Some of these approaches exhibit great difficulty in trying to exercise all branches of an application [21].

Monkey [16] is an application, included in the Android SDK, to stress an Android application by introducing certain user events and system events into an Android application. The Monkey program has been used by other analysis and testing approaches for Android [17], [26]. The Monkey program exercises an Android application running on an Android-enabled device or an emulator. The Mon-

```
boolean isMonkey = ActivityManager.isUserAMonkey();
boolean harness = ActivityManager.isRunningInTestHarness();
if (!isMonkey && !harness)
      this.showtime(); // malicious
else
      this.restrainFunctionality(); // benign
```

Fig. 4.   *Monkey Program And Automated Testing Detection.*

key program logs uncaught exceptions and instances of an application becoming unresponsive. If a malicious application developer wants to evade the automatic testing by the Monkey program, they can leverage the Android API by calling the android.app.ActivityManager.isUserAMonkey() method. The documentation for the method states "Returns "true" if the user interface is currently being messed with by a monkey" [24]. There is also a method called android.app.ActivityManager.isRunningInTestHarness() which will determine if the application is running in a test harness which may be performing some fuzzing of the application. Figure 4 shows how the two aforementioned method calls of the ActivityManager class can be used to limit the behavior of the application when conditions unfavorable to the application are detected.

An application may also call different Android API calls and look for certain hard-coded constants that may be returned when the application is running inside the emulator. For example, when a simple program to obtain the phone number of the device is run inside the emulator, an 11 character String starting with 1555 was returned as per our testing. Introducing some entropy into certain outputs or simply returning null from the emulator could aid in the analysis of applications that try to detect the presence of an emulator. These results could be used in the same way as they are in Figure 4 to restrain malicious behavior when the application is being analyzed.

The program uses concrete values for the primitive data types and Strings for execution. If the values for the primitive data types and strings are initialized in the program, then this value is used. Otherwise, a default value is assigned to the primitive data type variables and Strings which are used for execution. This occurs when a variable is not initialized in the resident code such as when an integer is returned from an Android API call for which the code is unavailable and unhandled by our program. When an object is returned from an API call, a stub object is created with no state. As instance variables of the object are set by executing the code, the state of the object will be modified and become more complete. In certain instances, all of the variables passed as parameters to a particular API call may be declared, initialized, and left unmodified before they are used as parameters. In this case, determining the actual values of the variables can be trivial. If one or more of the parameters to the API call are also parameters to the method call which contains the API call, then this greatly complicates determining their values since any number of paths through the application could have led up to the method call that contains the API call. Starting execution from the beginning of the application enables our program to keep track of the values of the variables as execution proceeds.

Adding context by obtaining the values of parameters to Android API calls can reveal important functionality by allowing a more comprehensive perspective of the application's functionality. Static analysis allows one to see the API calls, but keeping track of the parameters values and performing operations on them requires a higher level of sophistication. Examining the values of parameters used to create network connections to remote servers would enable visibility into the final or intermediate node in the communication. This would also allow the domain or IP address to be vetted with DShield [31] or something similar. In certain instances, static analysis of an Android application is not sufficient to be able to be able to determine the actual destination of a communication by itself.

D. Abstracting User Input and Interaction

The program cannot generate context-sensitive user input due to the automation of the process. We handle the Android API calls used to obtain input from the user by returning a String (or other object) that contains data which enables easy identification of how the user input is used subsequently. Notably, the android.widget.TextView class and its direct and indirect subclasses are commonly used to obtain input from the user for purposes other than simple navigation through the application. The framework generally returns a value of [USER_INPUT] for the aforementioned API calls which can be easily identified in files, network stream traces, and other data structures. Other Android API calls are also programmed to return certain hard-coded values for easy recognition. For example, a particularly common and sensitive call is android.telephony.TelephonyManager.getDeviceId() which returns the device's International Mobile Equipment Identity (IMEI). The framework returns a String object with the value of [DEVICE_ID] when the method call is encountered. The analysis framework logs the parameters to API calls that will open and send data over network sockets. We noticed that the application with a package name of org.microemu.android.avg.Main20100909131922453 will send out the IMEI of the phone via a network connection. On line 97 of the file org.microemu.android.GameMIDlet201009091309333671 _1_1.0_bc49deca52db130d6c735a91c901f12b/smali/org/ microemu/android/Update.smali, the API call java.net.URL.openConnection() is made with a URL value of URL:http://rakoo.cn/AndroidStore/getAd.do?goto=chan &chan=J08R1002&imei=[DEVICE_ID].

The user interaction component is also abstracted. Android applications can be heavily event-driven by relying on the user to navigate through the application using buttons, the keyboard, and the screen. We perform these events by automatically forcing execution into these events. To register an action to be performed when a button is clicked, the application will contain a class that will have code executed when the button is pressed. The application will register event listeners that will wait for

and handle user input events. Registering an event handler will make it so that a callback is performed with the event listener detects the event it is listening for. Calling the android.view.View.setOnClickListener(View.OnClickListener) method call will register an object that fulfills the View.OnClickListener interface, so that its onClick() method will be called when the click event occurs. The framework will always immediately execute the methods associated with the callbacks when it encounters them.

Once execution starting from the onCreate, onStart, or on-Receive method completes, the framework will start checking for methods to force execution into. We mimic the lifecycle for each application component by forcing it through its natural progression of states. The Android Developers website shows the state changes for the Activity lifecycle [25]. Each type of application component has its own lifecycle and callback methods which can be executed. In addition, there are certain callback methods according to the application component type that require certain conditions to be present prior to entering the method, so we force into these methods and begin execution. For example, the android.app.Activity.onLowMemory() method will only execute as a callback when the OS is running low on memory and the application wishes to perform some operations before it is terminated. If the application developer specifically wants to take some action, they will override this method to perform some operations. When this occurs, the code for this method is present in the smali file which represents the class that is a subtype of android.app.Activity. The framework sequentially, according to the lifecycle, looks for these callback methods to see if they have been implemented. The user may prefer to override and implement some of these methods as opposed to relying on the default implementation of the method from the superclass (e.g., android.app.Activity for Activities). To perform this forced execution into methods, a reference to the object that is associated with the application component as detailed in the AndroidManifest.xml file is maintained, as well as the state of the static variables. If one of these callback methods has been implemented, it will force the execution of this method to obtain coverage and record any sensitive API calls and the parameters, if any, to the call.

III. NETWORK AND FILE I/O

The program can open and communicate via network sockets and read and write to files located on the file system. Much of the file access and network access within an Android applications is achieved via classes residing in the Java API. Due to this fact, the framework uses the actual objects for these API calls which enable true execution to be achieved.

A. File I/O

Within the directory containing the analysis output for an application, a special directory is created to serve as a basis for mirroring the entire directory structure of and Android-enabled device by acting as the root directory. As files are created and referenced, the directories in the absolute path will be created, if necessary, to mimic the exact file structure. We also sanitize the paths encountered in the application by appending the relative path to the special directory representing the root directory of the device and removing any references to the parent directory that would allow an application to access files that are higher in the file hierarchy than the directory representing the root directory, although this is unlikely to occur. Therefore, reading and writing to files as contained in the application's code can be viewed for analysis. We do not have an automated tool for examining the contents of files to determine if any sensitive data has been written to file. The program does have path information, so that the location of the file can also determine if information is being written to the publicly accessible sdcard (i.e., /mnt/sdcard) or an application's private storage. Due to the repeated nature of the framework which executes the application component once for each unique path through the component, the names of files have a unique integer appended to the end of file name (or right before the file extension if it exists). This is necessary so that different executions of the tool do not interfere with files from previous executions and since different execution paths may lead to different data being read or written. Each instance of the execution module contains its own lookup table to translate from the file name in the program to the name of the actual file on the file system that is being used by the execution module.

B. Network I/O

The program will open network connections to network end-points and communicate via a hardened proxy. This technique is in place to reduce the possibility of external attacks on the host running the framework. Any data written to or read from a network socket will also be written into a file. The framework also uses a lookup table for network connections and appends a unique integer to the front of each domain name/socket, so that the current executions through an application component do not overwrite or modify network traces from previous executions. These files contain the actual data that is being transferred in between the framework and the remote host. The files can be examined for data exfiltration and detection of malicious payloads using a supplementary technique not currently provided by the framework. This approach allows visibility into the data even when encryption is used. Any code for a webpage is not actually loaded into a web browser, so no scripts are executed. In addition, commands within the program will not be executed so as to not compromise the host that is executing the analysis program. This will limit the precision of the analysis, but also prevents the running of binaries contained in the application or downloaded from the internet. We are looking into running the analysis program within a VM that can be easily reloaded and wiped so that more behavior of the application can be examined. Currently, any command issued by the program will have the parameter(s) to the call logged, so that the binary or command can be examined (potentially via another automated method).

File name: wrehifsdkjs.apk_9701d4cfb7d61e43028c10adf70d4efd.out/smali/frhfsd/siksdk/ujdsfjkfsd/WrehifsdkjsActivity$Progress.smali

frhfsd.siksdk.ujdsfjkfsd.WrehifsdkjsActivity.managedQuery(android.net.Uri, java.lang.String[], java.lang.String, java.lang.String[], java.lang.String), The Uri being accessed:content://com.android.contacts/contact, Columns to return :all columns, SQL where clause:all rows, SQL selection arguments:null, SQL order-by clause:default order, Line#:219

frhfsd.siksdk.ujdsfjkfsd.WrehifsdkjsActivity.managedQuery(android.net.Uri, java.lang.String[], java.lang.String, java.lang.String[], java.lang.String), The Uri being accessed:content://com.android.contacts/data/phones, Columns to return :all columns, SQL where clause:contact_id =? , SQL selection arguments:, SQL order-by clause:default order, Line#:322

frhfsd.siksdk.ujdsfjkfsd.WrehifsdkjsActivity.managedQuery(android.net.Uri, java.lang.String[], java.lang.String, java.lang.String[], java.lang.String), The Uri being accessed:content://com.android.contacts/data/emails, Columns to return :all columns, SQL where clause:contact_id = , SQL selection arguments:null, SQL order-by clause:default order, Line#:384

org.apache.http.client.methods.HttpPost.HttpPost(java.lang.String), URI:https://ftukguhilcom.globat.com/cgi-bin/registerAddressData.php, Line#:82

org.apache.http.client.methods.HttpPost.HttpPost(java.lang.String), URI:https://ftukguhilcom.globat.com/cgi-bin/confirmUserData.php, Line#:82

org.apache.http.client.methods.HttpPost.setEntity(org.apache.http.HttpEntity), HttpEntity String:data=\n&t=[PHONE_NUMBER]&app=Wrehifsdkjs, Charset: UTF-8, Line#:110

org.apache.http.client.methods.HttpPost.setEntity(org.apache.http.HttpEntity), HttpEntity String:t=[PHONE_NUMBER]&app=Wrehifsdkjs, Charset: UTF-8, Line#:110

org.apache.http.entity.StringEntity.StringEntity(java.lang.String, java.lang.String), String:data=\n&t=[PHONE_NUMBER]&app=Wrehifsdkjs, Charset:UTF-8, Line#:96

org.apache.http.entity.StringEntity.StringEntity(java.lang.String, java.lang.String), String:t=[PHONE_NUMBER]&app=Wrehifsdkjs, Charset:UTF-8, Line#:96

org.apache.http.impl.client.DefaultHttpClient.execute(org.apache.http.client.methods.Http UriRequest), Uri:https://ftukguhilcom.globat.com/cgi-bin/confirmUserData.php, Line#:113

org.apache.http.impl.client.DefaultHttpClient.execute(org.apache.http.client.methods.Http UriRequest), Uri:https://ftukguhilcom.globat.com/cgi-bin/registerAddressData.php, Line#:113

android.telephony.TelephonyManager.getLine1Number(), Line#:550

Fig. 5. *Framework output for Android.Exprespam malware.*

IV. RESULTS AND FINDINGS

We will discuss some results of analyzing actual Android malware. First, we will discuss an instance of malware that has been called Android.Exprespam which the mobile security community seemingly became aware of in January 2013 [33]. The APK is about 41 KB and the package name is frhfsd.siksdk.ujdsfjkfsd for this specific sample. The application will query the device's content providers to obtain personal data and exfiltrate it using Trasnport Layer Security (TLS). We ran the application through our framework to obtain the possible behavior of the application which took about 6.5 seconds to execute all paths through the application. Figure 5 shows some of the Android API calls that were logged. As the figure shows, the application uses the content providers to obtain contacts' names, phone numbers, and emails from the device. When logging the query to a content provider, we have also provided interpretation logic to promote understanding. For example, a null value for the projection parameter indicates that all columns should be returned for the query. The top line of the figure shows the name and relative path of the smali file that contains the following Android

API calls. The API calls in figure 5 have format of fully qualified method name, parameters and corresponding values, and line number. We have not implemented an internal data structure representing the actual contact data to be queried, so the content values do not show up after data= in the API calls. We plan to implement contrived values and store them internally in the framework so that they may be operated on and tracked.

We also examined another application with the package name of com.example.smsmessaging which is an application that receives phone numbers to spam via Short Message Service (SMS). In addition to the SMS spam capability, the application contains an application with a package name of com.example.adultflicks in its assets folder. The functionality of the com.example.smsmessaging application has been detailed on the internet, but we have not seen any analysis of the com.example.adultflicks applicaiton. Therefore, we will provide some analysis informed by the framework. The Activity named com.example.smsmessaging.Main will copy the adultflicks.apk to the external SD card and name the resulting file duplicate.apk. The Activity then creates an Intent object with an action of android.intent.action.VIEW and a Uri referring to the duplicate.apk file. This will prompt the user to install the application when the Intent object is sent. If the user installs the application, they will be asked to run the application. The application is very small at 176 KB and only contains a single Activity as its only application component. The framework analyzes the APK in about 1.2 seconds. On line 140 of the file adultflicks/smali/com/example/adultflicks/MainActivity.smali, a call to open a webpage with the android.webkit.WebView.loadUrl(java.lang.String) call is made with a String parameter of http://motherless.com/search?q=kim+kardashian. The application's only functionality is to open a WebView within the application to a pornographic website with the search query of "kim kardashian." The application also sets its own WebView client, so it can override the shouldOverrideKeyEvent(android.webkit.WebView, android.view.KeyEvent) to intercept and prevent the normal functionality of the back button.

V. RELATED WORK

To our knowledge, we have not seen another framework that attempts to execute the Dalvik bytecode and instrument its execution. There are a few approaches that utilize the Android emulator to perform dynamic analysis. Blasing et al. [26] use the Android emulator that comes with the Android SDK to perform dynamic analysis on Android application and use a tool to simulate user interaction. The tool also performs some static analysis by disassembling the Android application and identifying certain functionality. Burguera et al. [29] use a sandbox to perform dynamic analysis of Android applications and use a behavior-based approach to classify malware by examining the system calls of that the application makes. Zhou et al. [28] created a program to analyze the bytecode

of an Android application to create behavioral footprints on Android application and then use heuristics to detect classes of malware.

VI. Limitations

Our approach depends on the correctness of apktool's translation of the classes.dex file of an application into a set of smali files. We have not examined the source code of apktool to access the mechanics of the translation process. We do not have a complete implementation for the Android API which limits the precision of the analysis. We are testing the tenability of approaches to increase coverage for the handling of classes in the Android API.Depending on the complexity of the Android application, especially in regard to the number and nature of conditional statements, the analysis of the application can be computationally expensive. The framework allows for the number of loop iterations to be set as a maximum for each loop. This upper bound on loop iterations creates a trade off between precision and time. Complex applications can require a large amount of time to analyze, especially if they contain numerous nested loops. Recursion is limited to a certain number of recursive calls in an effort ensure that a recursive call eventually returns.The user input is abstracted by always providing the same input, although we force the execution into performing user events. The abstraction of user input aids in tagging the data so that it can be identified in files, network I/O, and as values to parameters in Android API calls. Due to the nature of forced-path execution, branches of code that are generally impossible to access will be entered and this may provide a result that is a false positive. We do not have any mechanism to differentiate branches that are unreachable under normal circumstances.

VII. Conclusion

We further developed a framework for exploring a potentially large-state space contained within an Android application. We exercised our approach by testing a large number of Android applications with our framework to exhibit its functionality and viability. This approach affords scalability and automation in analyzing Android applications since it does not require user interaction or an Android-enabled device. We plan to overcome the increase coverage of the Android API by investigating novel approaches which do not rely on coding a manual implementation of the Android API calls. The framework can also be used for program validation by determining what behavior can be exhibited by the application. The program also has various user set parameters that enable a trade-off between performance and the precision of the analysis. The program can serve as a an extensible basis to serve other useful purposes such as an interactive debugger, symbolic execution, and any other approach that requires analysis of an Android application to be performed on a computer.

References

[1] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Androidpermissions:User attention, comprehension, and behavior. In Proceedings of the 2012 Symposium on Usable Privacy and Security (SOUPS), 2012.

[2] P. G. Kelley, S. Consolvo, L. F. Cranor, J. Jung, N. Sadeh, and D. Wetherall. A Conundrum of Permissions: Installng Applications on an Android Smartphone. In Workshop on Usable Security (USEC), 2012.

[3] android-apktool - A tool for reengineering Android apk files. http://code.google.com/p/android-apktool/.

[4] smali - An assembler/disassembler for Android's dex format. http://code.google.com/p/smali/.

[5] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. Technical report, Carnegie Mellon University, 2007.

[6] J. Wilhelm and T. Chiueh. A Forced Sampled Execution Approach to Kernel Rootkit Identification. In Proceedings of the Symposium on Recent Advances in Intrusion Detection, 2007.

[7] S. Lu, P. Zhou, W. Liu, Y. Zhou, and J. Torrellas. PathEx- pander: Architectural Support for Increasing the Path Coverage of Dynamic Bug Detection. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Micro- architecture (MICRO), 2006.

[8] Z. Wang, R. Johnson, R. Murmuria, and A. Stavrou. Exposing Security Risks for Commercial Mobile Devices. In Proceedings of the 6th International Conference Mathematical Methods, Models, and Architectures for Computer Network Security (MMM-ANCS 2012), October 2012.

[9] R. Johnson, Z. Wang, J. Voas, and A. Stavrou. Exposing Software Security and Availability Risks For Commercial Mobile Devices. To Appear In Proceedings of the The Annual Reliability and Maintainability Symposium (RAMS 2013), January 2013.

[10] L. Xu, F. Sun, and Z. Su. Constructing precise control flow graphs from binaries. Technical report CSE-2009-27, Department of Computer Science, UC Davis, 2009.

[11] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. ACM Computing Surveys, 24(3):293 318, Sept. 1992.

[12] P. Oehlert. Violating Assumptions with Fuzzing. IEEE Security and Privacy, 3(2), March 2005.

[13] J. C. King. Symbolic execution and program testing. Commun. ACM, 19(7):385-394, 1976.

[14] Z. Zhu, G. Lu, Y. Chen, Z. Fu, P. Roberts, and K. Han. Botnet research survey. In 32nd Annual IEEE Int. Computer Software and Applications (COMPSAC 08), pages 967972, 2008.

[15] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction. In Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS), pages 128-138, Oct. 2007.

[16] UI/Application Exerciser Monkey — Android Developers. http://developer.android.com/tools/help/monkey.html.

[17] I. Burguera, U. Zurutuza, and S. Nadjm-Therani. Crowdroid: behavior-based malware detection system for android. In Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM11), 2011.

[18] 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, CCS 11, pages 627-638, New York, NY, USA, 2011.

[19] R. Johnson, Z. Wang, C. Gagnon, A. Stavrou. Analysis Android Applications' Permissions. In: Proceedings of the 6th International Conference on Software Security and Reliability ,2012.

[20] D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton. The Combinatorial Design Approach to Automatic Test Generation. IEEE Software, 13(5):8389, September 1996.

[21] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In NDSS, 2008.

[22] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing Malware with Virtual Machines. In Proceedings of the 2006 IEEE Symposium on Security and Privacy, May 2006.

[23] dex2jar - Tools to work with android .dex and java .class files. http://code.google.com/p/dex2jar/.

[24] Activity Manager — Android Developers. http://developer.android.com/reference/android/app/ActivityManager.html.

[25] Activity — Android Developers. http://developer.android.com/reference/android/app/Activity.html.

[26] Thomas Blasing, Aubrey-Derrick Schmidt, Leonid Batyuk, Seyit A. Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In 5th International Conference on Malicious and Unwanted Software (Malware 2010) (MALWARE2010), Nancy, France, France, 2010.

[27] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. London Mathematical Society, 42(2):230-265, 1936.

[28] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In Proceedings of the 19th Annual Network and Distributed System Security Symposium, NDSS 12, February 2012.

[29] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-Based Malware Detection System for Android. In Proceedings of the 1st Workshop on Security and Privacy in Smartphones and Mobile Devices, CCS- SPSM11, 2011.

[30] Premium Rate SMS Trojans in Google's Android Market - F-Secure Weblog : News from the lab. http://www.f-secure.com/weblog/archives/00002280.html

[31] dshield Home — DShield; Cooperative Network Security Community - Internet Security .http://www.dshield.org/.

[32] Bytecode for the Dalvik VM. http://source.android.com/tech/dalvik/dalvik-bytecode.html.

[33] Android.Exprespam — Symantec. http://www.symantec.com/security_response/writeup.jsp?docid=2013-010705-2324-99&om_rssid=sr-latestthreats30days.