HyDroid: A Hybrid Approach for Generating API Call Traces from Obfuscated Android Applications for Mobile Security

Kobra Khanmohammadi
Software Behaviour Analysis (SBA) Research Lab
Department of Electrical and Computer Engineering
Concordia University, Montréal, QC, Canada
k_khanm@ece.concordia.ca

Abdelwahab Hamou-Lhadj
Software Behaviour Analysis (SBA) Research Lab
Department of Electrical and Computer Engineering
Concordia University, Montréal, QC, Canada
abdelw@ece.concordia.ca

***Abstract*—The growing popularity of Android applications makes them vulnerable to security threats. There exist several studies that focus on the analysis of the behaviour of Android applications to detect the repackaged and malicious ones. These techniques use a variety of features to model the application's behaviour, among which the calls to Android API, made by the application components, are shown to be the most reliable. To generate the APIs that an application calls is not an easy task. This is because most malicious applications are obfuscated and do not come with the source code. This makes the problem of identifying the API methods invoked by an application an interesting research issue. In this paper, we present HyDroid, a hybrid approach that combines static and dynamic analysis to generate API call traces from the execution of an application's services. We focus on services because they contain key characteristics that allure attackers to misuse them. We show that HyDroid can be used to extract API call trace signatures of several malware families.**

***Keywords*—Android API call traces, Repackaging, Android Applications, Static and Dynamic Analysis of Apps, Mobile Security**

I. INTRODUCTION

Users usually trust the application stores to provide them with mobile applications that are secure. The problem is that it is possible for an attacker to download an application, manipulate its content by introducing malicious code, repackage it, and upload it into app stores. Repackaging is considered among the top ten risks for mobile devices [45]. In addition, Zhou et al. [1] showed that more than 85% of malwares are introduced in applications through application repackaging. A recent incident is reported in December 2015, where the malware "GhostPush" was discovered in 39 different Android repackaged apps [2].

There exist studies (e.g., [9, 10, 11, 12]) that aim to detect automatically repackaged (and malicious) applications. These studies vary depending on whether they rely on static or dynamic analysis techniques. The common practice is to extract attributes of an application such as opcodes, APIs, images, resources, user interface graphs and use them to model the application's behaviour. Among these features, API calls that are invoked by the application components seem to be the most reliable [22]. This is because it is difficult for an attacker to manipulate API calls.

Generating API call traces from an Android application is a challenging problem because the source code of repackaged applications is rarely available. Besides, these applications are often obfuscated, which hinders static analysis of source code. In particular, encryption and reflection, two widely used obfuscation techniques [25], make it almost impossible to analyze the code of a repackaged application.

In this paper, we present a technique for generating API call traces from the execution of Android applications despite the presence of obfuscation. Our approach, called HyDroid, does not need access to the source code either. HyDroid combines static and dynamic analysis techniques. We show how our approach can be used to generate API call traces from the execution of the service components of applications. We focus on the service component because we showed, in our previous work [4], that there is a high possibility of finding malwares by studying the application's services as opposed to other components.

HyDroid encompasses in two phases. The first phase (static analysis phase) consists of modifying and instrumenting the application's binary files. We achieve this by transforming the binary of an application into a higher-level representation using the Jimple grammar [24]. Once instrumented, in the second phase (dynamic analysis), we run the application by exercising all possible paths and generate the API call traces. We tested our approach to generate API call signatures of malicious service components of applications infected by fourteen families of malware.

The remainder of this paper is organized as follows. In the following section, we provide the background needed to understand Android apps. Section III contains an overview of the HyDroid approach. In Section IV, we present the result of our evaluation. We discuss threats to validity in Section V, followed by related work in Section VI. We conclude the paper and present future work in Section VII.

II. BACKGROUND ON ANDROID APPLICATIONS

A typical Android application contains four types of components: activities, services, content providers and broadcast receivers. Briefly, activities represent what the user can do with the application while providing user interfaces. Services are components that run operations in the background. Content providers are used to manage a shared set of application data. The broadcast receivers respond to system-wide broadcast announcements such as "the battery is low".

A service is usually defined to perform long-running operations or to perform work for remote processes such as playing music in the background, or fetching data over the network. In AndroidManifest.xml, a service is defined by a <*service*> tag containing some attributes. It is implemented as the subclass of class *Service* in Java code and the class

name has to be similar to the name defined in AndroidManifest.xml for the service.

An Android application, typically developed in Java, is uploaded as a zip file with suffix .apk in app stores. This zip file, generally, contains an application program as classes.dex, as well as application resources like pictures, music and xml files, which describe the layout information. It must also contain AndroidManifest.xml, containing information about the application such as the name, version, access rights, and referenced libraries of the app. Classes.dex is in the format of Dalvik Virtual Machine bytecode provided in Android.

III. THE HYDROID APPROACH

Figure 1 shows the phases of our approach for generating API call traces from the execution of the application services.

Phase 1: Static Analysis

In this phase, we modify the application code to make it possible to exercise the various code paths through a service life cycle. We achieve this goal by going through the methods of the services and modifying the if-statement conditions. If we can control the conditions, we can force a method to run following a specific code path. We change the conditions with simple Boolean parameters that can be used as passed input to the service methods of an application. This will provide the analyst with a way to control the code so as to run it following a desired path.

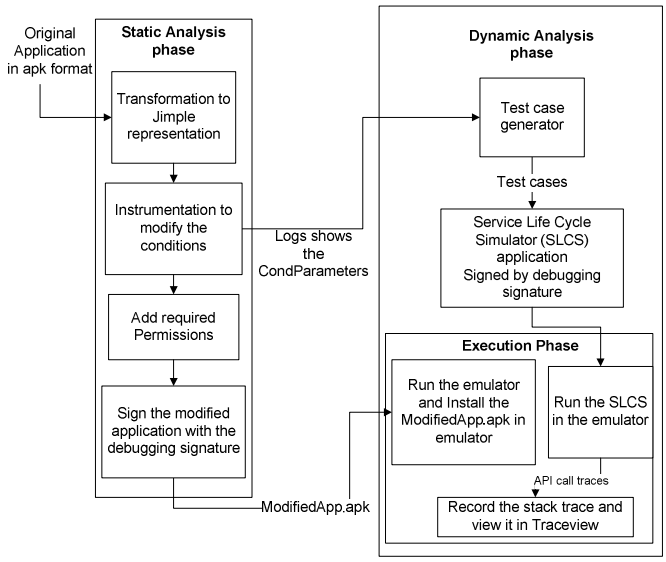

Figure 1. HyDroid Approach for extracting API call traces from Android app services in the presence of obfuscation

The example in Figure 2 shows the changes we apply to if-statements. In brief, three types of manipulations are done to if-statements. First, the whole condition is replaced by a Boolean variable. Second, assignment statements are done inside the if-statement body. Finally, exception handling is added. The condition in an if-statement is replaced by a Boolean variable called CondParameter. We will use CondParameter to exercise various paths of the code. For the second manipulation, we need to assign an acceptable value for *i* (used in the operations of the body of if-statement) to run the operations within the if-statement correctly. Assigning a correct value to *i* is not always possible because, for example, of the use of objects. In this case, we apply the third manipulation to handle possible exceptions by try-catch statements.

Example of conditions	The conditions after applying the changes

if (i > 100) {
f(i);
// rest of Operations;
}
	if (CondParameter) {
	i = 100 + ε;
	CondParameter = false;
	f(i);
	// rest of Operations;
	}

if(i≠ Null){
f(i);
// rest of Operations;
}
	if (CondParameter) {
	try{
	f(i);
	}catch(Exception e){
	e.printStackTrace();
	}
	CondParameter = false;
	// rest of Operations;
	}

Figure 2. An example of a condition in the code that has been modified

Since we do not have access to the source code, we need to manipulate the dex (binary) representation of the applications. We achieve this by using the soot framework [24]. More particularly, we transform the dex files into a Jimple representation [24]. Jimple was designed to facilitate the analysis of dex and bytecode files. The Jimple grammar for ifStmt is:

ifStmt → if conditionExpr goto label;
conditionExpr → leftOp condop rightOp
condop → < | > |= | ≠ | ≤ | ≥

In Jimple all variables are written in registers. In conditionExpr, leftOp (the left operand of the condition) is a register that will be compared by condop with rightOp (the right operand). We will evaluate the rightOp and its type and based on the condition operation, an acceptable value will be assigned to the register. Jimple also handles complex conditions and calls to functions in conditions. It achieves this by doing all the operations of the rightOp before the if-statement and assigning the result to a register, which will be placed as rightOp.For cases where the right operation is a primitive type, we assign a fake value to them by using assignStmt in Jimple. This strategy does not work for objects since we would not know which fake values to assign to them. To overcome this, we simply add a try/catch block and intercept the exception that would rise for an incorrect assignment to objects. We use the *Trap* class in soot to achieve this. We print the trace inside the catch block and allow the execution to continue.

Figure 3 shows an example of the transformations made to the pseudo code of a simple application. This program executes a malicious operation, which consists of sending data to a predefined phone when the application is installed on a real device. The application starts by checking if Playstore is installed. It then reads the list of packages in the device and then checks for finding package "vending", which belong to the Playstore application. As shown in Figure 3, we change the conditions to Boolean variables and add an exception handler to print out the trace.

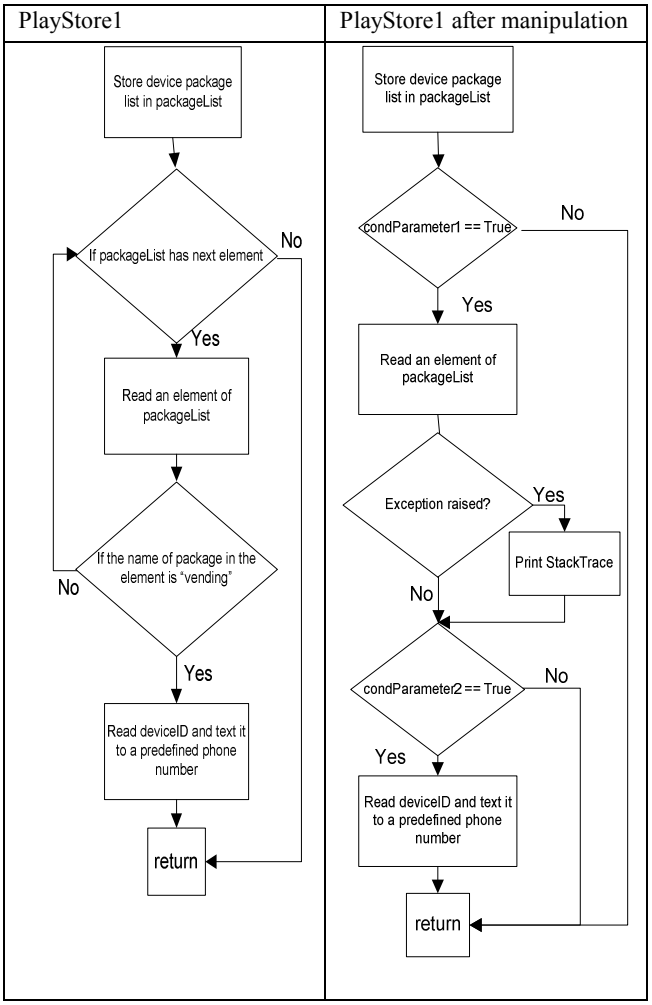

Figure 3. EmulatorDetection_PlayStore1 before and after manipulation in static analysis phase

In our modification of the code, to save space, we represent all the conditions found in the service method by a bit in our test case suite generation. For example, for five ifStmt, CondParameter can get all the combinations of values for five bits. CondParameter that has the value "10011" in binary shows that the first, fourth and fifth conditions are true and the second and third conditions are false.

Loops in Jimple are represented by ifStmt statements and gotoStmt. As our goal is to only extract the API call traces in all code paths, as it is shown in Figure 2, we assign false value to the condition inside the body of ifStmt in order to execute the loops only once. Switch statements are handled the same way as if-statements. We simply turn them into multiple if-statements and apply the same method as the one used to run simple if-statements.

After modifying the code conditions using soot, a new apk containing all the changes is generated. It is required to add permission for writing to the external storage in order to record the trace including API calls later on during execution of the application. We use a tool called APKtool to add the necessarily permissions to the AndroidManifest.xml file of the application.

Phase 2: Dynamic Analysis

The first step of this phase is to generate test cases to run the modified app. This consists of generating test values for CondParameters. We simply apply different combinations of values (True and False) for CondParameters for every execution to obtain good path coverage.

To run the test cases, we developed a program, that we call the Service Life Cycle Simulator (SLCS) application. SLCS simulates the life cycle of a service to interact with the services in the tested application. This application should have the same signature as the modified application for the two applications to run in an emulator.

While starting a service or bounding to it, some callback methods are called by Android during the life cycle of a service. We implement the service life cycle by calling the callback methods and providing test cases to start or bind to the service.

When the service is bound, apart from the callback methods, there are interactions between the service and the component that is bound to it. The component can call public methods of the service. The information such as implemented service calling approach in the application can be extracted in the static analysis phase using the soot library by analyzing the methods of the service components.

For simulating the service life cycle, we used Android JUnit testing for calling callback methods and simulating the execution of a service when it was started or bound. The Java reflection technique is also used to call service methods at run-time since we do not have access to the source code of the modified apps and we cannot directly call them in our testing code through SLCS. The class Debug and the tool Traceview[1] are used to generate and view the traces while running the simulation.

Note that apart from callback methods in a service life cycle, there is a set of callback APIs that are common to all application components including onConfigurationChanged() and onLowMemory(). It is necessary to consider them and call them as public methods while binding to a service.

Finally, it is worth mentioning that the SLCS application and the modified application are installed in an emulator with the same signed signature. This way, the components of the modified application can be called by SLCS. SLCS will start services in the modified application using a variety of values for CondParameters. The traces will be recorded using the Debug class during execution. We used Traceview to read and process the recorded traces. We prune the resulting traces so as to keep only the Android API calls and calls to the Java library that can reveal the behaviour of a malware.

IV. CASE STUDY

In the following case study, we assess the effectiveness of HyDroid to extract API call traces from malicious applications. As discussed earlier, we focus on the service components of applications and this case study looks at malware families, which contain malicious operations in

[1] https://developer.android.com/studio/profile/traceview.html

TABLE I. DISTRIBUTION OF OBFUSCATED AND NON-OBFUSCATED APPLICATIONS IN EACH MALWARE FAMILY IN GENOME DATASET

Malware Family	No. of apps	Obfuscation used	Num. of non-obfuscated apps
BgServ	9	9 app samples with name alteration and reflection	0
BeanBot	8	8	0
DroidKungFu1	34	None	34
DroidKungFu2	30	None	30
DroidKungFu3	309	None	309
GingerMaster	4	4 app samples with name alteration and reflection	0
GoldDream	47	6 sample with only name alteration and 13 samples with name alteration, reflection and encryption	28
GPSSMSSpy	6	None	6
HippoSMS	2	2 app samples with name alteration	0
NickySpy	2	None	2
Pjapps	58	None	58
Plankton	11	11app samples with name alteration and reflection	0
SndApps	10	None	10
Tapsnake	2	None	2

service components of Android applications. Note that a malware may manifest itself in other components as well. This case study addresses the following research question:

RQ2: Can we use HyDroid to extract API call signatures of a malware?

A. Dataset

We use the Gnome repository [44] which contains over 1200 malware samples, categorized by malware families. We selected fourteen malware families that manifest their malicious operation mainly through the service components of an application and contain more than one sample in the dataset so that we can find the correct malicious service by comparing the service code in different samples.

All selected malware samples read device's confidential data such as DeviceId and IMEI number, geographical location, received SMS or Phone call data and send them to the remote server. While some malware families [48] including GolDream and GPSSMSSpy, NikySpy , SndApps and Tapsnake finish their malicious operations to leakage of such confidential data, some other families, namely, BgServ, BeanBot, DroidKungFu families, GingerMaster, Pjapps, Plankton continue performing variety of operations by getting commands from the server. Different commands in samples include downloading file, installing payload, bloc king message, sending message to premium numbers, changing remote server address, sending GPS location, etc. HippoSMS [48] was implemented to send message to premium numbers. While there are four samples in Genome dataset, we carry out our experiment on only two samples, which use service component to host malicious operations.

For each malware family, the dataset contains applications that are obfuscated and others that are not. This allows researchers to use the non-obfuscated applications as a validation set.

To check the obfuscation used in applications, we retrieved the source code, using dex2jar[2] to covert a dex file into Java. The distribution of obfuscated and non-obfuscated applications of each malware family is shown in Table I.

B. Evaluation

To evaluate the effectiveness of HyDroid in extracting API call signatures from the execution of the services infected by these families of malware we have performed the following steps. In Step 1, we extracted the API call traces in methods through the life cycle of a malicious service in malware application samples by using Library Soot [24] and Infoflow [35]. Then, in Step 2, we extracted the API call traces using HyDroid and. Finally, in Step 3, we compared the API call traces extracted by HyDroid to the ones recorded in the first step.

We used Infoflow [35] in the Soot library to extract the call graph through the life cycle of a service in Step 1. Infoflow provided the call graph rooted from methods named as the starting point. As we want to extract the API calls through a service life cycle, we assigned callback methods in a service as staring points, including, onCreate(), onStart(), onDestroy(), onBind(). If onBind() is present, it is necessary to add public methods since they can be called directly like some samples in DroidKungFu2. For all the methods in the call graph of a service, we extracted the post dominator of all API calls through the control flow of each method. The post-dominator records the sequence of APIs called successively in a code path of a method.

In Step 2, we extracted the API call traces using HyDroid as explained in Section III. To evaluate whether the extracted API call trace is available in the extracted API call traces in Step 1, we looked for the longest sequence of API calls presents in the trace and in the post-dominator of the starting point method. Then, we removed that sequence from the trace and continued the comparison of the remaining APIs in the trace in the post-dominators of

[2]https://github.com/pxb1988/dex2jar

successor methods of the starting point method in the call graph. Our algorithm recursively continued such similar comparison for the rest of API calls in the successor methods in the call graph and its post-dominator.

To run HyDroid, we needed to exercise the obfuscated applications (the testing set) with different values of CondParameter conditions to be able to cover most of the execution paths. Here, in our experiment, applications with obfuscated techniques which has name alteration and also reflection methods while passed string parameters which showing the name of the called method are not encrypted, allowed us to use them in testing set. Because we are using Infoflow in Step 1, we can extract the indirectly called methods and the APIs they call.

C. Results

Table III shows the number of distinct API calls for each malware family. The entire API call traces of all malware families can be accessed online[3]. All the samples in these malware families have one malicious service except NikySPy which has seven malicious services.

TABLE II. NUMBER OF DISTINCT API CALLS FOR EACH MALWARE FAMILY.

No.	Malware Family Name	Number of distinct API calls
1	BgServ	106
2	BeanBot	29
3	DroidKungFu1	104
4	DroidKungFu2	82
5	DroidKungFu3	116
6	GingerMaster	70
7	GoldDream	72
8	GPSSMSSpy	45
9	HippoSMS	27
10	NickySpy	251
11	Pjapps	105
12	Plankton	71
13	SndApps	44
14	TapsnaKe	22

As an example, Figure 4 shows the post-dominators for the method CheckAndClearFile() in GoldDream. Based on post-dominator, all the APIs inside a curvy bracket are to be called consecutively before the method goes to exit.

In Figure 5, a sample trace of API calls extracted by applying HyDroid to the GoldDream-infected apps. The trace shows the API calls made by the infected code. The GoldDream malware starts by reading confidential data such as device ID and IMEI (see section marked (1) in the trace of Figure 5) and, then establishing a connection with a remote server (shown in (2)). It then reads the file containing recorded message and phone call information by a broadcast receiver component and send its content to the remote server (shown in (3)).

The experiments show that HyDroid can extract API call traces that characterize the malicious code from obfuscated apps. However, we recognize that it is time consuming to go through all "conditions statements" in order to exercise all the paths of the program. It is necessarily to improve the approach to selectively instrument only condition statements through which to execute the code. This is deferred as part of future work.

```
{entry,<com.GoldDream.zj.zjService:boolean
fileIsExists(java.lang.String)>,exit}
{<com.GoldDream.zj.zjService:java.lang.String
getKeyNode(java.lang.String,java.lang.String)>,
<java.lang.Integer: java.lang.Integer
valueOf(java.lang.String)>,
<java.lang.Integer: int intValue()>,
<java.io.FileInputStream: void <init>(java.lang.String)>,
<java.io.FileInputStream: int available()>,
<java.io.FileInputStream: void close()>, exit}
{<java.lang.String: int lastIndexOf(java.lang.String)>,
<java.lang.String: int length()>,
<java.lang.String: java.lang.String substring(int,int)>,
<android.content.ContextWrapper: java.io.FileOutputStream
openFileOutput(java.lang.String,int)>,
<java.lang.String: void <init>(java.lang.String)>,
<java.lang.String: byte[] getBytes()>,
<java.io.FileOutputStream: void write(byte[])>,
<java.io.FileOutputStream: void close()>, exit}
```

Figure 4. Post-dominator of method CheckAndClearFile() in GoldDream

```
(1)
android.telephony.TelephonyManager.getSystemService(Ljava.lang.Class;)Androi
d.telephony.TelephonyManager;
android.telephony.TelephonyManager.getDeviceId()Ljava.lang.String;
android.telephony.TelephonyManager.getSubscriberId()Ljava.lang.String;
android.telephony.TelephonyManager.getSimSerialNumber()Ljava.lang.String;
android.telephony.TelephonyManager.getLine1Number()Ljava.lang.String;
…
(2)
Ljava.net.HttpURLConnection.openConnection()Ljava.net.HttpURLConnection;
Ljava.net.HttpURLConnection.getResponseCode()I;
Ljava.net.HttpURLConnection.getInputStream()Ljava.io.InputStream;
…
(3)
Ljava.net.HttpURLConnection.setDoInput(Z)V;
Ljava.net.HttpURLConnection.setDoOutput(Z)V;
Ljava.net.HttpURLConnection.setUseCaches(Z)V;
Ljava.net.HttpURLConnection.setRequestMethod(Ljava.lang.String)V;
Ljava.net.HttpURLConnection.setRequestProperty(Ljava.lang.String;
Ljava.lang.String)V;
Ljava.net.HttpURLConnection.getOutputStream()Ljava.io.outputStream;
Ljava.io.InputStream.read(Ljava.lang.CharSequence;)I;
Ljava.io.outputStream.write(Ljava.lang.CharSequence;I;I)V;
android.content.ContextWrapper.openFileOutput(Ljava.lang.String;I;)Ljava.io.File
OutputStream
Ljava.io.FileOutputStream.write(Ljava.lang.CharSequence;)V;
Ljava.io.FileOutputStream.close()V;
android.content.ContextWrapper.openFileOutput(Ljava.lang.String;I;)Ljava.io.File
OutputStream
Ljava.io.FileOutputStream.write(Ljava.lang.CharSequence;)V;
Ljava.io.FileOutputStream.close()V;
Ljava.net.HttpURLConnection.openConnection()Ljava.net.HttpURLConnection;
Ljava.net.HttpURLConnection.getResponseCode()I;
Ljava.net.HttpURLConnection.getInputStream()Ljava.io.InputStream;
android.content.ContextWrapper.getSharedPreferences(Ljava.lang.String;I;)Landr
oid.content.SharedPreferences;
android.content.SharedPreferences.edit(Ljava.lang.String;I;)Landroid.content.Shar
edPreferences.Editor;
android.content.SharedPreferences.Editor.commit()Z;
```

Figure 5. An excerpt from a sample API call trace of GoldDream

[3]http://www.ece.concordia.ca/~abdelw/sba/qrs17

A. Discussion

We showed that our approach, HyDroid, works well in generating API calls from the execution of application services. Our approach is particularly useful to overcome obfuscation. There exist different obfuscation techniques, among which reflection, encryption, and control flow alteration, are the most advanced ones [25]. We discuss in how HyDroid addresses these obfuscation techniques in the following sections.

Reflection: Reflection is handled well in HyDroid because the reflective methods are called while the program is running. Our approach makes it possible to run the code through different code paths. If there is a reflective call in a code path, it will be captured in the log. The results show that HyDroid can go through all the reflective methods while executing code and extracting the API call traces. To exemplify, the following code in application Reflection3 form DroidBench[5] , used reflection to create an object of the class "de.ecspride.ReflectiveClass" and call its methods "setIme" and "getImei".

```
TelephonyManager telephonyManager =
(TelephonyManager)getSystemService(Context.TELEPHONY_SER
VICE);
String imei = telephonyManager.getDeviceId(); //source
Class c = Class.forName("de.ecspride.ReflectiveClass");
Object o = c.newInstance();
Method m=c.getMethod("setIme" +"i", String.class);
m.invoke(o, imei);
Method m2 = c.getMethod("getImei");
String s = (String) m2.invoke(o);
SmsManager sms = SmsManager.getDefault();
sms.sendTextMessage("+49 1234",null,s,null, null);   //sink, leak
```

It is possible to carry out static analysis by reading the code for finding the name of classes and methods passed in reflective package and changing all reflective calls to a direct call. However, this approach will not always work for a variety of reasons. First, the method name and class name can be created at run-time and are not static like the example above. Second, if the strings passed to the reflective methods such as “setIme” and “getImei” (as in the example above) are encrypted, they will need to be decrypted first.

Encryption: Encryption, another powerful obfuscation technique, can be used in different points of the code including encryption of strings, encryption of the whole class or encryption of resources of the application. As mentioned earlier, when string encryption and reflection can be used together, our approach works well to defeat the string encryption. In case the whole class is encrypted, instrumentation without decrypting the class is impossible. This type of obfuscation remains a challenge even for HyDroid.

Control Flow Alteration: Another obfuscation technique relies on adding some code to the original code. Sometimes this code is a dead code and is never executed. Therefore, it will not alter the execution of the operations in the original application but it makes it hard to read and analyze the code after obfuscation. This kind of obfuscation may cause HyDroid to instrument unnecessarily code, generating unneeded API calls. This problem, however, does not affect the detection of malicious apps using the generated API call traces. It only increase the computation time of the detection approach.

V. Threats to Validity

A threat to internal validity exists in the implementation of our approach, especially the simulator component (SLCS) we developed to exercise the various scenarios of an application. It is possible that an incorrect implementation may cause variation in results. However, we have mitigated this threat by manually reviewing the code and working through many examples. Another threat to validity exists in the way we assessed the results of HyDroid when applied to the fourteen families of malware in Case Study. We used a combination of binary code analysis and trace inspection. We mitigated this threat by carefully examining each segment of a trace and comparing it to the API calls in the code as well as to the description of these malwares provided in the Gnome documentation. A threat to external validity exists in generalization of our approach to other datasets. We only evaluated our approach on fourteen families of malwares, and further experiments with other family of malwares are needed to generalize the results of this study.

VI. Related work

Since our proposed approach combines static and dynamic analysis techniques, studies that apply static or dynamic analysis to Android applications are related to our work. There existing several studies that rely on pure static analysis for examining malware behaviour. These studies use various features including API calls [27, 28, 29, 18 and 46], strings contained in the applications [31 and 47], instruction code sequences [11], code chunks [17], and method call graphs [8, 12, 19].

Hanna et al. in [10] used n-grams of Dalvik opcode sequences to find similar codes with the goal of detecting buggy and vulnerable code reuse. Zhou et al. in [9] used static analysis to extract instruction code sequences for measuring the similarity among applications. ViewDroid [14] was proposed to identify similar applications based on similar activity graphs of applications. Crussell et al. [13] used static data dependency graph (DDG) of every method to detect repackaged applications.

As discussed earlier, static analysis approaches suffer from their inability to overcome reflection calls. Moreover, some malware such as DroidKungFu4 [48] extract the actual malicious payload from external. Thus, static analysis approaches cannot detect them. In this sense, to detect the malicious operations implemented in the code loading dynamically and studying the behaviour of native code, it is often suggested to proceed with dynamic analysis. Dynamic analysis is also a viable solution for the problem of code obfuscation as we showed in this paper. CopperDroid [20], CrowdDroid [30], AASandbox [33] and an on-device approach [3] presented dynamic analysis approaches to extract system calls and their parameters. Detecting malicious behaviour by extracting API call traces is also done by dynamic analysis in DroidRanger [32], TaintDroid [42] and DroidRanker [43]. These approaches execute an application or part of the code to find the signature of malicious behaviour known earlier.

Hybrid approaches have also been proposed. Rasthofer et al. in [34] presented a tool called Harvester in which static analysis is applied to extract parts of the code containing reflective calls. Then, they used dynamic analysis to run that part of the code to identify the method name called by reflection. Their proposed approach in combination with a static analysis tool, FlowDroid [35], is used for detecting malware. IntelliDroid [36] proposed for malware detection by locating the sensitive APIs and identifying the inputs needed to be passed to the specific methods in static analysis and using dynamic analysis to ran the code with the identified inputs. SMV-Hunter [37] first used static analysis to identify the suspicious vulnerable application and then applied dynamic analysis to detect codes vulnerable to Man-in-the-Middle attack. SmartDroid [38] identified the code paths from UI-based conditions into the sensitive behaviour in static analysis phase and applied dynamic analysis to expose the malicious operation. APPIntent [39] proposed hybrid analysis to determine data transmissions which is not triggered by the application's user. AppAudit [40] and ContentScop [41] used a combination of static and dynamic analysis approach for detecting a particular kind of malware. HyDroid is a generic approach that is designed to be used for any attacks that manifest themselves through the app's services.

VII. Conclusion and Future Work

A hybrid analysis approach is presented to utilize the benefits of static and dynamic analysis in order to provide an effective way to extract API call traces of a service component of Android applications. The proposed approach is designed to simulate the life cycle of a service component and collect logs of the called Android API calls. To have control over the code paths, static analysis is used to manipulate the control flow of the program through changes to the code conditions. We used the soot framework to analyze the binary files of the apps after transforming them into a higher-level representation using the Jimple grammar. We were also able to identify API call signatures of fourteen families of malware. Our approach, however, suffers from some limitations. First, it does not handle native code. Native code can be used for hiding malicious operation by malware [26]. In addition, if the code contains dynamic loading, instrumenting that part of the code is not possible. Finally, HyDroid exercises all code paths of a service method, which may caus scalability problems. We intend to address these limitations as part of future work.

Ackowledgement

This work is partly supported by the Natural Sciences and Engineering Council of Canada (NSERC).

References

[1] Y. Zhou and J. Xuxian, "Dissecting android malware: Characterization and evolution," Proc IEEE Symp. Security and Privacy (SP), IEEE press, 2012, pp. 95-109.

[2] Hidden Malware and the Ghosts of Mobile Technology, 2015, http://blog.cyren.com/articles/hidden-malware-and-the-ghosts-of-mobile-technology.html

[3] M. B. Attia, C. Talhi, A. Hamou-Lhadj, B. Khosravifar, V. Turpaud, and M. Couture, "On-device anomaly detection for resource-limited systems," Proc. of the 30th Annual ACM Symposium on Applied Computing, ACM, 2015, pp. 548-554.

[4] K. Khanmohammadi, M. Rejali, and A. Hamou-Lhadj, "Understanding the Service Life Cycle of Android Apps: An Exploratory Study," Proc. the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, ACM, 2015, pp. 81-86.

[5] https://github.com/secure-software-engineering/DroidBench

[6] https://github.com/douglascraigschmidt/POSA-15

[7] https://github.com/pxb1988/dex2jar

[8] M. Abdellatif, C. Talhi, A. Hamou-Lhadj, and M. Dagenais, "On the use of mobile GPU for accelerating malware detection using trace analysis," Proc. of the IEEE 34th Symposium on Reliable Distributed Systems Workshops (SRDSW), IEEE, 2015, pp. 42-46.

[9] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," Proc. of the second ACM conference on Data and Application Security and Privacy, ACM, 2012, pp. 317-326.

[10] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications," In Detection of Intrusions and Malware, and Vulnerability Assessment, Springer Berlin Heidelberg, 2012, pp. 62-81.

[11] H. Shahriar, and V. Clincy, "Detection of repackaged Android Malware," Proc. of the 9th International Conference for Internet Technology and Secured Transactions (ICITST), IEEE, 2014, pp. 349-354.

[12] W. Hu, J. Tao, X. Ma, W., Zhou, S. Zhao, and T. Han, "Migdroid: Detecting app-repackaging android malware via method invocation graph," Proc. of the 23rd International Conference on Computer Communication and Networks (ICCCN), IEEE, 2014, pp. 1-7.

[13] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," Proc. of ESORICS, 2012, pp. 37–54.

[14] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "ViewDroid: Towards obfuscation-resilient mobile application repackaging detection," Proc. of the 2014 ACM conference on Security and privacy in wireless & mobile networks, ACM, 2014, pp. 25-36.

[15] S. Jiao, Y. Cheng, L. Ying, P. Su, and D. Feng, "A Rapid and Scalable Method for Android Application Repackaging Detection," Proc. of the Information Security Practice and Experience, Springer International Publishing, 2014, pp. 349-364.

[16] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged Android applications," Proc. of the 30th Annual Computer Security Applications Conference, ACM, 2014, pp. 56-65.

[17] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco, "Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families," Expert Systems with Applications, 41(4), 2014, pp. 1104-1117

[18] Z. Luoshi, N. Yan, W. Xiao, W. Zhaoguo, and X. Yibo, "A3: Automatic Analysis of Android Malware," Proc. of the 1st International Workshop on Cloud Computing and Information Security. Atlantis Press, 2013.

[19] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call graphs," Proc. of the 2013 ACM workshop on Artificial intelligence and security, 2013, ACM, pp. 45-54.

[20] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic Reconstruction of Android Malware Behaviors," Proc. of the Symposium on Network and Distributed System Security (NDSS), 2015, pp. 8-11.

[21] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: hindering dynamic analysis of android malware," Proc. of the Seventh European Workshop on System Security, ACM, 2014.

[22] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," Proc. of the 2012 ACM conference on Computer and communications security, ACM, October 2012, pp. 217-228.

[23] Android developer site, http://developer.android.com/index.html

[24] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: converting android dalvik bytecode to jimple for static analysis with soot," Proc. of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, ACM, 2012, pp. 27-38.

[25] H. Huang, S. Zhu, P. Liu, and D. Wu, "A framework for evaluating mobile app repackaging detection algorithms," Proc. of the Trust and Trustworthy Computing Conference, Springer Berlin Heidelberg, 2013, pp. 169-186.
[26] R. Fedler, M. Kulicke, and J. Schütte, "Native code execution control for attack mitigation on android," Proc. of the Third ACM workshop on Security and privacy in smartphones & mobile devices, ACM, 2013, pp. 15-20.
[27] M. Alazab, S. Venkataraman, and P. Watters, "Towards understanding malware behaviour by the extraction of API calls," Proc. of the Cybercrime and Trustworthy Computing Workshop (CTC), IEEE, 2010, pp. 52-59.
[28] R. Islam, and I. Altas, "A comparative study of malware family classification," Proc. Information and Communications Security, Springer Berlin Heidelberg, 2012, pp. 488-496.
[29] D. J. Wu, C. H. Mao, T. E. Wei, H. M. Lee, and K. P. Wu, "Droidmat: Android malware detection through manifest and API calls tracing," Proc. of the Seventh Asia Joint Conference on Information Security (Asia JCIS), 2012, pp. 62-69.
[30] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," Proc. the 1st ACM workshop on Security and privacy in smartphones and mobile devices , ACM, 2011, pp. 15-26.
[31] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, and P. G. Bringas, "On the automatic categorization of android applications," Proc. of the Consumer Communications and Networking Conference (CCNC), IEEE, 2012, pp. 149-153.
[32] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets," Proc. of the 19th Annual Network & Distributed System Security Symposium, 2012.
[33] T. Bläsing, L. Batyuk, A.D. Schmidt, S.A. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," Proc. of the 5th International Conference on Malicious and Unwanted Software (MALWARE), IEEE, 2010, pp. 55-62.
[34] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime data in android applications for identifying malware and enhancing code analysis," Technical Report TUD-CS-2015-0031, EC SPRIDE, 2015.
[35] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," Proc. of ACM SIGPLAN Notices, 49(6), 2014, pp. 259-269.
[36] M. Y. Wong, and D. Lie, "IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware", Proc. of the 23rd Annual Network and Distributed System Security Symposium (NDSS), 2016.
[37] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in androidapps," Proc. of the 21st Annual Network and Distributed System Security Symposium (NDSS), 2014.
[38] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications," Proc. of the 2nd ACM workshop on Security and privacy in smartphones and mobile devices, ACM, 2012, pp. 93-104.
[39] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection," Proc. of the 2013 ACM SIGSAC conference on Computer & communications security, 2013, pp. 1043-1054.
[40] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective real-time android application auditing," Proc. Security and Privacy (SP), 2015 IEEE Symposium on,(May 2015) 899-914.
[41] Y. Z. X. Jiang, "Detecting passive content leaks and pollution in android applications," Proc. of the 20th Network and Distributed System Security Symposium (NDSS), 2013.
[42] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. G. Chun, L. P. Cox, and A. N. Sheth, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," ACM Transactions on Computer Systems (TOCS), 32(2), ACM, (2014) 5.
[43] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," Proc. of the 10th international conference on Mobile systems, applications, and services, ACM, 2012, pp. 281-294.
[44] Android malware Genome project, http://www.malgenomeproject.org/
[45] https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks
[46] W. Fan, Y. A. Liu, and B. Tang, "An API Calls Monitoring-based Method for Effectively Detecting Malicious Repackaged Applications," International Journal of Security and Its Applications, 9(8), 2015, pp. 221-230.
[47] H. Gonzalez, A. A. Kadir, N. Stakhanova, A. J. Alzahrani, and A. A. Ghorbani, "Exploring reverse engineering symptoms in Android apps," Proc. of the 8th European Workshop on System Security, ACM, 2015.
[48] malware families, https://www.symantec.com/