# Droid-AntiRM: Taming Control Flow Anti-analysis to Support Automated Dynamic Analysis of Android Malware

Xiaolei Wang
College of Computer, National University of Defense Technology, China
xiaoleiwang@nudt.edu.cn

Sencun Zhu
Department of Computer Science and Engineering & College of Information Sciences and Technology, The Pennsylvania State University, USA
szhu@cse.psu.edu

Dehua Zhou
Department of Computer Science, Jinan University, China
tzhoudh@jnu.edu.cn

Yuexiang Yang
College of Computer, National University of Defense Technology, China
yyx@nudt.edu.cn

## ABSTRACT

While many test input generation techniques have been proposed to improve the code coverage of dynamic analysis, they are still inefficient in triggering hidden malicious behaviors protected by anti-analysis techniques. In this work, we design and implement *Droid-AntiRM*, a new approach seeking to tame anti-analysis automatically and improve automated dynamic analysis. Our approach leverages three key observations: 1) Logic-bomb based anti-analysis techniques control the execution of certain malicious behaviors; 2) Anti-analysis techniques are normally implemented through condition statements; 3) Anti-analysis techniques normally have no dependence on program inputs. Based on these observations, *Droid-AntiRM* uses various techniques to detect anti-analysis in malware samples, and rewrite the condition statements in anti-analysis cases through bytecode instrumentation, thus forcing the hidden behavior to be executed at runtime. Through a study of 3187 malware samples, we find that 32.50% of them employ various anti-analysis techniques. Our experiments demonstrate that Droid-AntiRM can identify anti-analysis instances from 30 malware samples with a true positive rate of 89.15% and zero false negative. By taming the identified anti-analysis, Droid-AntiRM can greatly improve the automated dynamic analysis, successfully triggering 44 additional hidden malicious behaviors from the 30 samples. Further performance evaluation shows that Droid-AntiRM has good efficiency to perform large-scale analysis.

## CCS CONCEPTS

• **Security and privacy** → *Malware and its mitigation*;

## KEYWORDS

Android Malware; Anti-Analysis; Dynamic Analysis; Symbolic Data Flow Analysis

**ACM Reference format:**
Xiaolei Wang, Sencun Zhu, Dehua Zhou, and Yuexiang Yang. 2017. Droid-AntiRM: Taming Control Flow Anti-analysis to Support Automated Dynamic Analysis of Android Malware. In *Proceedings of ACSAC 2017, San Juan, PR, USA, December 4–8, 2017,* 12 pages.
DOI: 10.1145/3134600.3134601

## 1 INTRODUCTION

To contend against the rapid growth of Android malware, many automated analysis techniques have been proposed. In general, static analysis [5, 11] is a very powerful technique since it can reason about all execution paths in an application. However, new Android malware has raised various challenges to static analysis, such as obfuscation, reflection and dynamic code loading, etc. For such reasons, automated dynamic analysis [10, 32, 36, 40, 43] has been advocated in the context of Android malware analysis.

However, the possible malicious behavior can only be detected only if it is executed during dynamic analysis. Therefore, many test input generation techniques have been proposed to improve the code coverage of dynamic analysis. 1) Random fuzzing [19, 41], which applies randomly generated inputs. 2) Concolic testing [2, 12, 31], which uses contrete values and symbolically derived path constraints to exercise different paths and generate test inputs. 3) Targeted test input generation [38], which relies on static analysis to generate a small set of test inputs and only trigger the malicious behavior that an analyst cares about.

```
public void onCreate() {
  this.tel = (TelephonyManager)
     getSystemService("phone").getLine1Number();
  this.imei = (TelephonyManager)
     getSystemService("phone").getDeviceId();
  this.sim = (TelephonyManager)
     getSystemService("phone").getSimSerialNumber();
  if(!check(this.imei)){ //emulator is detected if return false
     System.exit(1); }
  else{ //start to do malicious things;  }
}
```

**Figure 1: Code example of NickSpy**

Although previous test input generation techniques can improve code coverage during dynamic analysis, they still cannot guarantee all the malicious behaviors will be triggered when anti-analysis techniques are employed by malware writers. Figure 1 shows a code snippet from malware *NickSpy*. In line 5, it checks the IMEI value of the device through a self-defined method *check*. The method *check* issues a HTTP request carrying the retrieved IMEI of the device to a specified URL, and it returns *true* if the HTTP response contains 'y' in a certain field of its payload. The program will be shut down when method *check* returns *false* (meaning an emulator is detected). In this way, emulator detection based anti-analysis can defeat current dynamic analysis easily and hide malicious behaviors.

In addition to emulator detection, other logic-bomb based anti-analysis techniques, such as time bombs and location bombs [11], have been increasingly adopted by malware. These anti-analysis techniques complicate dynamic analysis seriously. The corresponding malicious behaviors can be triggered only when all the demands of anti-analysis techniques are met, or they will be hidden. To trigger more hidden malicious behaviors during automated dynamic analysis, it is important to automatically tame such anti-analysis techniques.

**Our work**. To address the above problem, in this work, we implement a system named *Droid-AntiRM*. Our approach leverages three key observations: 1) Logic-bomb based anti-analysis techniques must control the execution of certain malicious behaviors; 2) Anti-analysis techniques are normally implemented through condition statements; 3) Anti-analysis techniques normally have no dependence on the program inputs. It is noteworthy that currently, we only focus on control flow based anti-analysis, not including ones that use data flow. From now on, we use the term *anti-analysis* to generally describe control flow based anti-analysis. Accordingly, we aim to tame these anti-analysis techniques by focusing on the related condition statements, not anti-analysis techniques themselves. We consider a condition statement *potential anti-analysis* when it controls the execution of malicious behaviors, and it is considered as *true anti-analysis* if it has no dependence on program inputs. For detected true anti-analysis, Droid-AntiRM replaces their condition statements as simple Boolean variables through bytecode instrumentation. Additionally, Droid-AntiRM returns the detailed symbolic variable information used in anti-analysis. This not only greatly simplifies a human analyst's effort on deciding whether a reported anti-analysis case is true or not, but also helps the analyst discover new anti-analysis techniques.

Our evaluation with 3187 malware samples shows that 32.50% malware samples employ anti-analysis, indicating that it is urgent to defeat them for effective malware analysis. A depth evaluation on 30 selected malware samples shows that Droid-AntiRM can achieve a true positive rate 89.15% during anti-analysis detection. Particularly, through white-listing some known APIs, the true positive rate can be increased to 98.06%. A case study with IntelliDroid [38]also shows that Droid-AntiRM can significantly improve automated dynamic analysis. Furthermore, on average Droid-AntiRM takes less than 14 seconds per sample to extract the anti-analysis condition statements, which indicates that it is efficient enough for large-scale analysis.

**Contributions**. To the best of our knowledge, Droid-AntiRM is the first approach to detecting and taming control flow anti-analysis techniques in Android malware. In summary, we make the following three contributions in this paper:

- We propose a novel approach to detecting possible anti-analysis in Android malware, which can complement the previous work.
- We design and implement Droid-AntiRM to detect and tame the anti-analysis used by Android malware.
- Experiments show that our proposed method has a good accuracy and is efficient during anti-analysis detection. A case study shows that through our method can improve dynamic analysis by triggering more hidden malicious behaviors effectively.

## 2 PRELIMINARIES

### 2.1 Motivating Example

Logic bombs (including time bombs) have been a big challenge for automated dynamic analysis. Figure 2 shows a code snippet taken from DroidKungFu [17], one of the most widespread malware families. In the first step, the service *SearchService* is started when receiving a BOOT_COMPLETED intent (code lines 3-6). When the service starts for the first time, it writes the current time in SharedPreferences and stops itself (code lines 7-11). Every time this service is restarted, it checks if the elapsed time between the current time and stored time has exceeded 24 hours. If so, the malicious behavior is triggered (code lines 13-17), or else the program is stopped (code line 12). In this way, a time bomb is constructed to hide the malicious behavior.

```
1  public class Receiver extends BroadcastReceiver{
2  public void onReceive(Context context, Intent intent){
3    if (intent.getAction().equals
          ("android.intent.action.BOOT_COMPLETED")){
4      Intent in = new Intent();
5      in.setClass(context, SearchService.class);
       // Starts com.google.search.SearchService.
6      context.startService(in);}
   //com.google.search.SearchService initializes.
7  public void onCreate(){
8    SharedPreferences p = getSharedPreferences("sstimestamp", 0);
9    long last = p.getLong("start", 0);
10   long cur = System.currentTimeMillis();
     // Service starts for the first time, store the start time
11   if (last == 0){p.putLong("start", cur); stopSelf();}
     //Service restarts, check if the interval exceeds 24 hours. Or
stops itself
12   else if(cur - last < 86400000) {stopSelf();}
     //Interval exceeds 24 hours, malicious behavior is triggered
13   else {//Leak Information Through HTTP Post;
14     String PI=updateInfo();//Collect private information
15     HttpPost httpRequest= new HttpPost(action);
16     httpRequest.setEntity(new UrlEncodeFormEntity(PI,"UTF-8");
17     new DefaultHttpClient().execute(httpRequest);
18   }
19 }// OnCreate ends
```

**Figure 2: Code example of DroidKungFu**

As shown in the example, the principle of time bombs is rather simple, because it only introduces a time delay before actually executing some malicious actions. In such situations, security analysts may not know what is a good time to stop the dynamic analysis, and neither can they easily speed up analysis [29]. Currently many malware implement anti-analysis in a similar way. To defeat this kind of anti-analysis, analysts knowing the time delay can reset the time clock of the dynamic analysis environment to a certain time or date. However, it is a non-trivial undertaking for analysts to extract this kind of anti-analysis automatically from malware, especially when the time related values are stored in files or received from network. Additionally, various other kinds of anti-analysis techniques, such as location bombs [16], emulator detection [27, 34], and remote command (from C&C server), challenge dynamic analysis as well. How to defeat such anti-analysis techniques automatically requires more research.

On one hand, current test input generation methods are not aware of such anti-analysis. Although many test input generation techniques have been proposed, such as CONTEST [2], Dyndroid [19], Condroid [31], IntelliDroid [38], they are mainly for the purpose of improving code coverage. They only focus on the input-dependent constraints and generate test inputs through a constraint solver. However, the above anti-analysis is mostly input independent, as shown in the two examples in Figure 1 and Figure 2. Therefore, the existence of these anti-analysis techniques is unknown to these test input generation techniques. Although some methods [38] can discover some time based anti-analysis techniques by modeling the related instructions, they can discover few possible anti-analysis techniques.

On the other hand, current anti-analysis solutions still have many limitations. For example, researchers have proposed to defeat emulator detection by building an emulator emulating all respects of a real device. To some extent, this is impossible. For instance, emulators will always expose timing and cache behavior that is distinguishable from real phones [30]. Other recent work [15, 29] proposed to defeat anti-analysis by adopting program slicing and hooking. They all need a configuration file for currently known anti-analysis. However, it is challenging to generate such files for all possible anti-analysis techniques. Moreover, they will also fail in the battle against unknown anti-analysis techniques.

Our method, Droid-AntiRM, tries to defeat various anti-analysis techniques automatically based on the three key observations mentioned in Section 1. It discovers anti-analysis by extracting the condition statements which control the execution of malicious behaviors and then checking whether they have dependence on program inputs. Droid-AntiRM tames the detected anti-analysis through bytecode instrumentation, and it does not need any manipulation or modification of the dynamic analysis environment. Droid-AntiRM is able to discover and defeat new anti-analysis techniques because it does not require any predefined configuration file to list all known anti-analysis techniques.

## 2.2 Definitions

For clarity, we formally introduce several concepts for our context.

- **Program Inputs**. Program input $I$ is an input parameter of initial program entry points, including various UIs, system-events, and some lifecycle entry points. The inputs derived from system calls, including those related to file system and network, are not considered as program inputs.
- **Target Methods**. Target method $t$ is a method analysts care about. Currently, we use a list of potentially sensitive or malicious APIs as target methods, which are further specified in Section 4.1.
- **Call Path**. A call path to a target method $t$ is a sequence of method invocations.
- **Control Dependency**. For a condition statement $cs$ and a target method $t$ (sensitive or malicious), $cs$ has control dependency on $t$ if $cs$ controls the invocation of $t$ directly or controls the invocation of other methods along the call path, which in turn invoke $t$ indirectly.
- **Input Dependence**. For a condition statement $cs$ and a program input $I$, $cs$ has input dependence on $I$ if the variables used in $cs$ have either direct or indirect data dependence on $I$.
- **Anti-Analysis**. In the context of our work, we consider a condition statement $cs$ as *anti-analysis* if $cs$ has control dependency on a target method $t$ and $cs$ has no input dependence on program inputs $I$.

## 3 SYSTEM OVERVIEW

Figure 3 depicts the overall workflow of our approach *Droid-AntiRM*. The initial inputs are APK files and a configuration file listing the target methods. At a high level, Droid-AntiRM comprises of three stages.

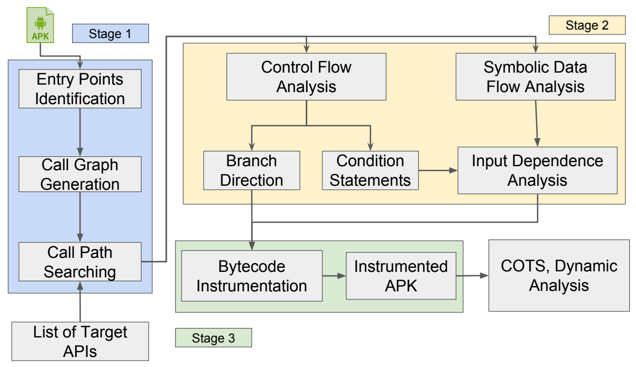

**Figure 3: System Overview**

In stage 1, Android APKs are first unpacked and entry points identification is performed by parsing various Android-specific files. Based on the identified entry points, call graph and control flow graph are generated through code traversal. With a list of target APIs, code traversal is carried out on the call graph to search for the call paths from identified entry points to target APIs. To illustrate the process, here we use the code example in Figure 4 , whose corresponding control flow graph is shown in Figure 5. First, the entry point *OnReceive* (at line 3) and the target APIs *abortBroadcast* (at line 22) and *sendTextMessage* (at line 24) are identified. Then the call paths to *abortBroadcast* and *sendTextMessage* are extracted, namely onReceive()⟶handle()⟶abortBroadcast and onReceive()⟶handle() ⟶sendTextMessage, respectively.

In stage 2, each extracted call path is taken as input to identify potential anti-analysis along it. Specifically, based on control flow analysis, Droid-AntiRM first extracts from the call path the condition statements which control the execution of target methods directly or indirectly. Such condition statements are considered as *potential anti-analysis*. Then symbolic data flow analysis is performed to generate the symbolic expressions of variables used in these condition statements. Finally, for each extracted potential

```
1  public class SystemEventReceiver extends BroadcastReceiver{
2  String action ="";
3  public void onReceive(Context context, Intent intent) {
4   if (intent.getAction().equals
          ("android.intent.action.BOOT_COMPLETED")){
5      action="BOOT_COMPLETED";}
6   else if (intent.getAction().equals
          ("android.intent.action.SMS_RECEIVED"){
7      action="SMS_RECEIVED";}
   // Invoke the handle method and end of OnReceive()
8   handle(intent);}
9  public void handle(Intent intent){
10  if(action=="BOOT_COMPLETED"){
11    Intent intent1 = new Intent(context, SystemService.class);
12    intent1.putExtra("from", "BOOT_COMPLETED");
13    context.startService(intent1);} //start to do malicious things
14  else { //SMS_RECEIVED
15    if(isVirtualDevices()) { System.exit(0); return;}//detect VM
16    else{ Bundle b=intent.getExtras();
17      Object[] pdus=(Object[])b.get("pdus");
18      for (int x=0;x<pdus.length;x++){
19        SmsMessage msg=SmsMessage.createFromPdu(pdus[x]);
20        String body=msg.getMessageBody();
21        String addr=msg.getOriginatingAddress();
22        if(addr.equals("95588")) {abortBrodcast();}
23          SmsManager sm=SmsManager.getDefault();
24          sm.sendTextMessage("123456", null, body,null, null);}}}
25 } //handle ends
26 public final boolean isVirtualDevices(){
27  model=Build.MODEL; device=Build.DEVICE;
28  if(!model.equals("sdk") && !device.equals("generic")){
29    return false; }
30  else { return true;} }  //isVirtualDevices ends
31 }//SystemEventReceiver ends
```

**Figure 4: Code example**

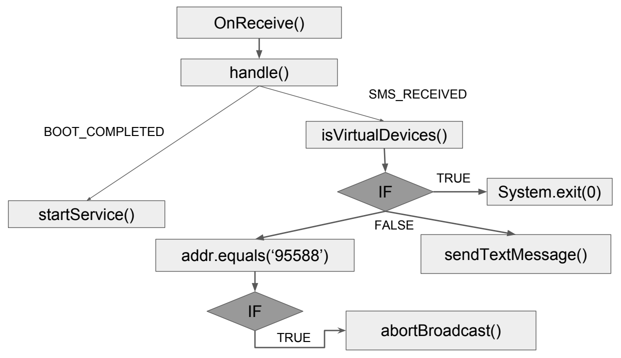

**Figure 5: Control Flow Graph of Code Example in Figure 4**

anti-analysis (i.e., condition statement), aided by symbolic expressions, we employ *input dependence analysis* to determine whether this potential anti-analysis is true anti-analysis or not.

In stage 3, for each detected anti-analysis condition statement, Droid-AntiRM alters its conditions through bytecode instrumentation. The condition is replaced by a simple Boolean variable, forcing the program later to take the path to the target method during dynamic analysis.

## 4 DETAILS OF DROID-ANTIRM

In this section, we will provide more details about the main components of Droid-AntiRM, namely, specifying target APIs, symbolic data flow analysis, input dependence analysis, and bytecode instrumentation.

### 4.1 Specifying Targeted APIs

Similar to IntelliDroid [38] and Condroid [31], our system also uses a list of potentially-sensitive APIs as target methods. These target methods are extracted from the results of PScout [6] and SuSi [28], which can be expanded easily.

Particularly, in this paper, we mainly focus on seven kinds of malicious behaviors and use their corresponding APIs as target methods, as shown below:

① **Premium SMS**: Send message to a premium number.
② **Blocking SMS**: Block a received SMS message.
③ **Deleting SMS**: Delete the stored SMS messages.
④ **Leaking information via SMS**: Leak private information through SMS.
⑤ **Network access**: Connect remote server through various HTTP API methods.
⑥ **Reflection and dynamic code loading**: Hide API through reflection or dynamic code loading.
⑦ **Root**: Get the root permission.

Because IntelliDroid [38] also uses these target methods, we can make a direct comparison with it in our evaluation.

### 4.2 Symbolic Data Flow Analysis

In addition to applying control flow analysis, we also implement static forward *symbolic data flow analysis*, which propagates the variable information intra- and inter-procedurally, and track the data flow of program inputs. Unlike symbolic execution which aims to improve high code coverage of dynamic analysis, symbolic data flow analysis is mainly used to generate and propagate symbolic expression of variables for data flow tracking and it does not involve a constraint solver. For each call path to a certain target method, the process of symbolic data flow analysis mainly includes two parts: *symbolic input data initialization* and *symbolic input data propagation.*

*4.2.1 Symbolic Input Data Initialization.* Generally speaking, each type of program input will be processed by the corresponding handler(s) and transformed into specific input parameters, which are the input data source. For example, consider again the code example in Figure 4, SMS event or boot completed system event inputs are initially represented as *intent* (line 3), which is the second parameter of the *OnReceive* method. By symbolic input data initialization, we initialize each input parameter as a symbolic expression: *(Inputdata i)*, where *i* is the index of the parameter, starting from 1. In this example, the two parameters of *onReceive*(), that is, *context* and *intent*, are initialized as *(Inputdata 1)* and *(Inputdata 2)*, respectively. In the following data flow analysis, this symbolic expression will be considered as initial program input data. Various system event input data are initialized in this way.

Another input data source needs to treated specially, which is user input data generated through user's interactions with an app. For example, a user can input data into an app through an *EditText* control which is located through a system call *findViewById(int)*. In our system, we also consider this type of data extracted through invoking *findViewById(int)* as program input data and initialize it as a single symbolic expression *(InputdataFromUI)*.

#### 4.2.2 *Symbolic Input Data Propagation.*

Symbolic input data propagation is implemented both locally within methods and across different methods. To illustrate this process, consider again the code example in Figure 4. The call path from *OnReceive* to *abortBroadcast* consists of two method invocations, namely OnReceive and handle. When the *handle* method is invoked, an input parameter *(intent)* is needed, which has been represented as symbolic expression '*(Inputdata 2)*' in our initialization process. Thus, the *handle* method will use '*(Inputdata 2)*' as its initial input data. In this way, symbolic input data can be propagated across different methods. Inside the *handle* method, the symbolic input data is processed through a sequence of SMS related operations. By modeling these operations, all related data flow information can be tracked using symbolic expression. For example, the string variable *addr* can be represented as symbolic expression '*(Inputdata 2).getExtras().get("pdus").createFromPdu().getOriginatingAdd ress()*'. In this way, symbolic input data can be propagated within each method. In addition to Android specific instructions, we also model other types of instructions (e.g., string related instructions) to precisely implement this propagation process. We will discuss this in detail in Section 5.3.

When symbolic input data propagates across different methods, two cases are treated specially in our current prototype, namely Inter Component Communication (ICC) and Inter Process Communication (IPC). Unlike parameter passing in simple method invocations, in ICC an *intent* data structure is used to pass information between components. As such, symbolic input data may be placed in some fields of an *intent* and is transmitted to another component. In this case, if we cannot correctly propagate the symbolic input data across different components, the variables in another component which depend on input data would be considered as input independent. This may cause false positives. To handle this case, when a method invocation is identified as ICC, a *key-value* map is constructed to store and pass the field information of the intent used in this ICC. By modeling the intent related instructions (e.g., `putExtra`), we can track which values are put in which fields and store them in this key-value map. For example as shown in Figure 4, before starting a service component, the string value "BOOT_COMPLETED" is attached to the "from" field of the *intent* through the `putExtra` method (lines 12) and the constructed key-value map will contain <"from","BOOT_COMPLETED">. Then, during our analysis, the intent and the constructed key-value map are both passed to the resolved method of the target component. In the resolved method, the propagated data can be retrieved using the received intent and key-value map.

In the IPC case, the solution is similar. For example, the *bundle* data structure is often used to transmit information between an invoking method 'Handler.sendMessage' and a resolved method 'Handler.handleMessage'. Same to *intent* in ICC, a key-value map is constructed for each *bundle* object to propagate symbolic input data across different processes.

Note that in both the ICC and IPC cases, the key-value map can be constructed and propagated only when both the stored fields and values are not obfuscated.

## 4.3 Input Dependence Analysis

To detect true anti-analysis, we need to determine whether a *potential anti-analysis* condition statement has dependence on program inputs. When it does not have dependence on inputs, it is considered as true anti-analysis. For the example in Figure 4, the execution of target method *abortBroadcast* is controlled by three condition statements at line 10, line 15 and line 22, and thus these three condition statements will be extracted as *potential anti-analysis*. Resolving the input dependence of a condition statement means to perform the input dependence analysis of its variables. In the simplest case, the variables used in a condition statement originate from the input parameters of the method. For example, the *addr* variable in line 22 originates from *intent*, the input parameter of the *handle* method. Because *addr* has been represented as symbolic expression '*(Inputdata 2).getExtras().get(pdus).createFromPdu().getOrigin atingAddress*', we can directly determine that it depends on the program inputs. We call this kind of input dependence direct data dependence.

On the other hand, we cannot simply claim a variable input *independent* when it has no direct data dependence on program inputs. This is because there may exist other kinds of variables which indirectly depend on inputs, such as *heap variable* and *return variable*. In this paper, we define a variable which can be used or set globally in another method as a *heap variable*, and a variable whose value is returned from another self-defined method as a *return variable*. Next, we discuss how to solve these complex cases in detail, and finally give a formal summary for anti-analysis identification.

#### 4.3.1 *Heap Variable.*

As shown above, for the target API *abortBroadcast*, the condition statement at line 10 is extracted. The variables used in this condition statement are string variable *action* and constant string. Apparently, both variables are input independent. However, *action* is a heap variable and it can be used or set globally by another method. Thus, we need to further track the possible definition or modification of this heap variable. In this example, we find *action* is defined globally but is assigned a new value in method *OnReceive*. Another round of symbolic data-flow analysis and control-flow analysis is performed along the path (called *auxiliary* path) from the entry point to each value assignment instruction over the heap variable. Analysis result shows that the value assignment is controlled by two condition statements, code at line 4 and 6. In addition to two constant strings, these two condition statements both depend on inputs. In other words, although heap variable *action* does not have direct data dependence on inputs, its value is indirectly dependent by program inputs, which we call *indirect data dependence*. In this paper, we also consider variables that have indirect data dependence on inputs as input dependent.

More formally, for each heap variable $h$ used in a condition statement, we extract its symbolic value by function *heapvalue(h)* to resolve the direct data dependence on inputs. We also extract its symbolic constraints by function *heapconstraints(h)* to resolve the indirect data dependence on inputs. Certainly, if there are other heap variables in the *auxiliary* path, the above analyses can be repeated.

#### 4.3.2 *Return Variable.*

In some cases, the variables used in condition statements are return values from other method invocations, such as the extracted condition statement at code line 15. The variable used by this condition statement is a return value from method *isVirtualDevices*, represented by symbolic expression *(isVirtualDevicesReturn)*. To handle such cases, the return values and return sites of the method need to be extracted. In this example, method *isVirtualDevices* can return two values, namely true and false. Because these return values are constant boolean values, they have no direct data dependence on inputs. However, there may also have indirect data dependence, similar to the above heap variable case. To extract the possible indirect data dependence, control flow analysis is implemented from the method entry point to these return sites (at code line 29,30) and the condition statement at code line 28 is extracted. The extracted condition statement includes two constant strings and two local variables. Analysis results show that none of these four variables have direct data dependence or indirect data dependence on inputs. As the return value in the condition statement at code line 15 has no dependence on inputs, this condition statement is input independent.

More formally, for each return variable $r$ used in a condition statement, we extract all its return values by function *returnvalue(r)* to resolve the direct data dependence on inputs. The constraints for each return value are extracted by function *returnconstraints(r)* to resolve the indirect data dependence on inputs. In the case of multiple return values and constraints, they are combined using a logical *OR* to generate the final symbolic expression. Similar to heap variable analysis, if there are other return values used in the return constraint, such analyses can be repeated.

*4.3.3 Summary of Anti-Analysis Identification.* For each extracted *potential anti-analysis* condition statement *cs*, it is identified as true *anti-analysis* (i.e., input independent) if the following properties all holds:

(1) For any variable $v$ in *cs*, the symbolic expression of $v$ does not contain 'Input'.
(2) For any variable $v$ in *cs*, *heapvalue(v)* does not contain 'Input' and *heapconstraints(v)* does not contain 'Input' when $v$ is a heap variable.
(3) For any variable $v$ in *cs*, *returnvalue(v)* does not contain 'Input' and *returnconstraints(v)* does not contain 'Input' when $v$ is a return variable.

A special case is that when data flow information of variables in a *potential anti-analysis* condition statement is lost, conservatively, we also consider this condition statement as anti-analysis. In this way, we can avoid possible false negatives although some false positives may be introduced. We will discuss this in our evaluation.

## 4.4 Bytecode Instrumentation

Through the above steps, the *anti-analysis* condition statements can be extracted. Furthermore, the appropriate branch directions for these condition statements are also determined by previous control flow analysis. For example, at code line 15 in Figure 4, if this condition statement is true, then the execution will shut down and the target method will not be triggered. Only if this condition statement is false, the target APIs can be triggered. To tame this *anti-analysis*, in this paper, we replace this condition statement as a simple Boolean value, namely *true* or *false*. In this way, during the following dynamic code analysis, the program can be forced to take the correct branch direction, which will trigger the target method(s). During this process, two cases need to be treated specially.

The first special case is *switch condition statement*. Compared to general 'if-else' condition statements, a switch condition statement may have many cases to fall through, so simply replacing it as a Boolean value is not feasible. Some malware (e.g., Geinimi) also adopt this kind of condition statements to control their behaviors. To solve this case, we conservatively transform these switch condition statements into a sequence of 'if-else' condition statements and then perform the control flow analysis. In our current prototype, for these samples with switch condition statements, bytecode instrumentation will be implemented twice. The first one is to transform switch condition statements into general 'if-else' condition statements and the second one is to replace the resultant condition statements.

The second special case is when the condition statements control a loop, such as *for* loop (at line 18 of Figure 4) or *while* loop. To illustrate this special case, we extract the code from lines 18-24 in Figure 4, which contains a *for* loop, and show its bytecode representation in Figure 6. Here the *for* loop is represented by bytecode lines 4,6,10,11 (this is also the way to express other types of loops). If the condition statement (at bytecode line 4) that controls this *for* loop is extracted and instrumented, unexpected results will be generated. For example, when choosing *false* for this condition statement during bytecode instrumentation, the malicious behaviors following it will not be triggered. Worse, when choosing *true* for this statement, the code will loop infinitely during runtime analysis. Thus, in this paper, we make a special treatment for these condition statements in loops. Even if these condition statements have no dependence on program inputs, we will *not* consider them as *anti-analysis* and hence no bytecode instrumentation will be performed on them.

```
1  $r10 =<android.os.Bundle: java.lang.Object
            get(java.lang.String)>("pdus");
2  $i0 = 0;
3  $i1 = lengthof $r10;
4  if $i0 < $i1 goto $r12 = $r10[$i0];
5  Return;
6  $r12 = $r10[$i0];
7  $r3 = <android.telephony.gsm.SmsMessage
            createFromPdu(byte[])>($r12)
8  $r14 =$r3.<android.telephony.gsm.SmsMessage: getMessageBody()>()
9  ......; abortBroadcast();
10 $i0 = $i0 + 1;
11 goto [?= $i1 = lengthof $r10]
```

**Figure 6: Bytecode representation of Code Example in Figure 4 (code lines 18-24)**

## 5 IMPLEMENTATION DETAILS

In this section, we provide some implementation details of Droid-AntiRM, including the tools built upon and some key components. The whole system is written in Java, consisting of 9,760 LOC.

We adopt the tools called *Dare* and *Apktool* to unpack the Android malware and convert the Dex code into Java bytecode files. The bytecode files are then converted into static single assignment format (SSA) by a static component of WALA [8]. Droid-AntiRM relies on *WALA* to build a call graph with a type-based heap model. It follows the approach described in FlowDroid [5] to model Android

Table 1: Anti-analysis Statistics in Real-world Malware

| #Time | #Location | #Emulator Detection |
|---|---|---|
| 856 | 14 | 166 |

application component lifecycle and search for entry points. To handle the pervasive use of IPC, the results in EdgeMiner [7] are integrated to model the control flow transfers through the Android framework and the ideas in Epicc [24] are also reused to precisely model ICC. As the underlying analysis engine for Droid-AntiRM, WALA is also used for providing other various functionalities, such as control flow graph generation, call path searching and points-to analysis.

To ensure that program input data information will not be lost, we must model various types of general instructions, such as string operations, and other Android specific instructions. In our current prototype, we mainly model 9 kinds of general instructions, such as *getfield* instructions, *arrayload* instructions, *binary operation* instruction, and *general invoke* instruction, etc. Additionally, we model the time, location, file, network related values and operations to track the external data that is not generated from program inputs. In practice, we model about 200 instructions from 13 related classes, such as java.util.Date, java.net.URLConnection, android.location.location, etc. Using the data generated by these instructions, we can discover new possible anti-analysis techniques.

Currently, we adopt *Soot* to implement bytecode instrumentation, as done in[4]. Soot transforms APK into an intermediate representation called *Jimple*. The bytecode instrumentation is implemented on the *Jimple*. It is worth mentioning that the overall bytecode instrumentation process comprises of three steps: intermediate representation transformation, call graph and control flow graph generation, bytecode instrumentation. While both this stage (stage 3) and stage 1 of our system (Figure 3) generate call graph and control flow graph, we generate them twice because different tools generate different representations, which prevents us from reusing them. In the following evaluation, the bytecode instrumentation time is the sum of the time for each of these three steps.

## 6 EVALUATION

In this section, we first study the usage of anti-analysis techniques in real-world Android malware, and then evaluate the effectiveness and efficiency of Droid-AntiRM to tame the anti-analysis techniques. Our prototype system is implemented and tested on an Intel i5-4460 CPU at 3.1G Hz host with 2GB of heap memory of the JVM.

**Dataset**. The Android malware dataset we use in our work are collected from VirusTotal [33], Drebin [3], Contagio [25] and Malware Genome [44]. In total, there are 3187 samples.

### 6.1 Usage of Anti-analysis Techniques

We first apply Droid-AntiRM to analyze the use of anti-analysis in the entire dataset. During our analysis, 17 samples failed due to memory error. By analyzing the results from the remaining successful cases, we make a simple statistics on *time*, *location*, *emulator detection* based anti-analysis techniques, as shown in Table 1. Of the total 3187 malware samples, 856 are found using time based anti-analysis, 14 using location based anti-analysis, and 166 using emulator detection based anti-analysis. In total, 1036 malware samples (32.50%) exhibit some anti-analysis behavior. While our analysis is based on a (small) portion of Android malware samples, the result does indicate that anti-analysis techniques have been adopted by many malware writers; hence, it is important to defeat these anti-analysis techniques to assist dynamic code analysis.

To discuss the effectiveness and efficiency of our proposed method in detail, next we aim to answer the following questions:

- **Question 1**: How accurate is our method in detecting anti-analysis in malware?
- **Question 2**: How effective is our method in improving current dynamic analysis by taming the detected anti-analysis (i.e., code instrumentation)?
- **Question 3**: How efficient is our method in detecting anti-analysis and instrumenting bytecode?
- **Question 4**: Can our method automatically detect anti-analysis techniques from recent malware?

### 6.2 Accuracy of Anti-analysis Detection

On the one hand, if some anti-analysis related condition statements are not extracted (i.e., false negatives), malicious behaviors can still be hidden, evading dynamic analysis. On the other hand, if in an app condition statements that are not related to anti-analysis are extracted and modified, the semantics of the app may be changed. Thus, we need to evaluate the accuracy of anti-analysis condition statements detection.

As far as we know, there is no well-known ground truth on the anti-analysis behavior of malware samples. As a best-effort solution, we choose to use some known samples which have been analyzed and documented in detail [9]. From the entire malware dataset (3187 samples), we randomly select at least one sample from each of the 23 malware families, and in total 30 samples are selected to perform in-depth manual validation. For each sample, Droid-AntiRM is used to extract the anti-analysis condition statements automatically. Then we validate these detected anti-analysis by manually examining all these samples' code after decompiling them [1]. To reveal false positives (i.e., anti-analysis cases reported by Droid-AntiRM are not real ones), we manually inspect the corresponding source code of detected anti-analysis. To reveal false negatives (i.e., anti-analysis cases which are not detected by Droid-AntiRM), we extract all the condition statements in each sample and manually inspect the source code of those undetected condition statements. As a result, Droid-AntiRM automatically extracts 258 anti-analysis instances from these selected samples. Our manual validation shows that Droid-AntiRM has detected 230 true positive instances and 28 false positives, and there is no false negative. The detailed results are shown in Table 2. Note that in our evaluation the commands we consider for command-based anti-analysis are from network rather than SMS. This is because SMS commands are dependent on program inputs, and hence anti-analysis based on SMS commands can be solved by many test input generation techniques, such as IntelliDroid [38].

**Case Studies**. For example, Droid-AntiRM automatically extracts 12 anti-analysis related condition statements from AnserverBot, including time based and command based anti-analysis. Aided by a

**Table 2: Anti-Analysis Extraction Results**

| Sample Name | #All | #TP | #FP | Type |
|---|---|---|---|---|
| ADRD | 6 | 4 | 2 | T |
| Anserverbot | 14 | 12 | 2 | T, C |
| Adware_1 | 4 | 4 | 0 | T, L |
| Adware_2 | 7 | 6 | 1 | T, L |
| BaseBridge | 17 | 13 | 2 2 | T, C |
| BgServ | 3 | 1 | 2 | T |
| DroidCoupon | 2 | 2 | 0 | E |
| DroidDream | 6 | 6 | 0 | T |
| DreamLight | 3 | 3 | 0 | T |
| DroidKungFu _1 | 13 | 11 | 2 | T,C |
| DroidKungFu_2 | 9 | 7 | 2 | T ,C |
| DroidKungFu_3 | 10 | 9 | 1 | T ,C |
| Exploit | 1 | 1 | 0 | T |
| FakeInstaller_1 | 17 | 16 | 1 | T, C |
| FakeInstaller_2 | 1 | 1 | 0 | T |
| Geinimi | 14 | 14 | 0 | T, C |
| GinMaster_1 | 11 | 11 | 0 | E, T |
| GinMaster_2 | 6 | 4 | 2 | E, T |
| GinMaster_3 | 3 | 2 | 1 | T |
| GoldDream | 11 | 9 | 2 | T, C |
| HeHe_1 | 20 | 18 | 2 | E, C, T |
| HeHe_2 | 16 | 14 | 2 | E, C, T |
| KMin | 10 | 8 | 2 | T, C |
| MobileSpy | 17 | 17 | 0 | E, C |
| Obad | 2 | 2 | 0 | T, E |
| Pincer | 6 | 6 | 0 | E |
| Pjapps | 9 | 9 | 0 | T, C |
| SmsSpy | 8 | 8 | 0 | T |
| Stealer | 10 | 10 | 0 | T,C |
| Zsone | 2 | 2 | 0 | T |
| Total | 258 | 230 | 28 | 89.15% |

**All**=Automatically Extracted Anti-Analysis; TP=Manually Identified Anti-Analysis; Number with green color indicates false positive caused by Reason 1; Number with red color indicates false positive caused by Reason 2. Type of Anti-Analysis: E=Emulator Detection, T=Time, C=Command, L=Location

time-based check, AnserverBot connects to the C&C server every 2 hours. In another example, a time-based check is extracted from the ADRD sample, which is used to check the time elapse since the last time the malware wrote in the *oldtime* field of the xml file. If the time is more than 6 hours, the malware collects some local information and sends it to a remote server, and then updates the *oldtime* field.

In the Obad sample, it detects emulator by checking the value of the Android *Build.MODEL*. If the value contains 'sdk', which is an indication of emulation, the malware will close some activities to avoid analysis. In some cases, such as Genimi, HeHe, Pjapps, etc, they will not perform certain malicious actions until receiving specific instructions from a command-and-control server (a specific 'time delay'). We can extract these command based anti-analysis because the remote commands are usually transmitted through network and have no dependence on program inputs. For example, nine anti-analysis related condition statements are extracted from the Pjapps sample, among which, five cases are based on remote commands.

**False Positives**. As shown in Table 2, the false positive rate is 10.85% (28/258). We manually analyze these false positive cases in detail and identify two root causes.

**Reason 1**. The first important reason is that we could not keep accurate track of the data and control flow information of program inputs as we expected. Although we have tried to model as many related instructions as possible, input data flow information may still be lost to cause false positives because of self-defined methods. Of the 28 false positive cases, 5 of them are due to this reason (as shown in green color). For example, in the KMin sample, a variable named *headers* is used in a condition statement of method *mmsHold*, which is responsible for leaking received messages. However, we cannot get the data flow information of this variable; thus, this condition statement is conservatively considered as anti-analysis. By manual inspection, we find that this variable is generated by a self-defined class *PduParser()* and method parseHeaders with a parameter *(Inputdata 2).getByteArrayExtra('data')*. In the method *parseHeaders*, a map is used to query and return each type of input string. Although Droid-AntiRM has kept tracking the data flow information of this parameter, it could not track the data dependence between the input parameter and the return value through the self-constructed map. Thus, the data flow information of the return value (namely variable *headers*) is lost, causing a false positive. This kind of false positives also exist in the BaseBridge and FakeInstaller samples.

**Reason 2**. The second reason is that some apps rely on certain input independent condition statements to ensure their proper execution. For example, some network related operations are used in condition statements to ensure that network access is available. The remaining 23 false positive cases are all due to this reason (as shown in red color). For example, in DroidKungFu, *NetworkInfo.isConnected()* is used in a condition statement to check whether there is a network connection before connecting to a C& C server. If not, it will not contact the server. In samples ADRD, Bgserv, GoldDream, some network related checks are also found, such as, *getActiveNetworkInfo*, etc. Although these condition statements have no dependence on program inputs, they are needed to ensure the normal running of both malware and benign apps. Therefore, we may white-list such generic APIs to avoid labeling network connection checking based condition statements as anti-analysis, hence removing such type of false positives. After white-listing these APIs in our experiments, we can reduce the overall false positive rate from 10.85% to 1.94% for the selected 30 samples. Indeed, by white-listing more network connectivity checking APIs, we can also avoid this type of errors in other malware samples. As a part of our future work, we will analyze more samples to expand the list of such APIs.

While our evaluation does not definitely exclude the possibility of false negatives, our evaluation results show an encouraging step towards automatic detection of anti-analysis techniques in Android malware. In addition to extracting anti-analysis related condition statements automatically, Droid-AntiRM also returns the detailed symbolic variable information used in these anti-analysis techniques, which can help decide whether a reported anti-analysis case is true or not. In practice, for each anti-analysis detected by Droid-AntiRM, on average we were able to justify it by reading less than 50 lines of related source code, hence greatly saving the analyst's time.

**Table 3: Comparison Results**

| Event | SMS | Intent | UI | Life |
|---|---|---|---|---|
| ADRD | ② | | ⑤ | ⑤ |
| Anserverbot | ②⑤ | | ⑤⑥ | ⑥ |
| Adware_1 | ② | | | ⑤ |
| Adware_2 | ② | | | ⑤ |
| BaseBridge | ①②⑤ | | | ⑥ |
| Bgserv | ①②⑤ | | | |
| DroidCoupon | ⑤ | | ⑤ | ⑤ |
| DroidDream | ⑤ | | ⑤ | |
| DreamLight | ⑤ | | ⑤⑥ | |
| DroidKungFu_1 | | | ⑤ | ⑤⑦ |
| DroidKungFu_2 | | | ⑤ | ⑤⑦ |
| DroidKungFu_3 | | | | ⑤ ⑦ |
| Exploit | ⑤ | ⑤ | ⑤⑥ | ⑥ |
| FakeInstaller_1 | ⑤⑥ | ⑤ | ⑤ | |
| FakeInstaller_2 | ⑤ | | ⑤⑥ | ⑤⑥ |
| Geinimi | | ③⑤ | | ③⑤ |
| GinMaster_1 | | | ⑤ | ⑤ |
| GinMaster_2 | | | ⑤ | ⑤ |
| GinMaster_3 | | | ⑤ | ⑤ |
| GoldDream | ① | | | ⑤ |
| HeHe_1 | ②⑤ | | | ⑤ |
| HeHe_2 | ②⑤ | | | ⑤ |
| KMin | ②⑤ | | ③⑤ | |
| MobileSpy | ②⑤ | | | ①⑤ |
| Obad | ② | | | ⑤ |
| Pincer | ②④ | | ⑤ | ⑥ |
| Pjapps | ②④⑤ | | | ⑤ |
| SmsSpy | ② | ④ | | ⑤ |
| Stealer | ② | | ⑤ | ⑤ |
| Zsone | ① ②⑤ | | ① | |

Row indicates the malware family and columns indicate the type of input injected. ⓘ indicates the type of malicious API triggered by IntelliDroid, where $i$ corresponds to the API index listed in Section 4.1. ⓘ(red color) indicates the type of malicious behaviors triggered after integrating with Droid-AntiRM.

## 6.3 Effectiveness

To demonstrate whether Droid-AntiRM can improve the effectiveness of current dynamic analysis techniques against Android malware, we choose IntelliDroid [38], a recently proposed dynamic analysis method with targeted input generation. For each target API (listed in Section 4.1) existing in an APK file, IntelliDroid can generate targeted test inputs to trigger it.

In our case study, we compare the trigger results of IntelliDroid on the original APK file and also on the instrumented-and-repackaged APK created by Droid-AntiRM. For fair comparison, here we use the same generated test inputs and target APIs for both the original APK and repackaged APK, and we test them twice on the same dynamic analysis platform using IntelliDroid. The dynamic analysis platform is a Nexus 7 emulator targeting Android-4.3.1. In this way, we can demonstrate the effectiveness of Droid-AntiRM by comparing the numbers of successfully triggered target APIs in the original APK and the new APK.

The comparison results are shown in Table 3. It shows that IntelliDroid can successfully trigger 51 target APIs without Droid-AntiRM. After integrating with Droid-AntiRM, IntelliDroid can trigger additional 44 target APIs. These additional target APIs are hidden by various anti-analysis techniques. Next we discuss some details.

### 6.3.1 *Taming specific time interval based anti-analysis.*

First, we show how our method can tame the time-based anti-analysis techniques. For example, consider the Anserverbot sample again. An *openconnection()* network behavior in method *com.sec. *.Onion.d()* is triggered to connect a C&C server when receiving a SMS. To trigger this behavior, IntelliDroid [38] extracts the path constraints through the call path to this network behavior, and then a specific test input SMS is generated. However, this behavior is not triggered successfully when the generated SMS is injected. Aided by Droid-AntiRM, we extract a condition statement along this call path, which implements a time-interval check. This condition statement is extracted from method *com.sec.*.Onion.c()* and is used to control the invocation of method *com.sec.*.Onion.d()*. Since this condition statement is implemented by using SharedPreference and it does not have dependence on program inputs, IntelliDroid [38] fails to trigger the behavior by simply injecting an SMS. Through extracting this condition statement and removing it by Droid-AntiRM, this behavior can now be triggered successfully by IntelliDroid. Many similar cases have been found in samples like DroidKungFu, Zsone, HeHe, etc, which can be resolved by Droid-AntiRM.

### 6.3.2 *Taming emulator detection based anti-analysis.*

To defend against dynamic analysis, various emulator detection techniques are also commonly used to hide malicious behaviors. In these cases, only test inputs are not sufficient to trigger hidden behavior. For instance, the Droidcoupon malware will start a service to perform network access behavior when receiving an SMS. However, a method named *isVirtualDevices()* is used to implement emulator detection and hide the above behavior. Merely injecting a specific SMS by IntelliDroid would not trigger this behavior successfully. Aided by Droid-AntiRM, we can extract this condition statement and tame it. Then the corresponding behavior can be triggered successfully when injecting the generated SMS by IntelliDroid again, as shown in Table 3. Similar cases are also found in malware Zsone, Pincer, etc. For these malware, in addition to the generated test inputs, it is essential that emulator detection should be tamed so that some behaviors can be triggered successfully.

### 6.3.3 *Taming remote command based anti-analysis.*

Many recent malware only acts when receiving specific commands from a C&C server. To analyze this behavior, analysts must set up a fake server to inject these specific commands to trigger the malicious behaviors during dynamic analysis, as does in IntelliDroid [38]. Pjapps malware starts a background service *MainService* when receiving any broadcast. Then, *MainService* is responsible for executing various tasks when receiving different specific commands. For example, when the 'note' command is received, this service will send an SMS. Although IntelliDroid [38] can start *Mainservice* successfully by injecting generated SMS, remoted commands have to be injected manually. However, this manual work is not always successful when these commands are encrypted. In this example, because these command check condition statements are dependent on network input and have no dependence on program inputs, Droid-AntiRM can tame them automatically, finally enabling IntelliDroid to trigger the hidden behaviors.

## 6.4 Efficiency

To evaluate the performance of Droid-AntiRM, we make the evaluation on the entire dataset (3187 samples) and record their analysis time. While samples need to be preprocessed before our analysis using tools Apktool and Dare, in this evaluation, we do not consider these preprocessing time. Instead, we mainly measure the performance of two parts in Droid-AntiRM, namely anti-analysis condition statements extraction and bytecode instrumentation. On average, the anti-analysis extraction for each app requires 13.3 seconds and 90% of these apps can be analyzed within 18.6 seconds.In particular, during this process, building call graph and loading class are the main sources of time consumption, averaging about 70% and 25% respectively. In contrast, the time cost of symbolic data flow analysis and input dependence analysis only accounts for less than 5%. On the other hand, as described in Section 5.4, different tools are involved in the three-step bytecode instrumentation process. The average time for instrumenting each app is 11.8 seconds, and 90% of these apps can be instrumented within 25.7 seconds. The experiment results show that Droid-AntiRM has good performance to scale its analysis to thousands of real-world Android malware.

## 6.5 Automatically Discovering Anti-analysis Behavior

We next report some interesting anti-analysis cases that Droid-AntiRM discovered from recent malware. Although some of these results have already been found earlier through manual investigation of security experts, to the best of our knowledge, our method is the first to discover these anti-analysis techniques automatically.

First, we find some interesting time interval related anti-analysis techniques. Unlike the time interval control in DroidKungFu, a more sophisticated way is found in sample Bgserv. Two values stored in SharedPreferences are used to construct this control, and their keys are BLOCK_START and BLOCK. Once an SMS is received, it is possible to block it unless when the time interval between current time and BLOCK_START is less than BLOCK. Another time based anti-analysis case is found in a recent sample named CallAPK, which also uses two SharedPreferences values: *usr_inactive* and *last_active*. Only when the sum of *usr_inactive* value and *last_active* value is greater than the current time value, a malicious service will be started.

Second, Droid-AntiRM can automatically extract those known emulator detection techniques from samples like Pincer, BaseBridge, etc. For example, Pincer malware checks whether it is running on an emulator by checking certain parameters like the IMEI, model name, phone number etc. In addition to these known techniques, some recent malware start to move the emulator detection functionality to the remote server. As stated earlier, NickSpy detects emulator by sending the IMEI value of the device to a remote server and waiting for its response. This kind of emulator detection can easily defeat current proposed method, such as Harvester [29].

## 7 LIMITATIONS

Next we discuss the limitations of our system, which malware authors could potentially exploit.

### 7.1 Call Graph Generation

A complete and accurate call graph is required for our analysis system to identify the target APIs and find the call paths. However, it is possible that our implementation does not model precisely the Android-specific components, such as ICC, asynchronous callbacks, etc. Our current prototype also cannot handle the usage of reflection, dynamic code loading, native code, or other various obfuscation techniques. That is, our generated call graph may also miss some edges, which will cause some call paths not to be found and cannot extract some anti-analysis along these call paths. We consider the complete and precise modeling of these aspects out of our scope and a subject for our future research.

### 7.2 Symbolic Data Flow Analysis

We rely on symbolic data flow analysis to track the data and control flow of program inputs. However, this is not an easy task. Our current prototype has two limitations. **1).** We have not modeled all the instructions which may transit data information. As such, our current prototype may miss some input dependent data information during symbolic data flow analysis, causing false positives. Nevertheless, it is conceptually easy to overcome this limitation by modeling more similar instructions. **2).** Our prototype cannot handle various obfuscation techniques, such as reflection and encryption, etc., which will impact the accuracy of symbolic data flow analysis. We acknowledge that our prototype needs to be improved to reduce the false positive rates, which we leave as our future work.

### 7.3 Attacking Input Dependence Analysis

If an attacker knows our approach, he can thwart input dependence analysis by deliberately transforming an input independent variable into input dependent through several operations, and hence cause false negatives. For example, if an input data string (e.g., SMS address) and another input independent string (e.g., device ID) are concatenated, and then a `substring` test is performed on the resulting string to generate the variable used in a condition statement. Since the generated variable appears to be input dependent, this condition statement will not be detected as anti-analysis. In this way, an attacker could probably construct a reasonably opaque predicate that is hard for existing test input generation techniques to solve and also includes some program inputs which can succeed in bypassing our system. To solve this problem, we can take a more conservative approach. That is, we can adjust the anti-analysis identification rules in Section 4.3.3 so that, if a condition statement contains a variable which has data dependence on both input data and non-input data, it will still be considered as anti-analysis. Note that in this way this attack will not bypass our detection. The trade-off is that an analyst may encounter some false positive cases for analysis.

## 8 RELATED WORK

Researchers have proposed various methods for analyzing the malicious behavior of malware. Below we discuss the most relevant works in these areas.

### 8.1 Static Analysis

FlowDroid [5] and Droidsafe [13] are well known static taint analysis tools for detecting privacy leaks in Android apps. Additionally, various static methods [7, 18, 23, 24, 35, 37] have been proposed to tackle the problem of inter-component data flow tracking. We reuse some ideas from the previous work. Naturally, our method shares the same limitations with these static methods. Another static analysis method most related to ours is TriggerScope [11]. There are three main differences between our work and TriggerScope. First, our goal is to improve dynamic analysis by detecting and taming anti-analysis techniques, while TriggerScope's goal is to detect malware through logic bomb detection. Second, our method focuses on input independent condition statements detection, while TriggerScope's method focuses on detecting some narrow condition statements, not on input dependence. As a result, TriggerScope may detect certain logic bombs which Droid-AntiRM may not. One special case is SMS-based logic bombs, which are mostly input-dependent (through passing intents). Third, TriggerScope currently focuses on SMS, time and location related logic bombs. Differently, Droid-AntiRM focuses on all possible anti-analysis techniques, including time, location, command, emulator detection, etc. Therefore, some anti-analysis techniques detected by Droid-AntiRM may not be detected by TriggerScope, such as command or emulator detection based cases.

### 8.2 Dynamic Analysis

To avoid the inherent shortcomings of static analysis methods, various dynamic analysis approaches have been proposed, including TaintDroid [10], CopperDroid [32], DroidScope [40], etc. Although dynamic analysis can overcome the shortcomings of static analysis in theory, two challenges need to be solved. The first big challenge is how to generate test inputs effectively to trigger the interesting behaviors. To solve this, many test input generation approaches [2, 12, 19, 41] have been proposed to complement these dynamic analysis tools. Another challenge is that many anti-analysis techniques have been used to complicate dynamic analysis tools, scuh as detecting the existence of the analytic components or the emulated execution environments. To solve this, many bare-metal dynamic analysis techniques are proposed, such as BareDroid [21], Malton [39], Ninja [22], etc. However, state-of-art solutions on mobile platforms can only restore the disk, and require a time-consuming system reboot. Recently, a transparent restoration mechanism without rebooting is proposed [14] to support bare-metal analysis on mobile platforms, which is particularly valuable for building a scalable bare-metal analysis platform with high throughput.

In addition to the above methods proposed for Android applications, researchers have also proposed many approaches [20, 26, 42] to analyze the PC malware. Since their main goal is to improve code coverage, they need to explore multiple paths and perform the forced-path execution of suspicious behaviors. Although they can effectively improve the code coverage and discover more suspicious applications behaviors, they still waste many computing cycles for analyzing the program paths that are irrelevant to suspicious behaviors. Compared to PC applications, Android applications usually have more program entry points and possibly more program paths, which may cause these methods ineffective and inefficient to discover the suspicious behaviors. In contrast, Droid-AntiRM only focuses on the program paths whose targets are suspicious behaviors, and therefore efficient for anti-analysis detection.

### 8.3 Hybrid Analysis

To overcome the limitations of both static and dynamic analysis techniques, a few hybrid static/dynamic analysis approaches have been proposed, such as IntelliDroid [38], Harvester [29], etc. Harvester also can remove anti-analysis techniques through program slicing. To determine whether a condition check is anti-analysis, a predefined configuration file listing all know anti-analysis techniques is needed for Harvester, which in turn can generate false negatives. Compared to Harvester, our method does not need this predefined configuration file and eliminate false negatives in theory. IntelliDroid is the most recent hybrid approach which can generate test inputs effectively for dynamic analysis. The main difference between IntelliDroid and our method lies on the different design goal. IntelliDroid is designed to generate test inputs, while our goal is to tame the anti-analysis techniques. Although IntelliDroid can generate test inputs effectively to trigger the malicious behaviors, it will fail to trigger these behaviors when facing anti-analysis techniques. By integrating test input generation techniques with taming anti-analysis techniques, we have shown the significant improvement on the power of dynamic analysis.

## 9 CONCLUSION

In this paper, we proposed an efficient and effective approach to tame the anti-analysis techniques used by Android malware apps. To evaluate the practicality of our approach, we implemented a prototype named Droid-AntiRM to extract the anti-analysis related condition statements and tame them by bytecode instrumentation. We evaluated it on 3187 malware samples, and our evaluation result demonstrates that it can automatically extract the anti-analysis related condition statements with an acceptable accuracy 89.15%. Our evaluation also showed that when integrated with other test input generation techniques (e.g., IntelliDroid), Droid-AntiRM can effectively trigger more hidden malicious behaviors. The performance evaluation shows that Droid-AntiRM is scalable to analyze and process a large number of Android malware.

## ACKNOWLEDGMENTS

We thank the anonymous referees for their valuable comments and helpful suggestions. We also give special thanks to Dr. Federico Maggi for shepherding our paper. The work of Wang was supported by the scholarship from China Scholarships Council Program (No.201603170276). The work of Zhu was supported through NSF CNS-1618684. The work of Zhou was supported by scholarship from China Scholarships Council Program (No.201606785006), Natural Science Foundation of Guangdong Province (No.2014A030310156), Science and Technology Project of Guangzhou City (No.201707010320).

## REFERENCES

[1] American Mathematical Society 2015. *Decompiler.* American Mathematical Society. http://www.javadecompilers.com.

[2] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the ACM*

*SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 59.

[3] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket.. In *NDSS*.

[4] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2013. Instrumenting android and java applications as easy as abc. In *International Conference on Runtime Verification*. Springer, 364–381.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[6] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 217–228.

[7] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. 2015. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework.. In *NDSS*.

[8] Julian Dolby, Stephen J Fink, and Manu Sridharan. 2015. *TJ Watson libraries for analysis (WALA)*. American Mathematical Society. http://wala.sf.net.

[9] Ken Dunham, Shane Hartman, Manu Quintans, Jose Andre Morales, and Tim Strazzere. 2014. *Android Malware and Analysis*. CRC Press.

[10] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 5.

[11] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. Triggerscope: Towards detecting logic bombs in android applications. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 377–396.

[12] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *ACM Sigplan Notices*, Vol. 40. ACM, 213–223.

[13] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe.. In *NDSS*. Citeseer.

[14] Le Guan, Shijie Jia, Bo Chen, Fengwei Zhang, Bo Luo, Jingqiang Lin, Peng Liu, Xinyu Xing, and Luning Xia. 2017. Supporting Transparent Snapshot for Bare-metal Malware Analysis on Mobile Devices. In *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACM.

[15] Wenjun Hu and Z Xiao. 2014. Guess where i am-android: detection and prevention of emulator evading on android. HitCon.

[16] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. 2014. Morpheus: automatically generating heuristics to detect Android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 216–225.

[17] Nicolas Kiss, Jean-François Lalande, Mourad Leslous, and Valérie Viet Triem Tong. 2016. Kharon dataset: Android malware under a microscope. In *The Learning from Authoritative Security Experiment Results (LASER) workshop*. The USENIX Association.

[18] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 229–240.

[19] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 224–234.

[20] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*. IEEE, 231–245.

[21] Simone Mutti, Yanick Fratantonio, Antonio Bianchi, Luca Invernizzi, Jacopo Corbetta, Dhilung Kirat, Christopher Kruegel, and Giovanni Vigna. 2015. BareDroid: Large-scale analysis of Android apps on real devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 71–80.

[22] Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards Transparent Tracing and Debugging on ARM. In *In 26th USENIX Security Symposium (USENIX Security 17)*. ACM.

[23] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 77–88.

[24] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX security symposium*. 543–558.

[25] Mila Parkour. 2011. Contagio malware dump. *blog sobre compartición de malware, recurso en línea disponible en: http://contagiodump. blogspot. com/, consultado el* 8 (2011).

[26] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-Force: Force-Executing Binary Programs for Security Applications.. In *USENIX Security Symposium*. 829–844.

[27] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*. ACM, 5.

[28] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks.. In *NDSS*.

[29] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*.

[30] Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. 2015. How current android malware seeks to evade automated code analysis. In *IFIP International Conference on Information Security Theory and Practice*. Springer, 187–202.

[31] Julian Schütte, Rafael Fedler, and Dennis Titze. 2015. Condroid: Targeted dynamic analysis of android applications. In *Advanced Information Networking and Applications (AINA), 2015 IEEE 29th International Conference on*. IEEE, 571–578.

[32] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors.. In *NDSS*.

[33] Virus Total. 2012. VirusTotal-Free online virus, malware and URL scanner. *Online: https://www. virustotal. com/en* (2012).

[34] Timothy Vidas and Nicolas Christin. 2014. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 447–458.

[35] Xiaolei Wang, Yuexiang Yang, Chuan Tang, Yingzhi Zeng, and Jie He. 2016. DroidContext: Identifying Malicious Mobile Privacy Leak Using Context. In *Trustcom/BigDataSE/ISPA, 2016 IEEE*. IEEE, 807–814.

[36] Xiaolei Wang, Yuexiang Yang, and Yingzhi Zeng. 2015. Accurate mobile malware detection and classification in the cloud. *SpringerPlus* 4, 1 (2015), 583.

[37] Fengguo Wei, Sankardas Roy, Xinming Ou, and others. 2014. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1329–1341.

[38] Michelle Y Wong and David Lie. 2016. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*.

[39] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. 2017. Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART. In *In 26th USENIX Security Symposium (USENIX Security 17)*. ACM.

[40] Lok-Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis.. In *USENIX security symposium*. 569–584.

[41] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*. ACM, 68.

[42] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering*. ACM, 272–281.

[43] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. 2013. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 611–622.

[44] Yajin Zhou and Xuxian Jiang. 2012. Android malware genome project. *Disponibile a http://www. malgenomeproject. org* (2012).