GroddDroid: a Gorilla for Triggering Malicious Behaviors

Adrien Abraham, Radoniaina Andriatsimandefitra, Adrien Brunelat, Jean-François Lalande, Valérie Viet Triem Tong

GroddDroid: a Gorilla for Triggering Malicious Behaviors

A. Abraham[†], R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande[*] and V. Viet Triem Tong
EPI CIDRE, CentraleSupelec, Inria, Université de Rennes 1, CNRS
IRISA UMR 6074, F-35065 Rennes, France

[†] ENS Cachan
F-94230 Cachan, France

[*] INSA Centre Val de Loire, Univ. Orléans
LIFO EA 4022, F-18020 Bourges, France

Abstract

Android malware authors use sophisticated techniques to hide the malicious intent of their applications. They use cryptography or obfuscation techniques to avoid detection during static analysis. They can also avoid detection during a dynamic analysis. Frequently, the malicious execution is postponed as long as the malware is not convinced that it is running in a real smartphone of a real user. However, we believe that dynamic analysis methods give good results when they really monitor the malware execution. In this article[1], we propose a method to enhance the execution of the malicious code of unknown malware. We especially target malware that have triggering protections, for example branching conditions that wait for an event or expect a specific value for a variable before triggering malicious execution. In these cases, solely executing the malware is far from being sufficient. We propose to force the triggering of the malicious code by combining two contributions. First, we define an algorithm that automatically identifies potentially malicious code. Second, we propose an enhanced monkey called GroddDroid, that stimulates the GUI *of an application and forces the execution of some branching conditions if needed. The forcing is used by GroddDroid to push the execution flow towards the previously identified malicious parts of the malware and execute it. The source code for our experiments with GroddDroid is released as free software[2]. We have verified on a malware dataset that we investigated manually that the malicious code is accurately executed by GroddDroid. Additionally, on a large dataset of 100 malware we precisely identify the nature of the suspicious code and we succeed to execute it at* 28%.

[1]This work has received a French government support granted to the COMIN Labs excellence laboratory and managed by the National Research Agency in the "Investing for the Future" program under reference ANR-10-LABX-07-01.

[2]Sources available at `http://kharon.gforge.inria.fr`

1. Introduction

Between 1% [15] and 9% [9] of Android applications are identified as malware. CheetahMobile reports that most of them come from alternative markets where automatic checks and malware sanitization procedures are missing [9]. Most of the time, users are infecting their own smartphone with a repackaged version of a legitimate application containing malicious code. Identifying a potential malware by studying the required permissions becomes difficult, especially because developers have difficulties to use permissions accurately [22].

To prevent the distribution of malware, Google has developed a service called Bouncer that analyzes statically and dynamically applications submitted on Google Play. Static analysis has strong limitations since malware resorts to a lot of techniques to hide the malicious behavior within legitimate applications. They can obfuscate their code, use reflection or dynamic libraries. Additionally, a lot of interesting information are only available at runtime, for example the content and the recipient of a SMS, the content of a message received by a remote server, etc. Dynamic analysis can bring more information on malware behaviors. Research efforts have to be done on the setup of efficient dynamic analysis platforms as it is not reliable to deploy large scale analysis tools on user's devices.

Dynamic analysis faces several problems. Malware can load their code dynamically [20], detect a virtual sandboxing of the application [23], use transformation attacks to escape signature based techniques [21]. Thus, the effectiveness of dynamic analysis for building real time detection tools is an interesting and active debate. Dynamic analysis tools are only useful if the malware is executed during the analysis. If the environment is virtualized, if the network is not setup, or if some APIs are missing, some simple checks may lead a malware to prevent the run of its code. We think that this sub-problem should be addressed, and is a first important step for dynamic detection solutions.

In this article, we propose a methodology and a tool called GroddDroid to automatically trigger and execute suspicious parts of the code of an application: our goal is to take as input an application, run it on a real smartphone and modify as few as possible the control flow of the application in order to force the execution of the suspicious part of the code. To achieve such a goal, we lead a two step approach. First, we identify the suspicious parts of the bytecode and compute a score (indicator of risk) for each function of the malware. Second, we introduce a new GUI stimulator, called GroddDroid, which runs the application by clicking on all possible detected buttons. Finally, we identify the remaining parts of the malware that has not been executed and we force the required control flow statements to push the flow to the unexecuted parts previously scored.

Our experimental results show that our GroddDroid executor has better code coverage than the Monkey [14] and A3E [7]. Combined with the control flow forcing method, the triggering of malicious code increases and we measured this improvement on a dataset of malware for which we have manually identified the malicious parts. On a larger dataset we also show that GroddDroid succeeds in executing the suspicious parts previously detected.

This article is structured as follows. Section 2 describes the problem of executing malware and presents a literature review. Section 3 gives a comprehensive overview of our solution. Three sections describe the different features of GroddDroid: Section 4 explains how the malicious code is targeted, Section 5 describes how is automatized the interaction with the GUI of the applications and finally Section 6 details how GroddDroid can force branches during the execution in order to execute the previously detected suspicious code. Section 7 presents our experimental results based on two datasets, a small one and a large one, and finally Section 8 concludes the article.

2. State of the art

As the production of Android malware is increasing, writing malware is becoming a regular job: Allix et al. explain that, most of malicious codes are copy paste of online tutorials or variation of the same original malicious code [2]. Thus, families of malware can be rebuilt [12] and we can measure the improvements of the sophistication of each family [4].

Two main approaches can be considered to inspect the behavior of malware: static approaches that analyze the available material of the malware as its bytecode, its resources, and dynamic approaches that run the malware and monitors its behavior. As reported in [18], different monitoring techniques can be used. Tainting techniques follows the information in the studied application [6, 11]. Virtual machine events crafting, system calls monitoring and method tracing can be implemented in the Dalvik virtual machine or at kernel level [8].

Bläsing et al. propose an hybrid approach, with a first static step and a second one that is dynamic and run in an emulator [8]. The first step statically analyzes the malware to extract relevant patterns such as JNI calls, binary executors, usage of reflection, etc. Then, the second step runs a dynamic analysis and monitors low levels system calls. Nevertheless, the two steps does not cooperate and the benefits of the static analysis is not reused for the dynamic step. We believe that dynamic approaches are promising approaches as long as they really observe the malicious behavior of a malware. However, malware authors are full of resources to evade dynamic approaches simply in delaying their malicious execution.

Well known analysis platforms like Andrubis [16] or SmartDroid [24] do not address this problem. They trigger all possible activities and generates possible interesting events. But if the malicious code is protected, for instance waiting a special event, the malicious code will never been executed. Thus, running automatically an ordinary application is a difficult challenge. As the Monkey stimulator [14] gives insufficient results, researchers have also contributed to automate the interactions with the GUI. In [10], Choudhary et al. give an overview of the current input generators for Android. Random strategies can choose graphical elements or choose system events in order to stress the application. For example, DynoDroid [17] repeats a loop "observe-select-execute" and implements different strategies for the selection phase. Model-based exploration strategies consider each activity as a state and each event is a possible transition. For example, AndroidRipper [3] discovers new states and generates the possible transitions dynamically during the execution.

Additionally, sophisticated techniques use combinations of methods to help the dynamic exploration, e.g. using the exploration strategy of A3E's tool of Azim et al. [7]. Their tool calls the activities that can be triggered by *Intents* if they are detected in the manifest of the application. This is a simple combination of static analysis and dynamic execution that intends to increase the covering of code.

Finally, since malware developers frequently reuse the same benign application to embed different malicious codes, the authors proposes PuppetDroid which is a solution that reuses previous recorded interactions if the repackaged application's GUI looks similar [13]. This example shows that the code coverage of the application is not the best way to study malware.

In this article, we do not care about covering the benign part of the code. Thus, we base our proposal on the idea that if we identify the parts of the code that are possibly malicious, and if a *normal* execution does not trigger this part of the code, then we should *force* the flow of execution

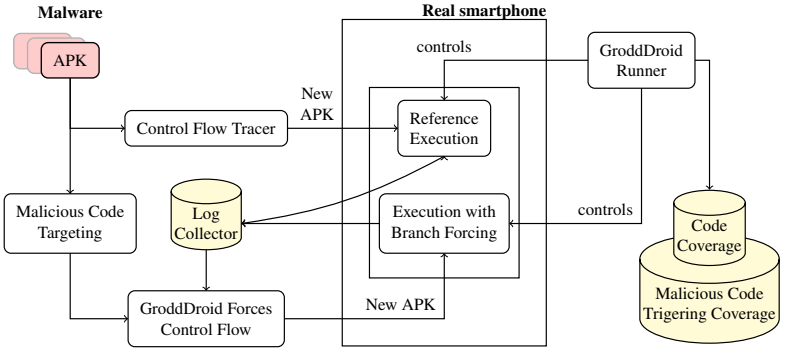

Figure 1: Overview of the GroddDroid framework

```
public static int
syracuse(int) {
  int $i0, $i1;
  $i0 := @parameter0: int;
 label1:
  if $i0 != 1 goto label2;
  return $i0;
 label2:
  $i1 = $i0 if $i1 != 0
       goto label3;
  $i0 = $i0 / 2;
  goto label1;
 label3:
  $i1 = $i0 * 3;
  $i0 = $i1 + 1;
  goto label1;
}
```

Listing 1: Jimple

```
public static int
syracuse(int n) {
  while (n != 1) {
    if (n n /= 2;
    else
      n = n*3 + 1;
  }
  return n;
}
```

Listing 2: Original Java code

in order to reach this part during subsequent executions. We show how a prior static analysis helps dynamic analysis in increasing its accuracy. The contribution of our article is precisely a static analysis that identifies the bytecode that seems dangerous followed by an automatic execution driven by the first analysis. In the next sections, after giving an overview of our solution, we depict all the components that achieve such a goal.

3. Overview

Figure 1 gives an overview of our proposal. Our approach can be divided into three different steps. First, we instrument the suspected application to observe the behavior of the application under analysis and get a reference execution. This instrumentation enables to learn which branches of the execution are taken during a run (see Control Flow Tracer on Figure 1). Thus, the GroddDroid runner executes the new APK on a smartphone in order to get the reference execution.

The second step consists in identifying the possible malicious code inside the malware using a static analysis of its bytecode (Malicious Code Targeting).

During the third step, GroddDroid uses the execution log of the reference execution to determine which control flow has to be forced to reach the parts of the code identified as malicious. A new APK is produced where the control flow is modified accordingly. The GroddDroid runner executes the new APK and new logs are generated. This step can be repeated for processing all the malicious parts of the identified code.

In the following, Section 4 describes the heuristic that identifies the potential malicious code, Section 5 explains how the reference execution is automated and Section 6 presents how the control flow is forced.

4. Automatic identification of malicious code

Android applications are packaged and distributed as APK files. These files are archives that do not contain the original Java code but only the pre-compiled Dalvik bytecode and the resources used by the application. In this article we propose a heuristic for targeting directly suspicious bytecode.

4.1. Handling application's bytecode

We extract the bytecode by using the Soot framework [5] that is able to represent the Java bytecode as several intermediate representations. The main representation is based on the Jimple language, that has the same semantic as the Java language but with fewer instructions (only 15). This reduced set of instructions makes Jimple a practical language for static analysis and optimizations. Listing 1 (resp. 2) shows an example of the Jimple (resp. Java) representation of a method `syracuse`. The type is still available and control flow constructs are similar: the `while` loop is simplified with a conditional jump and a backward jump.

With this Jimple representation, Soot allows to programmatically manipulate the application code: each instruction is wrapped into an object that extends a `Unit` object. In the example of Listing 1, an instruction such as `$i1 = $i0 % 2;` is represented by an `AssignStmt` object (for assignment statement), a subclass of `Unit`, from which some values can be accessed, like the target of the assignment (`$i1`) and the `RemExpr` (for remainder expression) of the assigned operation. The control flow of a program can also be analyzed through the conditional and unconditional jump instructions `IfStmt` and `GotoStmt`.

In order to target suspicious code, we propose to search some particular types that are more frequently encountered in malware. For extracting those types, we use Soot's ability to give the types used in each instruction of the program.

Table 1: Proposed risk scores by class categories

Category	Associated classes	Score
SMS	android.telephony.SmsManager	50
Telephony	android.telephony.TelephonyManager	20
Binary	java.lang.Runtime java.lang.Process java.lang.System	10
Dynamic	dalvik.system.BaseDexClassLoader dalvik.system.PathClassLoader dalvik.system.DexClassLoader dalvik.system.DexFile	10
Reflection	java.lang.reflect.*	3
Crypto	javax.crypto.* java.security.spec.*	3
Network	java.net.Socket java.net.ServerSocket java.net.HttpURLConnection java.net.JarURLConnection	3

4.2. Suspicious code targeting

Aafer et al. showed that some Java methods of the Android API are noticeably more frequently used in malware code than in benign applications code [1]. The difference can go from 20% to 50% of additional usage in the case of some sensible API calls such as `getSubscriberId`. In the following, we briefly summarize the API calls that are the most impacted by this difference and that will be the core of our scoring function (see [1] for full statistics).

android.telephony.TelephonyManager: `getSubscriberId` and `getDeviceId` give a unique identifier of the phone. `getLine1Number` gives the phone number associated with the SIM card.

android.app.Service: while services are perfectly common in benign applications, some overridden methods like `onCreate` frequently contain malicious code.

android.context.pm.PackageManager: this component allows listing of installed applications and their installation.

android.telephony.SmsManager: this class contains `sendTextMessage`, that allows to send send a SMS.

java.lang.{Runtime,Process}: these standard components of Java allow the application to execute native programs (`exec`), monitor their output (`getOutputStream`) and their shutdown (`waitFor`).

We propose to compute a *risk score* for each method in the bytecode: we compute the sum of the score associated to each `Unit` of the method using the scoring constants of Table 1. This table defines a score value for each suspicious class following on the observations of Aafer et al. [1]. Our heuristic is that malicious code has a higher probability to be the code with the highest score.

Table 2: Scoring result of the SaveMe malware

Method	Nb Units	Score
com.savemebeta.GTSTSR: void CHECK()	2	100
com.savemebeta.SCHKMS: void fetchContacts()	2	100
com.savemebeta.Scan: int uploadFile(java.lang.String)	12	36
com.savemebeta.Analyse: void onCreate(android.os.Bundle)	4	32
com.savemebeta.CHECKUPD: void onCreate()	4	32
com.savemebeta.CO: void onCreate()	2	16

We give an example of scoring result in Table 2 for a sample of the SaveMe [3] malware family. After inspecting manually the results, it appears that the heuristic successfully targets the malicious methods of this malware if we exclude the reflection class. Indeed, this class is used in many parts of application and the scoring was less accurate if reflection was considered, even with a low score.

Note that the score is high for two methods that have only two suspicious calls in it (TelephonyManager). On the contrary, the `uploadFile` method has 12 calls using `HttpURLConnection`. Of course, false positive targeting may happen, i.e. benign code targeted instead of the malicious code that is elsewhere in the application. Nevertheless, as shown in Section 7.1, our experimental results and manual checks show that the targeting works well.

5. Stimulating the graphical user interface

As discussed in the state of the art, many tools have covered the problem of stimulating the graphical interface in order to have a good coverage of an application. Unfortunately, most of the open source software that are cited in [10] are no longer supported. Thus, it becomes technically difficult to use the previous contributions in order to execute modern malware that use new versions of Android's APIs. For these reasons, we have chosen to recode a GUI runner, called the GroddDroid runner and to keep the Monkey [14] as a point of comparison.

5.1. Run by a monkey

The Monkey hits randomly the graphical interface and should be stopped arbitrarily because there is not guarantee that all possible activities have been visited.

The Monkey is often combined with the generation of events like SMS or phone calls and by starting all possible activities and services. Using such techniques [16] helps the Monkey but cannot achieve good results as we show at the end of the section.

[3] `http://www.infosecurity-magazine.com/news/socialpath-malware-backs-up-to-cc`

5.2. Run by the Gorilla GroddDroid

The GroddDroid runner is based on uiautomator[4] which is a python wrapper to the Google API for testing purpose. GroddDroid pushes the malware on the smartphone and launches it main activity. For each displayed activity, GroddDroid collects the graphical elements that may trigger additional code. For now, we only collect *Button* objects as our first objective is to trigger the maximum number of activities. We could also manipulate forms, radio buttons, etc. If clicking on the button leads to a new activity, GroddDroid analyses it and repeat the same operation as before. Else, it triggers the next element of the activity or gets back to the previous activity. Of course, GroddDroid detects: dead-end activities where nothing new can be activated: GroddDroid generates the event "Go back button" to return to the previous activity; crashes: the application should be started again and GroddDroid should return to the activity where it crashed: the graphical element that triggered the crash is blacklisted; loops: the current activity has already been explored and GroddDroid should backtrack.

5.3. Code coverage comparison

Figure 2 studies the code coverage results of GroddDroid, compared to the Monkey [14] and A3E [7]. For this experiment, we used a large dataset of 100 malware samples, as described later in Section 7.1. For each bar of the graph, we compute the number of applications that have the same coverage ratio. On the top graph, we consider that a method has been covered if the runners enters the method. On the bottom graph, we consider that a branch is covered if the execution flow executes the first instruction of the branch.

For the applications that have a low coverage (less than 15%), GroddDroid and Monkey have similar results. This can be explained by the fact that some malware samples crash just after being started: we observed 23 crashes for our dataset of 100 malware. A3E has surprisingly lower performances than the two others. We believe that, because A3E has been released several years ago, the tool cannot handle correctly more recent versions of Android applications. Thus, we want to emphasize that we believe that better results could be obtained with A3E if the source code was actualized. For coverages greater than 20%, GroddDroid is slightly better than Monkey.

Of course, GroddDroid cannot go above 80%: it is well known that executing 100% of the code is extremely difficult as it would require generating all possible inputs and to be sure that no dead code is present. We give more precise results in Section 7.

[4] `https://github.com/xiaocong/uiautomator`

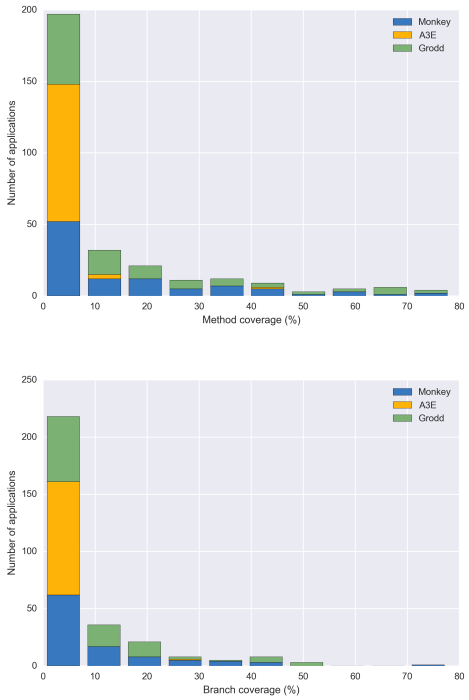

Figure 2: Number of malware with same code coverage

6. Forcing malicious code to execute

In this section, we present the ability of GroddDroid to force the conditional branches that have been identified as an execution path leading to the instructions targeted by the algorithm of Section 4. To identify how the control flow should be modified, we build our analysis on a reference execution. First, we compute an execution path from an execution point belonging to the reference execution and leading to the targeted suspicious code. Second, we modify the application bytecode in order to *force* this new path.

6.1. Control-Flow Tracer

We use Soot to instrument the application bytecode and add tracing information that allows us to know precisely which methods and conditional branches have been explored by the GroddDroid runner. We use the *Log* class of the Android API where the static method `Log.i` prints informations in a system log that is readable in real-time. The Control-Flow Tracer inserts calls to the `Log.i` method with unique identifiers, called *tags*, at the beginning of each method and conditional branch of the application. In the case of large applications, this means that we have to add thousands of these calls, but experiments have shown that it can be reliably done on every tested application.

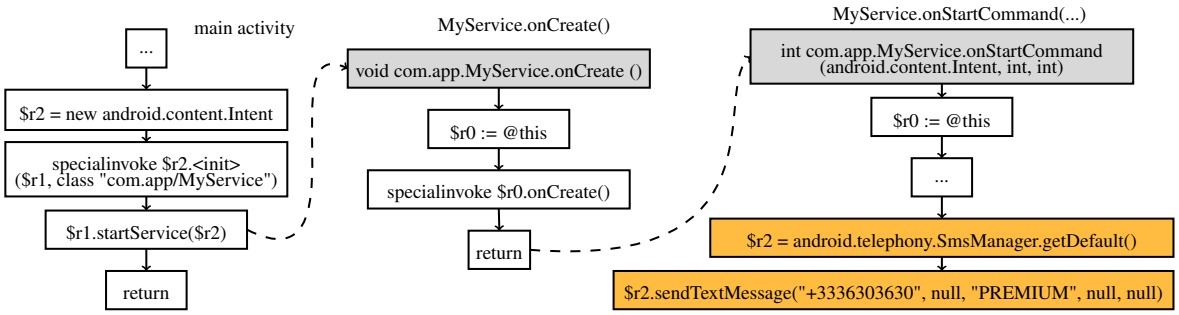

Figure 3: Example of ACFG reconstruction for an activity starting a service

GroddDroid runner executes the application once these calls are inserted in the bytecode. This execution forms the reference execution. The printed tags are collected and stored in the Log Collector (cf. Figure 1). Thus, GroddDroid obtains the precise list of branches that have been executed and not executed.

In the following, we explain how we compute an execution path that reaches these parts of the bytecode. We suppose that we want to force a particular targeted method, e.g. the most scored., and we show how we modify the reference execution to force the execution of the targeted method.

6.2. Determining an execution path

Our algorithm that determines an execution path to a targeted method is based on the control flow graph of the application. We use control flow graphs (CFG) computed from the bytecode: in these graphs, nodes are instructions of a method and the directed edges between nodes represent the possible succession of instructions. It is relatively easy to compute the control flow graph of each method appearing in the bytecode. Unfortunately, the computation of the control flow graph of the whole application is trickier.

We take as input the CFG of all methods. These graphs have one entry point and possibly several output points. Any point of a CFG $\mathcal{C}_1$ can be connected to an entry point of another CFG $\mathcal{C}_2$ if there exists an inter-procedural flow from $\mathcal{C}_1$ to $\mathcal{C}_2$, e.g. if a node of $\mathcal{C}_1$ is an instruction that invokes the method whose CFG is $\mathcal{C}_2$. In addition to direct invocations, there are also method calls that indirectly connect two CFGs. These method calls are specific to Android and are related to the creation of application components such as activities and services. For example, to display a new activity, developers must not instantiate an activity object themselves, but rather call the `startActivity` method of the Android API, with an `Intent` object as argument. The system reacts to this API call by instantiating a new activity object and calling its method `onCreate`. As Grace et al. [15] reported, even though developers do not see explicitly the calls to `onCreate`, these system actions are well-defined in the documentation. We exploit this knowledge of the well-defined semantic of these API calls to determine the implicit flows for the Android-specific behavior. Thus, we created a control flow graph for the whole application (called ACFG) from the union of CFGs of methods. We give an example of reconstruction of an implicit flow in Figure 3. In an application `com.app`, the main activity starts a service. The call `$r1.startService($r2)` calls a method of `$r1` which is a reference on `this`. Thus, we cannot see the direct link to the method `onCreate()` of `MyService` as it is called later by the system. The system also calls, when the service is created, the method `onStartCommand` where the service runs the functional code e.g. in this example, the malicious code that sends SMS. Thus, we analyze such special invocations and we create the two dotted lines in the ACFG graph to reflect these dependencies.

Once we have computed the ACFG, we can reconstruct the shortest execution path, from a targeted instruction, back to an entry point of the application, or to a method that was executed during the reference execution. This particular execution path becomes our targeted execution path.

6.3. Forcing branches

We propose here to modify the bytecode of the application to force the execution of our targeted execution path. To achieve the forcing, we first collect the tags of conditional jumps in the targeted execution path. Then, we replace each conditional jump that may divert the execution from the targeted execution with an unconditional jump to force the desired branches.

For example, Listing 3 shows a protection code that determines if the execution takes place on an emulator, and starts doing something suspicious otherwise. During a first execution on a emulator, only the branch 1 would be explored. Listing 4 shows the same sample as Jimple. To force the other branch of the conditional jump, we replace it with an unconditional jump, pointing to the first Unit of the branch 2, as shown by Listing 5.

```
if (isOnEmulator())
  return; // Branch 1
else
  manager = SmsManager.getDefault(); // Branch 2
```

Listing 3: Sample code of a conditional jump

```
  $z0 = staticinvoke <DummyClass: boolean isOnEmulator()
      >();
  if $z0 != 0 goto label1;
  return; // Branch 1
label1: // Branch 2
  $r6 = staticinvoke <SmsManager: SmsManager getDefault()
      >();
```

Listing 4: Same sample code, in Jimple

```
  $z0 = staticinvoke <DummyClass: boolean isOnEmulator()
      >();
  goto label1; // Forced branch 2
  return; // Branch 1
label1: // Branch 2
  $r6 = staticinvoke <SmsManager: SmsManager getDefault()
      >();
```

Listing 5: Same sample code with forced control flow

Then, we create a new version of the application with this modified control-flow. This new version is a reduction of the original one since it offers a strict subset of the possible executions: all executions of the new version were possible in the original application. Finally, the modified APK with one or several control flow modifications is rebuilt and executed by GroddDroid. With a new run, the runner should collect new tags in the Log Collector, indicating that the previously unexplored branches have been executed, thus triggering the possibly malicious parts of the code.

7. Experiments

7.1. Experimenting the targeting algorithm

To evaluate the soundness of the targeting algorithm presented in section 4, we have used a small collection of seven malware namely the Kharon15 dataset. Every malware belonging to the dataset has been manually reversed in order to be able to locate and understand its malicious code. More precisely, malware in this dataset have been picked from the Genome Project [25] and Contagio mobile[5] and are samples of BadNews (2013), a remote administration tool [19]; Cajino (2015), a spyware; DroidKungFu (2011), a remote administration tool [25]; MobiDash (2014), an agressive adware; SaveMe (2015), a spyware; SimpleLocker (2014), a ransomware; WipeLocker (2014), a data eraser.

We have run the targeting algorithm presented in Section 4.1 on these malware and we wanted to know if our algorithm was able to point out the code identified as malicious by the manual analysis. Table 3 details these first results. This first analysis is promising: except one analysis that has crashed, and one analysis (MobiDash) that ranked first a legitimate method but found other methods with secondary scores containing malicious code, the highest ranked methods correspond to the malicious behavior of malware.

[5] http://contagiominidump.blogspot.fr/

Table 3: Scoring results on the kharon dataset

Malware	Highest ranked method	Successful Targeting	Most scored method
BadNews	80	ok	gathers user information (phone number, IMEI, ...)
Cajino	200	ok	sends SMS with parameters from a C&C server
DroidKungFu	50	ok	starts a binary containing the exploit *udev*
MobiDash	147	wrong	gathers user information for legitimate use
SaveMe	100	ok	sends SMS with parameters from a C&C server
SimpleLocker	-	crash	-
WipeLocker	150	ok	sends SMS

We have also evaluated the results provided by the targeting algorithm on a larger dataset. This second dataset has not been as much studied as the previous used dataset but it contains a higher diversity of malicious codes. This second dataset contains one hundred malware samples obtained from AndroTotal. We evaluate how many methods are scored by the targeting algorithm and computes the distribution of the score value on this larger dataset. Results are represented graphically, for each malware numbered from 1 to 100 on the x axis, in Figure 4. For each malware we draw a square when a method is scored by the heuristic. The higher the score is, the darker the associated color is. This second experiment shows that the heuristic has computed a score greater than 0 for 83 malware over 100. Moreover, a malware has, on average, 1410 methods and 12.34 methods with a score greater than 0. In other words, less than one percent of bytecode methods are considered as potentially malicious by our targeting algorithm. Furthermore, 35.82% of the methods that are scored have a score higher than 25 which is shown by the light gray squares on Figure 4. Only 15 malware present high scored methods (greater than 150). This second experiment allows us to conclude that our targeting algorithm identifies only few methods in the malware code which reduces the scope of further analysis.

Lastly Figure 6 depicts the *genome* of the second dataset, as seen by our heuristic. Many malware use telephony calls (IMEI, etc.) and also manipulate SMS. The use of the network is of course a common usage and cryptographic primitives are used in 27 cases.

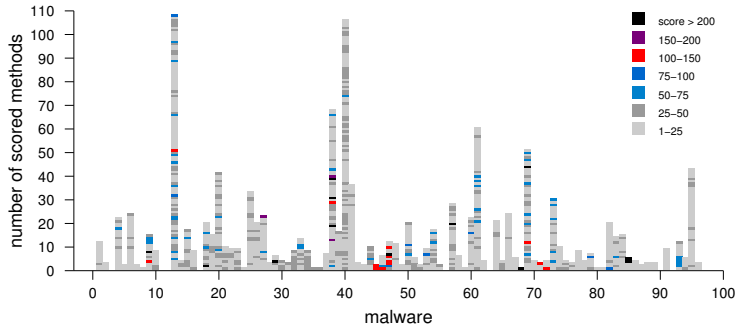

Figure 4: Scoring of the detected suspicious methods for the large dataset

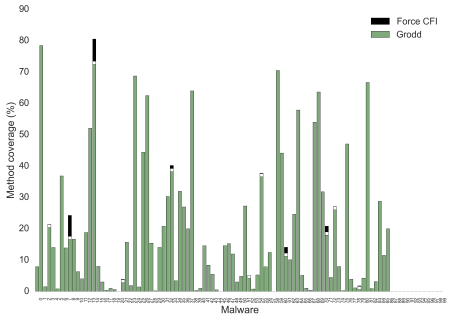

Figure 5: Method coverage for the large dataset

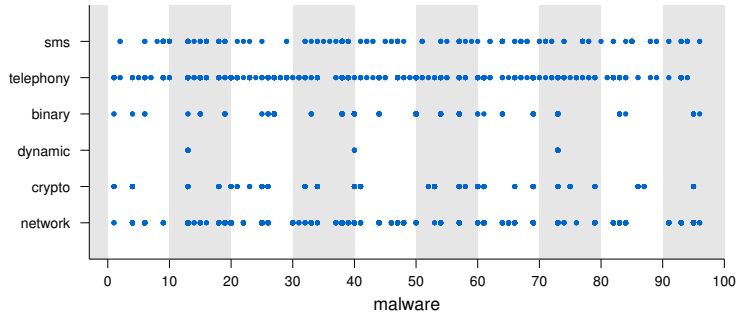

Figure 6: Genome of the large dataset with the categories of Table 1

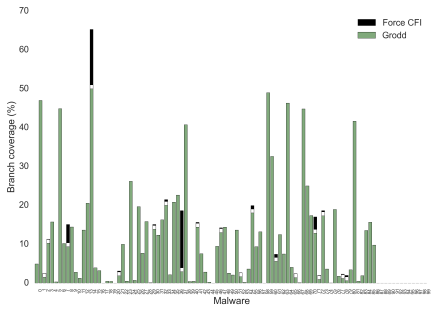

Figure 7: Branch coverage for the large dataset

7.2. Experimenting GroddDroid forcing

To conclude, we have forced the execution of the most scored method for each malware appearing on the two datasets. Our experiments have been done on a Nexus 5 smartphone under Android 4.4, connected to a Quad core PC with 4GB of RAM. Before executing each malware, we reinstall a fresh operating system in order to delete any modifications that the previous malware would have done. On the large dataset, we obtain a coverage of 16.31% of the methods (10.07% of all branches). When the highest scored method is not executed, GroddDroid runs a second run and forces the required branches. We obtain an extra $+0.19\%$ of covering for methods ($+0.41\%$ for branches). As expected, the increase is greater for branches as GroddDroid forces control flow conditions. These results are better than the coverages obtained with the Monkey (14.99% for methods, 9.35% for branches) or with A3E (1.46% for methods, 0.49% for branches). Figures 5 and 7 show the method and branch covering ratio of GroddDroid (green) for the 100 malware, numbered on the x axis from 1 to 100. The extra bar on the top (in black) represents the additional coverage obtained by GroddDroid when it forces branches.

If we just consider the suspicious methods that have been identified by the heuristic of Section 4, then the covering of these methods goes up to 24% without any forcing. If GroddDroid forces branches, the covering of the suspicious methods obtains an extra value of $+4\%$. On the other hand, Monkey executes 20% of the targeted methods. Thus, in total, GroddDroid succeeds in executing 28% of the suspicious methods with an additional $+8\%$ compared to Monkey. Note that this additional performance result of $+8\%$ is reduced by the fact that 23 malware crash during the first seconds of execution. Naturally, for these ones, GroddDroid will never be able to reach the targeted method, similarly to Monkey or A3E. For a few malware samples, GroddDroid does not succeed in launching the application. This is because the malware has no main activity and is only triggered with a system event or should be started as a service. We need to add this feature for GroddDroid in future works, in order to launch the required services or intents.

8. Conclusion

We have presented GroddDroid, a framework dedicated to the discovery and the automatic execution of malicious codes driven by static analysis of the application bytecode. The originality of GroddDroid is its ability to force branches in order to reach the suspicious code despite of the malware developer's countermeasures that protect its triggering.

Experimental results on a well studied set of malware samples showed that the targeting phase is accurate. For a dataset of 100 malware, GroddDroid succeeds in executing 16.31% of the methods on average. Using its ability to force branches, GroddDroid targets the suspicious methods and additionally succeeds in executing $+0.19\%$ of all methods. Thus, for the methods that are scored as the most suspicious, GroddDroid obtains an executing ratio of 28%.

References

[1] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. *Security and Privacy in Communication Networks*, 127:86–103, 2013.

[2] K. Allix, Q. Jerome, T. F. Bissyande, J. Klein, R. State, and Y. L. Traon. A Forensic Analysis of Android Malware – How is Malware Written and How it Could Be Detected? In *IEEE 38th Annual Computer Software and Applications Conference*, pages 384–393, Vasteras, Sweden, July 2014. IEEE Computer Society.

[3] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *27th International Conference on Automated Software Engineering*, pages 258–261, Essen, Germany, Sept. 2012. ACM Press.

[4] A. Apvrille. The evolution of mobile malware. *Computer Fraud & Security Bulletin*, 2014(8):18–20, 2014.

[5] S. Arzt, S. Rasthofer, and E. Bodden. Instrumenting Android and Java Applications as Easy as abc. In Springer, editor, *Fourth International Conference on Runtime Verification*, volume 8174, pages 364–381, Rennes, France, Sept. 2013. Springer Berlin Heidelberg.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 49, pages 259–269, Edinburgh, UK, June 2014. ACM Press.

[7] T. Azim and I. Neamtiu. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *OOPSLA'13*, Indianapolis, USA, Oct. 2013.

[8] T. Blasing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak. An Android Application Sandbox system for suspicious software detection. In *5th International Conference on Malicious and Unwanted Software*, pages 55–62. IEEE Computer Society, Oct. 2010.

[9] CheetahMobile. 2014 Half Year Security Report.

[10] S. R. Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? In *30th International Conference on Automated Software Engineering*, Lincoln, USA, 2015. IEEE Computer Society.

[11] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation*, pages 393–407, Vancouver, BC, Canada, Oct. 2010. USENIX Association.

[12] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan. Android Security: A Survey of Issues, Malware Penetration and Defenses. *IEEE Communications Surveys & Tutorials*, 17(2):998–1022, 2015.

[13] A. Gianazza, F. Maggi, A. Fattori, L. Cavallaro, and S. Zanero. PuppetDroid: A User-Centric UI Exerciser for Automatic Dynamic Analysis of Similar Android Applications. arXiv:1402.4826, Feb. 2014.

[14] Google. UI/Application Exerciser Monkey.

[15] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *10th International Conference on Mobile Systems, Applications, and Services*, pages 281–294, Low Wood Bay, Lake District, UK, 2012. ACM.

[16] M. Lindorfer and M. Neugschwandtner. ANDRUBIS-1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, San Jose, USA, Sept. 2014. IEEE Computer Society.

[17] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: an input generation system for Android apps. In *9th Joint Meeting on Foundations of Software Engineering*, page 224, Saint Pertersburg, Russia, Aug. 2013.

[18] S. Neuner, V. V. D. Veen, and M. Lindorfer. Enter Sandbox: Android Sandbox Comparison. In *3rd IEEE Mobile Security Technologies Workshop*, San Jose, USA, May 2014.

[19] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware Thanasis. In *Seventh European Workshop on System Security*, pages 1–6, Amsterdam, The Netherlands, Apr. 2014. ACM Press.

[20] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Network and Distributed System Security Symposium*, volume 14, pages 23–26, San Diego, USA, Feb. 2014. The Internet Society.

[21] V. Rastogi, Y. Chen, and X. Jiang. Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks. *IEEE Transactions on Information Forensics and Security*, 9(1):99–108, Jan. 2014.

[22] R. Stevens, J. Ganz, V. Filkov, P. Devanbu, and H. Chen. Asking for (and about) Permissions Used by Android Apps. In *5th International Conference on Social Informatics (SocInfo)*, pages 31–40, Kyoto, Japan, Nov. 2013. IEEE Computer Society.

[23] T. Vidas and N. Christin. Evading Android Runtime Analysis via Sandbox Detection. In *9th ACM Symposium on Information, Computer and Communications Security*, pages 447–458, Kyoto, Japan, June 2014. ACM Press.

[24] 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. In *Second Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 93–104, Raleigh, USA, 2012. ACM.

[25] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE Symposium on Security and Privacy*, number 4, pages 95–109, San Francisco, USA, May 2012. IEEE Computer Society.