GPFinder: Tracking the Invisible in Android Malware

Mourad Leslous, Valérie Viet Triem Tong, Jean-François Lalande, Thomas Genet

GPFinder: Tracking the Invisible in Android Malware

Mourad Leslous
EPI CIDRE
Inria, CentraleSupelec, Univ. Rennes 1, CNRS
IRISA UMR 6074,
F-35065 Rennes, France
mourad.leslous@inria.fr

Valérie Viet Triem Tong
EPI CIDRE
CentraleSupelec, Inria, Univ. Rennes 1, CNRS
IRISA UMR 6074,
F-35065 Rennes, France
valerie.viettriemtong@centralesupelec.fr

Jean-François Lalande
INSA Centre Val de Loire
Univ. Orléans
LIFO EA 4022,
F-18020 Bourges, France
jean-francois.lalande@insa-cvl.fr

Thomas Genet
EPI Celtique
Univ. Rennes 1, Inria
IRISA UMR 6074,
F-35065 Rennes, France
thomas.genet@irisa.fr

Abstract

Malicious Android applications use clever techniques to hide their real intents from the user and avoid detection by security tools. They resort to code obfuscation and dynamic loading, or wait for special events on the system like reboot or WiFi activation. Therefore, promising approaches aim to locate, study and execute specific parts of Android applications in order to monitor for suspicious behavior. They rely on Control Flow Graphs (CFGs) to obtain execution paths towards sensitive codes. We claim here that these CFGs are incomplete because they do not take into consideration implicit control flow calls, i.e., those that occur when the Android framework calls a method implemented in the application space. This article proposes a practical tool, GPFinder, exposing execution paths towards any piece of code considered as suspicious. GPFinder takes the Android framework into account and considers explicit and implicit control flow calls to build CFGs. Using GPFinder, we give global characteristics of application CFGs by studying a dataset of 14,224 malware and 2,311 goodware samples. We evaluate that 72.69% of the analyzed malicious samples have at least one suspicious method reachable only through implicit calls.

1 Introduction

The smartphone market has known a fast growth in recent years, and these devices become omnipresent in daily life. Smartphones are mostly governed by the Android operating system (87% of the smartphone market in the second quarter of 2016 [12]). Naturally, Android has become a major target for malware of numerous types [10]. Malware is, for its authors, a simple way to make money by sending messages to premium numbers, ransoming the user or remotely controlling the device's resources. To distribute their malware, authors use repackaging/piggybacking techniques: they inject malicious code in popular applications and redistribute them in alternative markets [8, 15].

To counter Android malware spread, many detection approaches have been proposed. Some of the proposed techniques perform static analysis [6, 18], while others try to observe malicious behavior [21]. Unfortunately, Android malware use clever tricks to avoid detection and, in the end, is able to perform malicious activities. For instance, to avoid static analysis, malware use code obfuscation [5, 19] and dynamic code loading from which the malicious code is downloaded from a remote server or loaded from a local file. Additionally, Android malware also tries to avoid dynamic analysis by executing its malicious code only under certain circumstances such as checking the country where the smartphone is located, or waiting for a system event, a command from a remote server, or a specific duration [9].

Recent approaches try to automatically characterize malicious behavior [2, 20, 24, 26]. They rely on a combination of static and dynamic analysis. A first static analysis of the code identifies the most suspicious locations in the code and then a particular run of the application targets the execution of the code previously identified as suspicious.

ConDroid [20] aims to launch suspicious code in the app to reveal malicious activities. It lets the definition of suspicious code types up to the user, such as dynamic loading of code. First, ConDroid finds a path from an entry point, such as lifecycle methods and input events, to the suspicious code location. Second, it performs an adaptive concolic execution by instrumenting the app and setting the necessary variable values in order to observe its behavior.

Similarly, GroddDroid [2] automatically triggers and monitors suspicious code. First, it locates code considered to be suspicious which is protected or hidden (ciphered, encoded, obfuscated or dynamically loaded) or when it calls a sensitive API method identified in [1], such as sending a SMS. Then, GroddDroid exhibits execution paths from the entry points, which can be `Activity.onCreate(Bundle)` or other similar entry points, to the suspicious code. Next, the app is instrumented by forcing the necessary branches in the execution path to reach the malicious code when the malware is launched.

These approaches strongly rely on the computation of application global control flow graphs (CFGs) that represent all execution paths in the program [3]. Such CFGs are useful only when they are complete, or at least in this context, when they contain the necessary execution paths towards suspicious code. Unfortunately, these approaches do not take into consideration all types of execution paths because they only analyze the application code, which leads to missing paths that pass though the Android framework.

The goal of this article is to automatically exhibit execution paths towards all possible suspicious locations in the code by computing global CFGs with implicit edges. This is implemented in *GPFinder* (for GroddDroid Path Finder) as the main practical outcome of this work. GPFinder helps security analysts retrieve execution paths that may trigger the malicious code, even when they pass through Android framework's callbacks. When studying these executions paths, the security analyst can understand how the suspicious code is protected by triggering conditions. We use GPFinder to study a collection of 14,224 malware samples and we show that including implicit calls to build CFGs improves the analysis. We evaluate that 72.69% of the samples have at least one suspicious code location which is only reachable through implicit calls. Furthermore, we analyze the common structures of Android malware, we highlight their favorite entry points and how they use implicit calls.

The rest of the article is structured as follows. Section 2 details the importance of implicit calls in CFGs and Section 3 discusses the impact for the Android framework. Next, Section 4 details how the implemented tool, GPFinder, takes advantage of CFGs to study the inner-structure of an Android malware set. Section 5 discusses the completeness of malware's CFGs in the literature. Lastly, Section 6 and 7 discuss the results and conclude the paper.

2 Execution Paths with Implicit Transitions

Android applications are distributed as archives that contain resource files, native libraries, an application manifest and the Dalvik bytecode. Essential building blocks of an Android app cooperate and may have independent lifecycles. *Activities* manage screens of the user interface; *Services* perform long-running background tasks such as playing music; *Content providers* manage shared data, such as SQLite databases; and *Broadcast Receivers* receive system-wide broadcasts announcing events such as SMS reception. Android applications are written mostly in Java and compiled to Dalvik bytecode. The bytecode is stored in a `.dex` file which is distributed with the resources needed to execute the application. A program in Dalvik bytecode format can be easily translated into Jimple intermediate representation [23] by Soot [22], which makes it easy to compute a CFG for each method independently at the granularity of a Jimple statement. In these graphs, an oriented edge between a node $\mathcal{A}$ and a node $\mathcal{B}$ indicates that statement $\mathcal{B}$ could be executed immediately after statement $\mathcal{A}$.

Method CFGs constitute an important step towards accurate static analysis. Nevertheless, we are mostly interested by the *global* or *inter-procedural* CFG that represents all execution scenarios for the whole application. Obtaining an execution path towards a malicious code location shows how the malware is executed, and how it is protected by triggering techniques. The global graph is constructed by connecting all the method graphs, i.e., by adding edges representing inter-procedural calls. There exist two types of inter-procedural calls: *explicit* and *implicit*.

Explicit Call: A method `a()` *explicitly* calls a method `b()` when the code of `a()` contains a call (an `invoke` statement) of the method `b()`. For example, in Listing 1 line 11, the statement `run(content)` is an explicit call to the method `MailTask.run(String)`. For such a case, we build an edge from the node representing the invoke statement `run(content)` of the method `doInBackground()` towards the node containing the first statement in the CFG of the method `run(String)`. This is represented on the right part of Figure 1

Implicit Call: A method `a()` *implicitly* calls a method `b()` when the following conditions hold:

1. `a()` contains a call (an `invoke` statement) of a method `c()` which is defined in the Android framework.
2. `c()` invokes the method `b()` either directly or through a sequence of method calls in the framework that ends by an invocation of `b()`.

```
1  public class ClientActivity extends
       Activity {
2    protected void onCreate(Bundle bundle) {
3      /* ... */
4      MailTask mt = new MailTask("",
           ((Context) this))
5      mt.execute(new Integer[0]);
6    }
7  }
8  public class MailTask extends AsyncTask {
9    protected String doInBackground
         (Integer... args) {
10     /* ... */
11     run(content);
12     return "doInBackground:" +
           this.content;
13   }
14   public void run(String arg) {
15     /* ... */
16     String str2 = ((TelephonyManager)
           context.getSystemService("phone")).
           getDeviceId();
17     ArrayList localArrayList = new
           ArrayList();
18     localArrayList.add(new
           BasicNameValuePair("imei", str2));
19     localArrayList.add(new
           BasicNameValuePair ("count",
           Integer. toString(i)));
20     localArrayList.add(new
           BasicNameValuePair("notebook",
           "Number:" + i + "\r\n" + str1));
21     String url = "######.com/MailTask.php";
22     HttpSend.postData(url, localArrayList);
23   }
24 }
```

Listing 1. Implicit call in a real malware

For example, the method `doInBackground()` in Listing 1 is implemented by the application but invoked by the Android framework. This method does not have an incoming control flow edge starting from the application itself, and that is why such methods are called callbacks. If a malicious code is located in a method which is implicitly called, it will be considered as unreachable by most existing static analyzers since they do not take into account the Android framework.

For example, Listing 1 details an example of code extracted from a spyware[1] that sends sensitive information like the device ID and the contact list to a remote server. In this malware, the entry point is `ClientActivity.onCreate()`. The malicious code is mainly the last statement, namely `HttpSend.postData(url, localArrayList)` appearing in the method `run(String)`. This statement leaks sensitive information previously retrieved by calling `context.getSystemService("phone").getDeviceId()`. The main goal of dynamic analysis is thus to observe this application executing the suspicious method `run(String)`. The method `doInBackground()` is *implicitly* called when running `MailTask.execute()`.

Obviously, the CFG of this code which is depicted in Figure 1 could be incomplete if one does not take into consideration the implicit call from `MailTask.execute(new Integer[0])` (line 5) to `MailTask.doInBackground (String... params)` (line 9). Additionally, running directly `MailTask.execute()` by instrumenting the application without finding a complete path from an entry point is meaningless since the suspicious method will be isolated from its context and could not have access to objects built in `ClientActivity.onCreate()`. A standalone analysis of the app code could not reveal the existence of such a call. Thus, we have to analyze additional code outside the application, i.e. in the Android framework to determine implicit calls and build a reliable CFG.

3 Considering the Android Framework

Implicit edges in the global CFG are due to methods implemented by the application but invoked by the Android framework (callback methods). In EdgeMiner [7], Cao *et al.* are already concerned about the lack of these callbacks in global CFGs. They pointed out that the Android framework is aware of the callbacks existence thanks to the so-called registration methods. A registration method is defined in the Android framework space and called by the application. The registration method calls the callback method directly or through a sequence of method calls inside the framework space. Cao *et al.* have statically analyzed the 24,089 classes of the Android framework and extracted a list of 5,125,472 *registration-callback* pairs responsible of implicit control flow calls. These summaries are under the form `registration#callback#position`, where `registration` and `callback` point out the involved methods and the integer position denotes the place of the `registration`'s argument responsible of calling the `callback`. For instance, EdgeMiner contains the following rules that indicates that a call to the method `execute()` induces a call to `doInBackground()`.

```
AsyncTask AsyncTask.execute(Object[]) #
Object AsyncTask.doInBackground(Object[])
# 0
```

[1] malware SHA-256: 45d21e32698d1536a73e42c1e5131c29ca94-b9d9d1bd5c744bd74ffc2af6853e

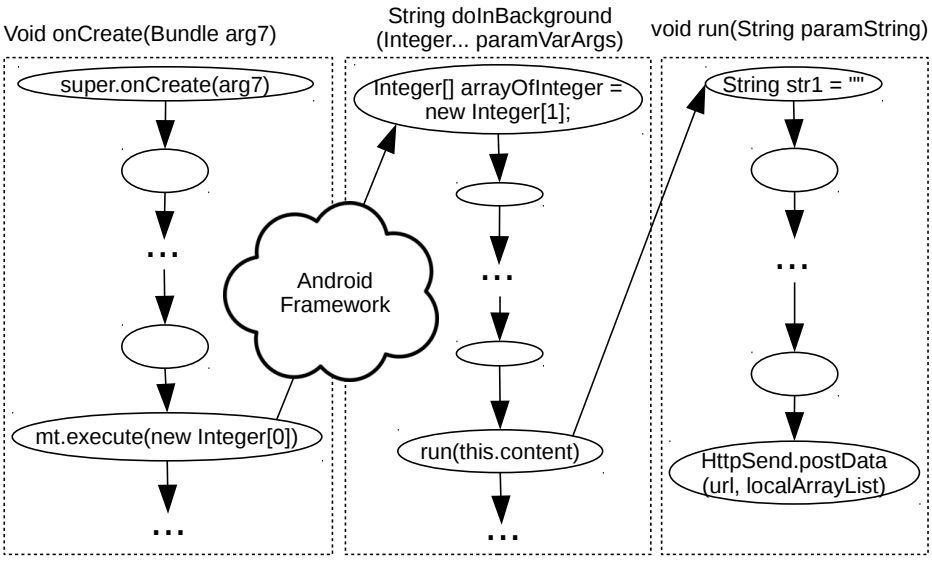

Figure 1. Global CFG with implicit calls

The link between the registration (`execute()`) and the callback (`doInBackground()`) is the defining class `AsyncTask` of the registration method which is of the same type as the callback class. This information is given by the `position 0` indicated in the above-mentioned EdgeMiner rule.

We propose to go one step further than EdgeMiner and combine the analysis of the Dalvik class hierarchy with the EdgeMiner rules in order to compute a global CFG with implicit edges of any Android application. With a global CFG computed by our tool GPFinder, we intend to find all execution paths leading towards a specific method in the application bytecode, especially suspicious ones.

GPFinder computes method graphs and then connects them with implicit edges. For each pair `(invoke(b()), a())` where `b()` is a framework method and `a()` is a method overridden in the application code, we add an edge from node `invoke(b())` to node `a()` iff. we find a rule `registration#callback#position` in EdgeMiner summaries where `b()` equals or overrides `registration` and `a()` overrides `callback`. A method `x()` overrides any `callback` or `registration` when the following conditions hold:

Name: The overriding method in the app code has the same name as in the EdgeMiner rule.

Defining class: The defining class of `x()` is a subclass of the one defining the `callback/registration`.

Return type: The type returned by `x()` is a subtype of the one returned by the `callback/registration`.

Arguments: Any argument of `x()` is a subtype of the corresponding argument in the `callback/registration`.

In addition, if the `position` $p = 0$, the `callback` class must be a subtype of the registration class. If $p > 0$, the `callback` class must be the same as the p^{th} argument of the registration method.

4 How to Reach Suspicious Code

In this section, we explain how GPFinder works and we show two practical experiments performed with it. First, we detail a complete analysis on a malware sample performing SMS fraud and exfiltrating personal data. This first experiment explains how GPFinder improves further security analysis. In a second part, we detail an analysis that takes as input a collection of 14,224 applications considered as known malware. On this malware set we exhibit all possible execution paths starting form entry points and leading to malicious code locations. The malicious code is here automatically located by a heuristic detailed hereafter, which means that the targeted code is malicious or at least suspicious. The second analysis gives an overview of the malware features such as favorite entry points, most frequent malicious code types, the average number of execution paths leading to malicious code locations, the average number of triggering conditions protecting the malicious code from dynamic analysis, and the average number of implicit calls protecting the malicious code from static analysis. Lastly, we analyze a collection of 2,311 goodware samples to emphasis the difference between the characteristics of malicious and benign applications.

4.1 GPFinder's Analysis Steps

Suspicious code location. GPFinder automatically identifies suspicious methods in the application's bytecode. For that purpose, it relies on a heuristic explained in [2]. Intuitively, the more a method uses sensitive API calls the more it is suspicious. Sensitive API methods have been split into categories related to networking, telephony, cryptography, binary code execution, SMS, and dynamic code loading. Note that one can remove or add any method or class to this list. GPFinder sets a score of risk for each category and computes the total risk for each method. Methods with non-zero scores become targets for the next analysis step.

Control flow graph computation. GPFinder computes the global control flow graph with implicit interprocedural calls and highlights all the execution paths starting from an entry point and leading to each suspicious method.

GPFinder's contribution. GPFinder gives valuable information with a relatively short time of analysis for the security experts since it automatically locates the most suspicious code, computes all execution paths towards these suspicious sites and explains how the malware is protected by triggering conditions. For example, we analyzed the malware sample mentioned in Section 2 which performs SMS fraud and exfiltrates personal data. For this piece of malware, the analysis took 13.6 seconds, and GPFinder found a total of 13 suspicious methods and exhibited 22 execution paths in the global CFG, starting from entry points and leading to methods considered as suspicious. These execution paths contains 14 implicit edges. All of them are presented in the tool's output which permits to understand how the malware exploits the framework. Finally, GPFinder details executions paths one after the other. For each of them it details the sequence of method calls, and points out how many conditions protect the malicious code.

Most of the conditions that are in the execution paths are just ordinary, nevertheless some of them are interesting from a security point of view. Indeed, some conditions are used by malware to trigger malicious actions. Listing 2 shows a triggering condition example where the *IMEI* of the device is sent to a remote server when a SMS is received. This condition is extracted from the previous malware sample.

4.2 Experiment on a Dataset of Malware

We led a similar experiment on a collection of 14,224 detected malware samples randomly chosen from a database provided by *koodous.com*. The global CFG computation takes an average time of 94.23 seconds per sample of an average APK size of 190 kB. We show below the synthesis of this experiment.

```
1 public void onReceive(Context
      paramContext, Intent intent) {
2   if(!intent.getAction().equals("android.
        provider.Telephony.SMS_RECEIVED")) {
3     /* ... */
4     localArrayList.add(new
          BasicNameValuePair("imei",
          "IMEI"));
5     new HttpSend("http://up.#######.com/",
          localArrayList).execute(new
          Integer[0]);
6   }
7   /* ... */
8 }
```

Listing 2. Triggering condition

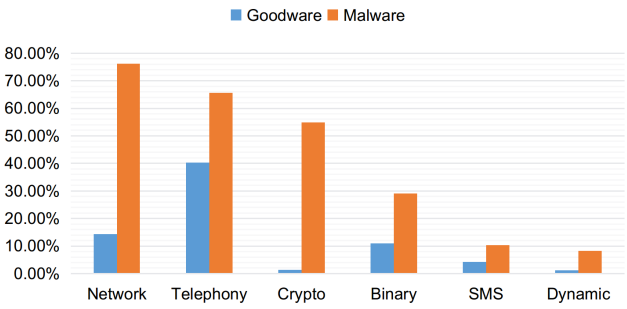

Figure 2. Ratio of APKs calling a considered category of suspicious API

Suspicious code type. In the whole malware set, we found 159,053 suspicious methods, which correspond to 4.5% of the total methods in the collection. This means an average of 11.18 suspicious methods per application. Figure 2 depicts the ratio of APKs (in orange) found in the malware collection that have a positive score risk divided by categories of suspicious code (cf. Section 4.1 for suspicious code categories).

Entry points. Android applications can be launched by a number of events, such as when the app launcher is pressed, an Intent is received, etc. Consequently, an Android application does not have only one entry point but a set of entry points like lifecycle callbacks (`onX()` methods). There exist mainly seven entry lifecycle callbacks belonging to three main categories: Callbacks allowing to create, start or resume an `Activity`, those enabling to create, start or bind a `Service`,

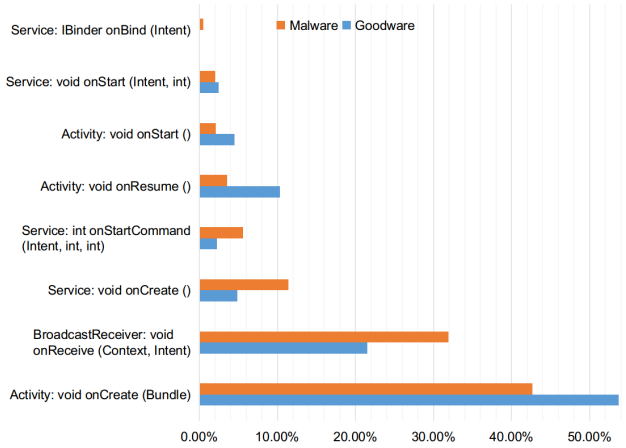

Figure 3. Use of entry points for reaching suspicious methods

and lastly a callback (`BroadcastReceiver: void onReceive`) that wakes up the application when it is notified by a system event.

Among these entry points, we evaluate which ones are the most used to reach the suspicious code. Results are detailed in Figure 3 (in orange). Our experiments reveal that malware prefer `BoadcastReceiver.onReceive(Context, Intent)` and `Activity.onCreate(Bundle)` over other entry points. The use of the latter is common as it enables to launch applications using their launcher icon. Nevertheless, the heavy usage of `onReceive()` permits to trigger malicious actions whenever the app receives an Intent broadcast like `BOOT_COMPLETED` or `SMS_RECEIVED`. These entry points permit to easily add malicious code to a benign application without much because the malicious code tends to be independent from the benign one.

Implicit transitions leading to suspicious code Implicit edges in the application's global CFG prevent security analysis tools that rely on CFGs from reaching some malicious code if the Android framework code is not taken into account.

Our results shows that 61.34% of all suspicious methods are reachable (they have at least one path leading to them from an entry point.) We found also that 47.82% (almost half) of the reachable suspicious methods are reachable **only** through implicit interprocedural calls. More globally, 72.69% of malware have at least one suspicious piece of code hidden behind implicit calls without any alternative execution path. These results show the importance of including implicit interprocedural calls in the phase of building application CFGs, since they almost double the number of reached suspicious methods in our analyzed malware dataset. Obviously, an analysis tool that relies on application CFGs to reach targeted code without taking into consideration this type of calls could miss a part of the malicious behavior.

We have also focused on the nature of implicit calls. We discovered that one of the most used implicit calls is due to the pair of registration-callback: `(Thread.start(), Runnable.run())`. Such callback are used to launch a thread from the main application. They can be used to perform heavy asynchronous tasks like downloading data from Internet or encrypting files, which may slow down activities and affect the user experience or force Android to kill the application. We find also that many callbacks from the `Handler` class are used. A `Handler` can be used to schedule messages and runnables to be executed later and to enqueue an action to be performed on a different thread. Once again, using an handler helps create an execution separated from the main thread. We also found a callback of a different nature: `(setOnClickListener(), onClick())` which is related to elements of the graphical interface suggesting that malware may be triggered by actions performed by the end user.

Triggering condition In this experiment, we found an average of 12.34 conditions per path leading to suspicious code location. These conditions are a mix of necessary checks for the app to work, and of triggering conditions that protect the malicious behavior in order to run only under certain circumstances. Some conditional branches are directly related to the malicious behavior such as `if(sms.getMessageBody().equals("GetBook"))` that checks the received attacker message and sends the phone address book to a remote server.

4.3 What about Benign Applications?

To emphasize the difference between malicious and benign applications structure, we analyzed a set of 2,311 applications considered as benign, and provided by AndroZoo [4]. The analysis took an average time of 86.29 seconds per app, and the apps have an average size of 80 Kb.

We did the same analysis on these benign applications as the one performed on the malicious ones. Figure 2 shows the usage of sensitive API calls in the analyzed benign application set (in blue). We can see that malicious applications have more suspicious calls than the benign ones. The proportions are bigger for suspicious API calls such as those used for encryption. These methods are often used by malware to decrypt binary code in order to load it dynamically and to encrypt personal data before sending it to remote servers.

Figure 3 shows the usage of entry points by the set of analyzed goodware samples (in blue). The main information that we can extract from this figure is the difference in usage of `BroadcastReceiver: void onReceive (Context, Intent)` between benign and malicious apps. As mentioned before, malware rely a lot on system events to launch malicious actions unbeknownst to the user.

5 Related Work

Being able to take into account implicit calls appears to be a key point for improving recent works on static analysis of Android malware. For instance Flowdroid [6] achieves static taint-analysis of Android applications and relies on CFGs which are computed from various sources, including layout XML files, executable code and the manifest file. This work should benefit from our computation of a global CFG that takes the framework into account. In the same way, Lillack et al. [16] use taint analysis to know which parts of an Android application are influenced by the platform's configuration, e.g. when Bluetooth is activated. Klieber et al. [14] rely on FlowDroid for intra-component taint analysis, and on Epicc [17] for inter-component analysis. This work handles calls that occur when an `Activity` calls another one to propagate the taint. Nevertheless, authors do not propose a solution for other types of implicit calls, which leads to imprecise results. Graa et al. [11] tried to get FlowDroid handle the control flows that leak information implicitly. They mainly focus on implicit flows that occur due to conditional branches. Nevertheless, they also do not also take in consideration implicit calls generated by the Android framework.

Some approaches have made attempts to handle some callbacks. Wu et al. [25] build callback graphs for synchronous callbacks, like for classes `AsyncTask` and `Handler`, in addition to application components, namely *Activity*, *Service*, *Broadcast Receiver*, and *Content Provider*. Authors focus only on main classes and methods, and neglect other callbacks that may be called by the framework. In [13], authors use lifecyle callbacks of Android applications to build a model of the application and then detect malicious behaviors. This approach focuses only on lifecyle callbacks and does not handle other types of implicit calls.

None of the works cited above are able to handle most of the implicit calls due to the Android framework itself. As shown in Section 4, malware can easily hide behind implicit calls which implies that these approaches suffer from a lack of precision.

6 Discussion

To connect different method CFGs, GPFinder uses API summaries generated by EdgeMiner which is built for Android version 4.2. Thus, for a better results, it should be updated. Nevertheless, as we showed in Section 4.2, the most used implicit calls are related to multitasking and message exchange, which have not changed a lot since Android 4.2 as far as we know.

Our experiments show that we can almost double the coverage of suspicious code by including implicit calls while building global CFGs, although, there is no other accurate implicit calls tool to compare GPFinder to. Thus, we do not have statistics about the accuracy of our tool, but it depends on the used summaries, in this case EdgeMiner's.

Implicit calls can easily be used by Android malware to hide their code. This is not specific to Android malware nor to Android, but it is a feature of the Java language. However, Android heavily uses event-driven callbacks, a characteristic that can be easily exploited by malware authors.

7 Conclusion

This article proposes GPFinder, a practical solution to help security experts to understand and analyze Android malware. GPFinder determines the suspicious code locations in Android applications. Then, for each method in the bytecode considered as suspicious, GPFinder exhibits all execution paths that start from an entry point and lead to that method. For that purpose, GPFinder is the first approach able to take the Android framework itself into account by computing a global control flow graph with implicit edges related to the callback mechanism.

We have evaluated, on a collection of 14,224 Android malware samples, how implicit interprocedural calls are used by malware. Our experiments show that 72.69% of malware have at least one suspicious piece of code hidden behind implicit calls without any alternative execution path. We demonstrated that we can easily almost double the coverage of suspicious code by including implicit calls while building global CFGs. We have evaluated that malware uses an average of 12.45 conditions, including triggering ones, to protect malicious code from dynamic analysis.

Acknowledgements

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`.

All the code described here, and experiments' inputs and outputs are available at `http://kharon.gforge.inria.fr/gpfinder.html`.

References

[1] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *Security and Privacy in Communication Networks*, pages 86–103. Springer International Publishing, 2013.

[2] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande, and V. Viet Triem Tong. Grodddroid: a gorilla for triggering malicious behaviors. In *10th International Conference on Malicious and Unwanted Software*. IEEE Computer Society, 2015.

[3] F. E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[4] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 468–471. ACM, 2016.

[5] A. Apvrille and R. Nigam. Obfuscation in android malware, and how to fight back. https://www.virusbulletin.com/virusbulletin/2014/07/obfuscation-android-malware-and-how-fight-back, 2014. Accessed: 2017-06-27.

[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 *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[7] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.

[8] 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, Secondquarter 2015.

[9] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android applications. *2016 IEEE Symposium on Security and Privacy (SP)*, pages 377–396, 2016.

[10] Google. Android security : 2016 year in review. https://source.android.com/security/reports/Google_Android_Security_2016_Report_Final.pdf, March 2017. Accessed: 2017-06-27.

[11] M. Graa, N. Cuppens-Boulahia, F. Cuppens, and A. Cavalli. *Detecting Control Flow in Smarphones: Combining Static and Dynamic Analyses*, pages 33–47. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[12] IDC. Smartphone os market share, 2016 q2. http://www.idc.com/prodserv/smartphone-os-market-share.jsp, 2016. Accessed: 2016-10-21.

[13] M. Junaid, D. Liu, and D. Kung. Dexteroid: Detecting malicious behaviors in android apps using reverse-engineered life cycle models. *Computers and Security*, 59:92 – 117, 2016.

[14] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, pages 1–6, New York, NY, USA, 2014. ACM.

[15] L. Li, D. Li, T. F. Bissyande, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro. Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting. *IEEE Transactions on Information Forensics and Security*, 12(6):1269–1284, jun 2017.

[16] M. Lillack, C. Kästner, and E. Bodden. Tracking load-time configuration options. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 445–456, New York, NY, USA, 2014. ACM.

[17] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 543–558, Berkeley, CA, USA, 2013. USENIX Association.

[18] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS*, volume 14, pages 23–26, 2014.

[19] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: Evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 329–334, New York, NY, USA, 2013. ACM.

[20] J. Schütte, R. Fedler, and D. Titze. Condroid: Targeted dynamic analysis of android applications. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, pages 571–578, March 2015.

[21] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *Network and Distributed System Security (NDSS) Symposium*, 2015.

[22] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[23] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations, 1998.

[24] M. Y. Wong and D. Lie. 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)*, 2016.

[25] T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang. Light-weight, inter-procedural and callback-aware resource leak detection for android apps. *IEEE Transactions on Software Engineering*, 42(11):1054–1076, Nov 2016.

[26] 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 *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104. ACM, 2012.