Isolating malicious code in Android malware in the wild

Valérie Viet Triem Tong, Cédric Herzog, Tomás Concepción Miranda,
Pierre Graux, Jean-François Lalande, Pierre Wilke
CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA, Rennes, France
firstname.lastname@inria.fr

Abstract

A malicious Android application often consists of a benign part which is the body of the application, and a malicious part that is added later, by repackaging. Fast and efficient analysis of Android malware depends on the analyst's ability to quickly locate malicious code and have a clear representation of it. To do this, the analysis tools must allow the suspicious code to be quickly located and isolated from the rest of the application. In this article, we propose in a first part to synthesize recent works from the literature and to refresh older research works in order to highlight the discriminating characteristics of malicious code. Then, we propose a heuristic to reveal the suspicious methods of an Android application by static analysis. Finally, we discuss an algorithm to recover the malicious graft. This graft should contain the methods considered suspicious as well as the code calling these suspicious methods.

1. Introduction

The code of an Android application consists of multiple packages, classes, Java or native methods. When such an application is malicious, statically understanding the attack requires to first accurately locate the malicious methods. Few applications contain *only* code produced by the attacker. Other malware are formed from healthy applications to which malicious code has been added: malware authors can simply decompress a benign application, then add their malicious code to it before finally repackaging it: these repackaged applications have been named piggybacked apps by Li *et al.* [12]. In the Android context, a classic assumption is that most malware are repackaged applications.

Malicious code localization can first be done manually. For example, the first datasets of malware were manually reversed: one of the pioneering projects is the Android malware Genome Project [19] presented in 2012. This dataset initially contained 1,200 malware samples, covering most of the existing Android malware families, collected between August 2010 and October 2011. This dataset was mainly maintained thanks to student efforts in charge of the reverse and classification of the malware. However, when handling larger-scale malware datasets, the manual reverse engineering does not scale anymore. Thus, we need an automatic method to locate suspicious code.

Our long term goal is to explore different methods to quickly locate malicious code. More precisely, we would like to distinguish the code that implements the malicious intent from the benign code that supports the application. To achieve this goal, we have first compiled and completed various investigations on malware and goodware to highlight the features specific to Android malware. Our experiments were conducted over one malware dataset and one goodware dataset, each containing 5000 unique applications published between 2015 and 2018. These datasets (named GM19) have been carefully constructed to avoid statistical biases. Secondly we propose a heuristic resulting from this study. This heuristic guides a static analysis by highlighting in the application control flow graph the methods considered suspicious because they are more used by malware than by goodware. Finally, we identify the malicious graft (the malicious code written by the attacker) in an application by identifying the code handling data acquired by these methods considered suspicious.

2. Android applications background

An Android application is an archive (an `.apk` file) that usually includes a collection of ressources and the code of the application compiled in the DEX file format. In this article, all the `classes.dex` are decompiled into Jimple and we compute the interprocedural control flow graph with implicit flows from this representation. A control flow graph is an oriented graph where nodes are Jimple statements and an oriented edge from a node A to a node B indicates that statement B can be executed immediately after the statement A. Such a graph can be easily recovered for each method in the bytecode using Soot [3]. The inter-procedural con-

Apps type	Average number of packages		Average number of methods		Average number of invoke statements
	Declared	Dead	Declared	Dead	
GOOD	121	69%	16,059	63%	9844
MAL	199	52%	12,539	60%	8138
Average	160	59%	14,300	62%	8991

Table 1. Average usage of packages and methods for GOOD and MAL datasets

Obfuscation techniques	Dataset nature	Ratio of obfuscated applications
Identifier Renaming (*)	Google Play	43%
	Third-party apps	73%
	Malware	63.5%
String Encryption (*)	Google Play	0%
	Third-party apps	0.1%
	Malware	5.3%
Java Reflection (*)	Google Play	48.3%
	Third-party apps	49.7%
	Malware	51%
Native method usage (**)	GOOD	25.8%
	MAL	62.5%
Packer usage (**)	GOOD	0.06%
	MAL	10.88%

(*) Experiments conducted in [5]
(**) Our own findings

Table 2. Evaluation of some obfuscation techniques used by goodware and malware

trol flow graph is constructed by connecting all the method graphs, *i.e.* by adding edges representing inter-procedural calls. Explicit inter-procedural calls permit to connect two graphs from a method A and a method B when a node in the graph for A explicitly calls the entry point of the graph for B (by an `invoke` statement). Implicit calls are (sequence of) calls that start in the application space, continue in the Android framework and end in the application space. Formally, a method `f` calls implicitly a method `g` if `f` calls a method of the runtime `h` (*e.g.* `Thread.start()`) which itself calls the method `g`. The interprocedural control flow graph with implicit flows can be recovered using GPFinder [9]. In the following, we simply refer to the interprocedural control flow graph with implicit flows of an application simply as the control flow graph or simply $\mathcal{G}$.

We rely in the following on a quantitative study on Android applications. To make this study as objective as possible and avoid statistical biases, we constructed two datasets[1] MAL and GOOD, named GM19, with the following features: MAL and GOOD each contain the same number (5000) of elements. MAL is composed of malware from VirusShare [15]; and GOOD of benign applications from AndroZoo [2], where we keep only those confirmed as non-malicious by VirusTotal [8]. We discard random samples in order to ensure a uniform temporal distribution between 2015 and 2018, and thus avoid biases in the characteristics of APKs due to date differences (for example, API methods found only in newer versions of the SDK in goodware *vs.* old API methods in malware, avoiding concept drift [14]).

Table 1 details the number of packages, classes and invoke statements found in these applications. Note that a high dead code rate is observed because APKs usually include `android.*` and `com.google.*` packages without fully using them.

[1] `https://gitlab.inria.fr/cidre-public/malcon19`

3. Discriminant Features

One of the major challenges of automatic malware analysis is to differentiate between malicious and benign code. In general, malicious code is code whose result will cause damage to whoever executes it. This code is very similar to benign code and we think that the only characteristics that can differentiate malicious code from benign code are:

1. The result of the execution of malicious code goes against the user. It can attempt to contact a remote control server, encrypt user data, access sensitive data (geolocation, contacts, IMEI, etc.), make calls or send messages to premium rate numbers, take control of the device. For all this, malicious code can use some libraries (crypto, TelephonyManager, ...) *more often* than benign code would.

2. The attacker's gain increases as long as his code is not analyzable and detectable by common anti-virus software. Therefore, the attacker tries to protect his code against **(a)** static analysis and **(b)** dynamic analysis. To do this, he obfuscates his code, and delays the execution of its payload to trigger the attack only on a real device when not under analysis.

3. On Android, some malware are distributed directly as entire applications. Many other malware are distributed by hiding in popular third-party applications, encouraging users to install it. These fake applications are referred to as piggybacked applications and are simply repackagings of benign applications where some malicious code has been grafted.

We believe that characteristics (1) refer to the content of the code while characteristic (2) refers to the form (is it obfuscated or not) (a) and structure (is the payload accessible

directly from an entry point) (b) of the malicious code and (3) impacts the internal structure of the whole application code. We now detail how these features can be exploited (or have been exploited) in Android malware analysis.

Content of the malicious code (1) In 2013, Aafer *et al.* described malware through their usage of API functions, packages, and parameter level information [1]. Relying on this description, they proposed a detection method that distinguishes malware from benign applications. In particular, their work has highlighted a list of APIs that reveal the presence of potentially malicious code.

This seminal study was very important and has been used by many approaches: mostly as a basis [7, 16–18], fewer for comparison [6]. This work was conducted in 2013 and we conducted a similar study on malware from 2019 in order to update these results.

We have listed all the API methods invoked by the samples of MAL and GOOD datasets. Among these methods, it appears that at least 30 methods are invoked by samples in the MAL dataset and are never invoked by samples in the GOOD dataset, see Figure 1. We also computed the top 30 methods with the highest difference between malware and benign apps. Our results are presented in Figure 2. Comparing to the methods highlighted by Aafer *et al.* in 2013, we notice that the preferred method for malware is still `getSubscriberId` in the `TelephonyManager` API. But the rest of the top 30 has changed: nowadays malware get more information about the device they are running on (about the network, the wifi), rather than manipulating services, SMS messages, and timers. We can note the usage of `getPackageManager`, which allows to make administrative tasks with the OS, and of `getApplicationInfo`, which allows to check if an application is debuggable. These operations can obviously be used by malware, for example to become persistent or to disable some applications that would analyze them.

Form of the malicious code (2a) The malware developer implements malicious code protections to prevent analysis and therefore detection. These protections are of two types, depending on whether they are protective against static analysis or dynamic analysis. Common obfuscation techniques that protect the code against static analysis are variable renaming, string encryption, reflection, packing (encryption of all or part of the bytecode) and usage of native code. Bacci *et al.* [4] proposed to automatically identify whether a sample under analysis has been modified by means of obfuscation techniques including disassembling followed by reassembling, repackaging, renaming packages, using call indirections, inserting junk code, renaming identifiers, encoding data, reordering code. Dong *et al.* [5] investigated how obfuscation techniques are really used by

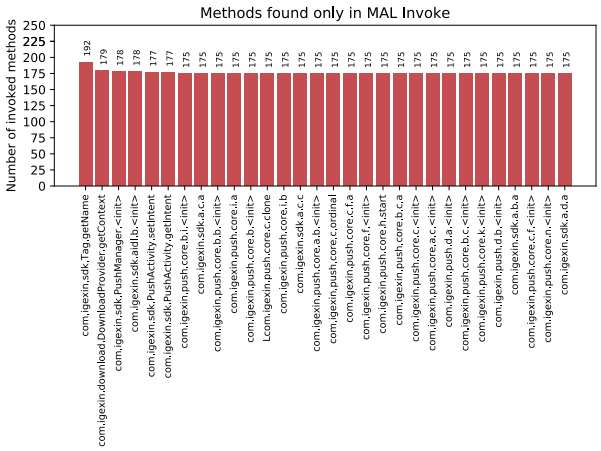

Figure 1. Methods invoked only in the samples in MAL

malware in the wild. They have evaluated how three obfuscation techniques (identifier renaming, string encryption, and Java reflection) are really used by Android applications of three typical datasets (Google Play, third-party markets, and malware). Their study has revealed that the percentage of malware using identifier renaming is 63.5%, which is more than for applications available on Google Play (43%), but slightly less than for third-party apps (73%). String encryption is not used by benign applications and only by 5.3% of malware. The proportions of reflection deployment in benign apps and malware are similar (around 50%). To complete this study, we have explored how native code obfuscation is used by malware and goodware. To detect if an application resorts to native code obfuscation we have checked the presence of methods declared as native in the DEX file of APKs from GOOD and MAL dataset. According to our investigation, we have found that malware use way more native methods than goodware (62.5% *vs.* 25.8%). This can be explained by the necessity of malware to obfuscate their code and, thus, to use native code. We have also quantified the usage of known packers by running APKiD [13] over the GOOD and MAL datatsets. We have observed that malware use more packers than goodware (10.88% *vs.* 0.06%). This can also explain the higher usage of native methods in DEX files: packers rely on native methods to load the packaged code. The results from Dong *et al.* and our own findings are gathered in Table 2.

Structure of the malicious code (2b) The protection of malware against dynamic analysis is of a different nature. A code is protected against dynamic analysis when it is not executed immediately after the application is launched. From the malware code point of view, this means that the pay-

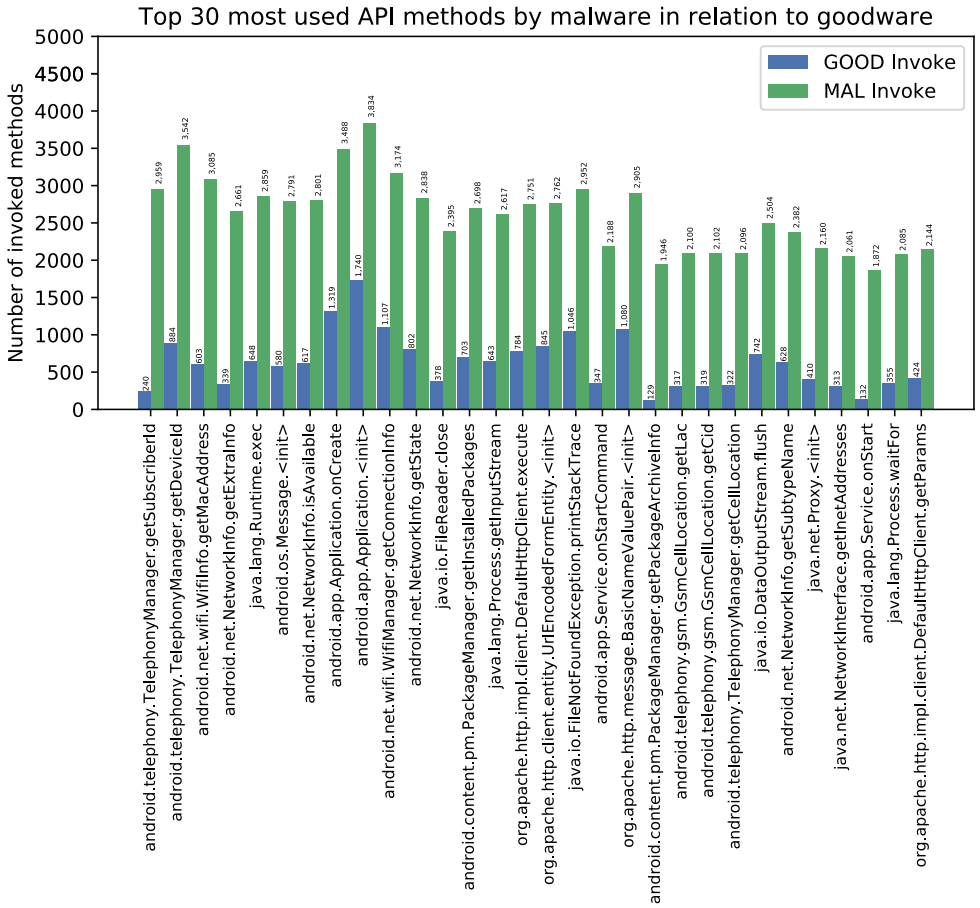

Figure 2. TOP 30 of the highest difference in methods invoked in MAL **and** GOOD

load can only be reached from an application entry point by passing through one or more conditional statements which are triggering conditions. These conditions ensure that code is only executed when the environment context appears to be suitable for malicious code, outside an analysis platform. These conditions are various: they can delay the execution in time, check the presence of emulators, check that user actions are performed. Leslous *et al.* [10] explored execution paths towards any piece of code considered as suspicious in Android applications. First, their study revealed that the malicious payload is regularly hidden behind implicit control flow calls (*i.e.* flows occurring when the Android framework calls a method implemented in the application space) making usual static analyzers believe that the malicious code is unreachable. Their study has also revealed an average of 12.34 conditions per execution path leading to suspicious code locations. 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.

Internal structure of application hosting malicious code (3) As mentioned above, malicious code is often hosted by a benign application and the resulting application is called a piggybacked application. These applications have been investigated by Li *et al.* [11] and they have built a large dataset of piggybacked and benign applications pairs. This dataset was obtained by searching for pairs of applications with highly similar code. To know the similarity between two applications, each method of each application is abstracted by a string encoding the different types of statements of the method. Then the similarity between these two applications is reduced to the similarity between two sets of strings. On this dataset, Li *et al.* described how piggybacked applications differ from benign ones: what actions are performed, what payloads are inserted, and so on. Among several insights, they claimed that piggybacking is done with little sophistication, in many cases automatically, and often via library code.

Our conclusions To conclude, we believe that the studies mentioned above provided a good indication of the characteristics of Android applications hosting malicious code. First, these applications use more libraries than others (result of Aafer *et al.* in 2013, updated here). Then, malicious code can be protected against security analysis. The protection methods that differentiate them from goodware are mainly string encryption and native code based obfuscation. Lastly, the malicious code may have been added in an initially healthy application, so it forms an independent part grafted to the original code.

In the remainder of this article, we use the first two conclusions to decide whether an Android application is suspicious or not (Section 4). We propose to evaluate the use of suspicious APIs by Android applications and assess their potential threat levels. This type of study helps us distinguish malware from goodware but is not enough to quickly locate the malicious part in all the code of an application since it does not allow us to find all the parts of the code written by the attacker, nor to highlight the structure of the malicious code. For this reason we propose in Section 5 to isolate the malicious graft from the healthy code using the data dependency graph.

4. Highlighting suspicious methods

Section 3 quantifies method invocations in MAL and GOOD datasets, allowing us to highlight which classes and methods are statistically more used by malware than by goodware. Now, we propose to build a heuristic that can be used by static analysis to study the profile of an application according to its use of APIs preferred by malware rather than by goodware. A heuristic file lists methods that should be preferred by a malware than by a goodware. Here, our heuristics files are filled using the study presented in the previous section. Our problem is therefore to select enough methods not to wrongly classify too much goodware and not to wrongly dismiss too much malware, *i.e.* to be neither too selective nor too little selective.

To tune this heuristic we separate MAL in two subsets: a training set of $4,000$ samples and a test set of $1,000$ samples. Then, we build a heuristic parametrized by a distance d and a threshold t. A distance d means that the methods listed in the heuristic have been chosen because in the previous study, these methods were invoked more than $d\%$ by malware than by goodware. Choosing a distance of 0% means we add in our heuristic a method present as much in the GOOD dataset than in the MAL dataset. H_0 is therefore very non-discriminating. On the contrary, a distance of 100% means we add in our heuristic methods exclusively present in the MAL dataset. We have computed 11 heuristics with a distance d going from H_0 to H_{100} by steps of 5%.

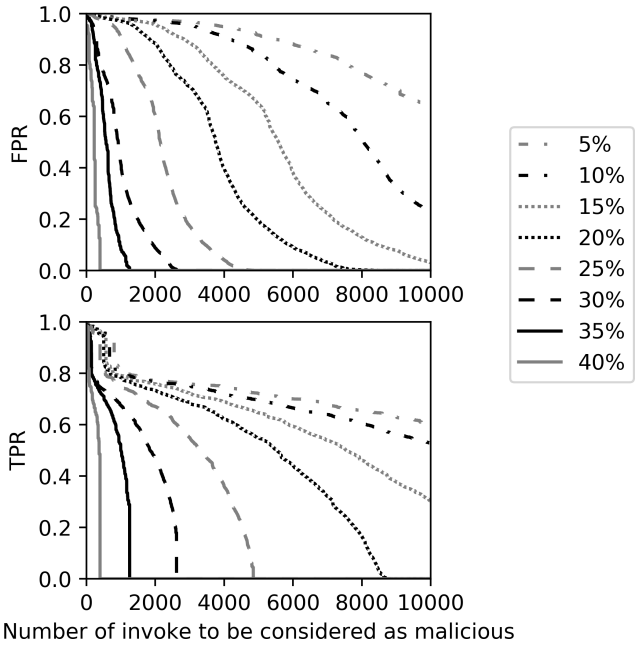

Figure 3. Evaluation of the impact of the distance choice

We measured the accuracy and relevance of these heuristics on the remaining test set of $1,000$ samples in MAL and a similar test set issued from GOOD. We count the number of invocation methods listed in the heuristics for each heuristic H_0 to H_{100}. We define a detection threshold t: an application is considered as malicious if it uses more than t methods occurring in a heuristic H_d. We evaluate the impact in term of true positive rate (TPR) and false positive rate (FPR) of a threshold value from 0 to 10000 in Figure 3.

Finally, from this results, we draw the ROC curves, as shown in Figure 4, for all heuristics. By maximizing the true positive rate while minimizing the false positive rate (point closest to the upper left of the ROC curve), we found that the best parameters are when using a distance of 35% and a threshold of 900 suspicious invokes, making H_{35} with the threshold of $t = 900$ invocations of suspicious methods above which the application is considered malicious.

5. Isolation of suspicious code

We conclude this article by focusing on the control flow graph and the data dependency graph of an application. In the control flow graph, we can highlight methods that seem suspicious because one or more of their instructions invoke a suspicious API function according to the previously described heuristics. This methodology leads to the identification of methods in the bytecode. The highlighted code

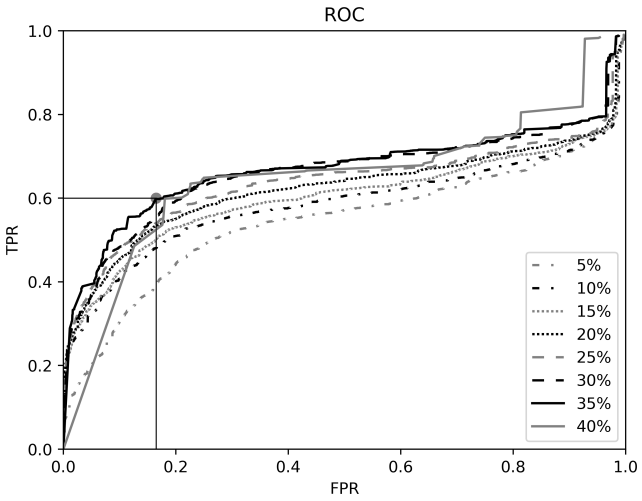

Figure 4. Comparaison of ROC curves

can be grouped or scattered in the graph. This first step can be used to decide whether or not an application is malicious as we have proposed in the previous section. This method does not allow to understand the structure of the malicious code because it only reports a set of methods without any link between them. We now propose to try to separate malicious code from healthy code by assuming that the malicious code contains the suspicious methods and that the malicious code manipulates data contaminated by instructions considered suspicious.

Suspicious instructions and suspicious methods An instruction is suspicious if it is an invoke of an external suspicious Android API or when it depends on data generated by other suspicious instructions. A method is considered suspicious if it contains at least one suspicious instruction. The set of suspicious methods is recursively computed from the data dependency graph of the application. This data dependency graph is computed using Soot [3] and GroddDroid [10]. It represents the data dependency between a bytecode instruction i and the set of previous instructions that modify the registers impacting i.

Figure 5 depicts the control flow graph of a sample of Airpush issued from the AMD dataset[2]. This sample has 830 methods, among them $23\,(2.8\%)$ invoke an external suspicious Android API (3 invoke a telephony API, 3 invoke a system API and 17 invoke a network API). The suspicious methods are colored in black in Figure 5. The computation of methods depending on at least a suspicious instruction leads to the identification of $330\,(39.8\%)$ suspicious methods in the following packages:

- `com.flurry*` : 20 methods over 44
- `com.bugsense*` : 27/60
- `com.mobclix.android*` : 101/378
- `com.ZGisNcvn*` : 177/310
- `com.boa.whis*` : 5/8

Here again, the precision depends above all on the heuristics chosen: if the heuristic is too broad, then the error is further amplified by the search for methods manipulating data that are incorrectly labeled. In a random sample[3] from MAL having 6127 methods, we found a graph composed of 11 connected components for a total of 2289 methods with the heuristic H_5, 2248 methods (loss of 41 methods) with the heuristic H_{35}, and a total of 1876 (loss of an entire package) with the heuristic H_{50}. The different grafts for H_{35} are depicted in gray in Figure 6.

For each of these heuristics, we found the following number of components depicted here by their size:

- H_5 : 49 components
  (1, 1, 1, 1, 1, 4, 1, 1, 1, 1, 6, 1, 1, 1, 1, 1, 6, 1, 1, 1, 1, 1, 2, 1, 1, 2, 1, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 12, 1, 1, 1, 1, 1, 2214)
- H_{35} : 6 components
  (1, 1, 1, 10, 1, 2372)
- H_{50} : 2 components
  (1, 2355)

Malicious connected component and grafting point When an attacker grafts malicious code to an application, he ensures that the execution of malicious code is possible through the execution of the original application. To do this, he can either add a new entry point to the application, this is what we have called a *"coarse"* graft, or he can modify one (or more) methods of the application so as to modify at least one normal execution path to drift it to the malicious code, this is what we have called a *"fine"* graft. The malicious code can be gathered in a newly added library.

From the graph point of view, a grafting point corresponds to an articulation point (*i.e.* a method whose removal would increase the number of connected components) that maximizes the number of suspicious methods contained in a single component. To highlight this suspicious graft, we highlight connected components of the control flow graph containing only suspicious methods obtained in the previous step: these nodes are suspicious either because they invoke a suspicious API or because they manipulate data acquired from suspicious APIs.

[2]SHA256: 84cde6a088b3303dfc4db7fe25443a82094d89441b68f396d2e8c2b70ce963fc

[3]SHA256: d0faedd5a230685ac027f7e1136015dc5ffa5ef7ba12344b757cd90b57141e25

Figure 5. Method in the CFG depending on a suspicious instruction in AIRPUSH

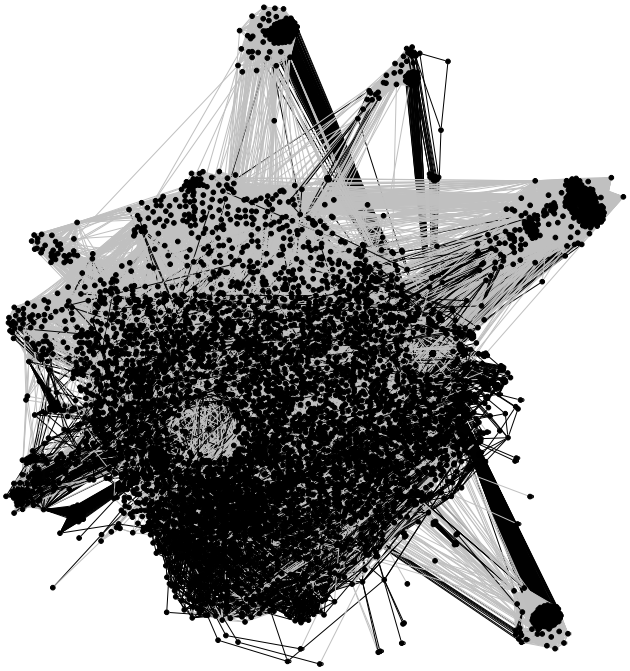

Figure 6. Estimation of a malicious graft with H_{35}

6. Conclusion

In this article we have addressed the difficult problem of accurately locating malicious code in the entire code of an Android application. First, we conducted a broad study of the different characteristics that can lead to identify this malicious code. We have updated the list of classes and packages preferably used by malware rather than goodware. This first part was done by randomly selecting goodware and malware sets *in the wild* with a uniform distribution of numbers of samples between 2015 and 2018 and a distribution of the size of the goodware similar to the distribution of size of malware. This choice of input datasets allows us to limit the bias that datasets representing only certain families can bring. We deduced a heuristic that can be used to detect whether an application is a malware or not. This heuristic relies on the classes and methods used by the application. We have shown that, using this heuristic in conjunction with the data dependency graph allows to locate malicious code grafts. Hash values and heuristic files used here are available on demand.

References

[1] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In T. Zia, A. Zomaya, V. Varadharajan, and M. Mao, editors, *Security and Privacy in Communication Networks*, pages 86–103, Cham, 2013. Springer International Publishing.

[2] 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*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

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

[4] A. Bacci, A. Bartoli, F. Martinelli, E. Medvet, and F. Mercaldo. Detection of obfuscation techniques in android applications. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ARES 2018, pages 57:1–57:9, New York, NY, USA, 2018. ACM.

[5] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild. In R. Beyah, B. Chang, Y. Li, and S. Zhu, editors, *Security and Privacy in Communication Networks*, pages 172–192, Cham, 2018. Springer International Publishing.

[6] A. Feizollah, N. B. Anuar, R. Salleh, and A. W. A. Wahab. A review on feature selection in mobile malware detection. *Digital Investigation*, 13:22 – 37, 2015.

[7] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 576–587, New York, NY, USA, 2014. ACM.

[8] Hispasec Sistemas. Virus Total. `https://www.virustotal.com`.

[9] M. Leslous, V. Viet Triem Tong, J.-F. Lalande, and T. Genet. GPFinder: Tracking the Invisible in Android Malware. In *12th International Conference on Malicious and Unwanted Software*, pages 39–46, Fajardo, oct 2017. IEEE Conputer Society.

[10] M. Leslous, V. Viet TriemTong, J. Lalande, and T. Genet. Gpfinder: Tracking the invisible in android malware. In *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 39–46, Oct 2017.

[11] 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. *Trans. Info. For. Sec.*, 12(6):1269–1284, June 2017.

[12] 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, 2017.

[13] E. Novella. Apkid: "peid" for android applications. *Black Hat Europe*, dec 2018.

[14] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro. Tesseract: Eliminating experimental bias in malware classification across space and time. *arXiv preprint arXiv:1807.07838*, 2018.

[15] VirusShare. VirusShare.com - Because Sharing is Caring. `https://virusshare.com/`.

[16] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In M. Kutyłowski and J. Vaidya, editors, *Computer Security - ESORICS 2014*, pages 163–182, Cham, 2014. Springer International Publishing.

[17] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 303–313, Piscataway, NJ, USA, 2015. IEEE Press.

[18] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1105–1116, New York, NY, USA, 2014. ACM.

[19] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109, May 2012.