Poster: Using Implicit Calls to Improve Malware Dynamic Execution

Mourad Leslous, Valérie Viet Triem Tong
EPI CIDRE
CentraleSupelec, Inria, Univ. Rennes 1, CNRS,
F-35065 Rennes, France
mourad.leslous@inria.fr,
valerie.viettriemtong@centralesupelec.fr

Jean-François Lalande
INSA Centre Val de Loire
88 Bld Lahitolle
CS 60013
18022 Bourges Cedex
jean-francois.lalande@insa-cvl.fr

***Abstract*—The number of Android malware has been increasing for the last 5 years. These malware use more often evasion techniques to hide their malicious intent and avoid analysis tools. In this work, we focus on triggering the most suspicious parts of code in malicious applications in order to monitor their behaviors using dynamic analysis tools for a better understanding of their activities. To do this, a global control flow graph (CFG) is used to exhibit an execution path to reach specific parts of code. Here we explain why using only explicit interprocedural calls may lead to a partial build of the CFG.In this poster, we explain that concept and propose a solution that improves malicious code reachability by means of integrating implicit calls.**

I. The Need for Malware Triggering

Android OS is deployed on more than 80% of the global mobile market [1]. With this domination, Android malware number knows a fast growth in app markets especially on unofficial ones. According to a recent report [2], a variety of Android malware types has been found, as phishing, ransomware and SMS/call fraud malware. Different static analysis approaches have been used to detect them. Nevertheless, malware authors use many techniques to evade static analysis, such as string obfuscation, reflection, dynamic code loading and execution of native code. They use also several techniques to evade dynamic analysis, that mostly consists in delaying the attack by executing malicious code after a specific time, after receiving a system event or a remote server command. Despite of these protection mechanisms, in this work, we focus on triggering the suspicious part of the code in order to capture automatically the dynamic behavior of suspicious application. Thus, by reproducing this dynamic analysis at large scale we aim at better understanding malware activities.

In a previous work [3], we have introduced GroddDroid, a tool that automatically triggers and executes the suspicious code of an application. Firstly, GroddDroid locates the suspicious code inside the application using a method explained in DroidApiMiner [4]. This approach is based on the fact that some framework classes are used more often by malware than by benign apps. For instance, the class *TelephonyManager* is used by malware to send premium SMSs, and the class *PackageManager* is used to detect the presence of specific security programs and to install new apps without the user's knowledge. Then, GroddDroid builds a global control flow graph (CFG) of the app. This CFG contains, for each call of a method from another one, an edge that represents it. Then, GroddDroid exhibits an execution path from an entry point, which can be an `Activity`'s `onCreate()` or other similar entry points, to the malicious code. Next, the app is instrumented to force necessary branches in this path to arrive to the malicious code. Finally, the application is executed to observe the real malware behavior using dynamic analysis tools such as Blare [5], which tracks information flows caused by the execution of the app.

II. Including Implicit Calls into CFGs

GroddDroid succeeds on a set of 100 malware to trigger 28% of suspicious code, instead of 20% using only pseudo-random event injection. These results need improvement. One of the limits of GroddDroid is that it uses only explicit method calls as interprocedural edges to build apps' global CFGs. For example, calling the malicious method `evil()` of the object `foo` from `MainActivity.onCreate` would be `foo.evil()`. This kind of calls is explicit and can be observed and used to construct the global CFG as the edge `MainActivity.onCreate()` $\longrightarrow$ `foo.evil()`. On the other hand, an implicit call is the fact that a method present in the app's code get called by the framework while there is no explicit call to it in the app's code. Let us take the example of the method `MyCmp.compare(Object, Object)` that overrides `Comparator.compare(Object, Object)` and calls the malicious method `Foo.evil()`. If we call the method `Collections.sort(list,new MyCmp)` from `MainActivity.onCreate()`, the method `MyCmp.compare(Object, Object)` will be called implicitly to sort the `list`'s elements. In this case, we have two explicit edges: `onCreate` $\longrightarrow$ `sort` and `compare` $\longrightarrow$ `evil`. The Global CFG is incomplete because it does not contain the implicit edge `sort` $\longrightarrow$ `compare`, and thus, no path from `onCreate` to `evil` (cf. Figure 1). Ignoring this type of calls may lead to lose information about the application behavior, and therefore, an incomplete CFG will prevent from monitoring a dynamic execution that reaches the suspicious functions.

We suggest to resolve this problem by integrating implicit calls into the app's global CFG to improve malicious code triggering. Obviously, listing manually all possible implicit calls in the Android framework is likely impossible due to the large codebase of the latter. CHEX [6] uses heuristics to find prospective calls of methods present in the app's code and discover data flows. It connects potential callbacks to the containing object's constructor. This proposition may give incorrect estimations because it uses just a model of the Android framework when analyzing the app. Another methodology has been proposed by EdgeMiner [7]: it analyzes the codebase of the Android framework and performs an inter-procedural backward data flow analysis to extract a list of *registration-callback* pairs. For example, it extracts the pair: `java.util.Collections: void sort (java.util.List, java.util.Comparator) # java.util.Comparator: int compare (java.lang.Object, java.lang.Object) #1`, where `1` is the position of the `compare`'s defining class (`Comparator`) in the parameters' list of the registration method `sort`. EdgeMiner analyzes the framework once for each Android version, and the results can be used to find implicit calls in Android apps running on that specific version.

At the moment, we are working on integrating the work proposed by EdgeMiner into GroddDroid. The problem we face is that results from EdgeMiner are not applicable directly, because of the inheritance relation between methods in app's code and those in EdgeMiner results. For instance, callbacks are always implemented in app's code and they override framework methods. Registration methods can also be overridden in both, framework's and app's code. In addition, the inheritance relation may be indirect. For instance, if we take the previous example of the malicious method `evil()` that has been called from `MyCmp.compare`, the EdgeMiner registration-callback pair mentioned in the previous paragraph is not applicable directly in this case, because `MyCmp` implements `Comparator`. Consequently, we have to match the registration method with another method that receives a `MyCmp` object instead of `Comparator` as parameter, and match the callback method with another method that overrides it. In this particular case, we should add an implicit edge from `Collections.sort()` to `MyCmp.compare()` instead of an edge from `Collections.sort()` to `Comparator.compare()`. As a result, for each combination of invoke site and method in the app's code, we have to run over all EdgeMiner registration-callback pairs to check if there is one that matches them and allows to add a new implicit edge to the global CFG.

III. Executing Malware

Our goal is to improve the malicious code execution beyond the 28% obtained just by taking into account explicit calls. We expect that using implicit calls may improve suspicious code runtime coverage and thus the quality of dynamic analysis. We designed an algorithm that resolves this inheritance problem and implemented it. We are now working on optimizing the execution time of our tool in order to be able to run it on big applications and on a large set of random malware. The next step is to test the new version of GroddDroid with the CFG construction method improved with implicit calls on a wide range of malware. At the end of this execution and triggering campaign, we will measure the malicious code coverage with just random event injection, forcing branches without implicit calls, and forcing branches while taking into consideration implicit calls between methods.

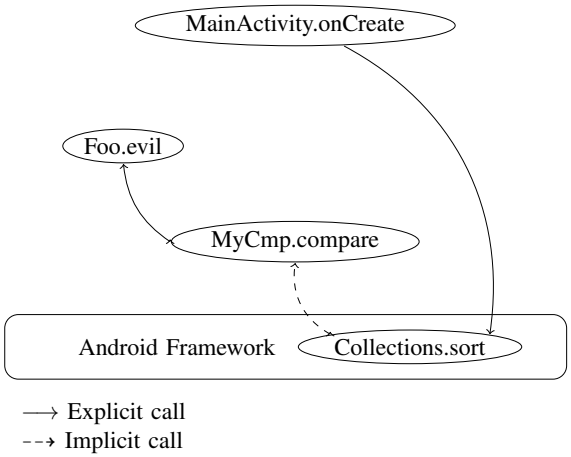

Fig. 1. `sort`⟶`compare` implicit call

References

[1] IDC. (2015) Smartphone os market share, 2015 q2. [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[2] Google, "Google report: Android security 2014 year in review," 2015. [Online]. Available: https://static.googleusercontent.com/media/source.android.com/en//security/reports/Google_Android_Security_2014_Report_Final.pdf

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

[4] 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*. Springer International Publishing, 2013, pp. 86–103.

[5] J. Zimmermann, L. Mé, and C. Bidan, *Recent Advances in Intrusion Detection: 5th International Symposium, RAID 2002 Zurich, Switzerland, October 16–18, 2002 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, ch. Introducing Reference Flow Control for Detecting Intrusion Symptoms at the OS Level.

[6] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 229–240. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382223

[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.