# How Discover a Malware using Model Checking

Fabio Martinelli, Francesco Mercaldo
Institute for Informatics and Telematics (CNR)
National Research Council of Italy
Pisa, Italy
name.surname@iit.cnr.it

Vittoria Nardone, Antonella Santone
Department of Engineering
University of Sannio
Benevento, Italy
{vnardone, santone}@unisannio.it

## ABSTRACT

Android operating system is constantly overwhelmed by new sophisticated threats and new zero-day attacks. While aggressive malware, for instance malicious behaviors able to cipher data files or lock the GUI, are not worried to circumvention users by infection (that can try to disinfect the device), there exist malware with the aim to perform malicious actions stealthy, i.e., trying to not manifest their presence to the users. This kind of malware is less recognizable, because users are not aware of their presence. In this paper we propose FormalDroid, a tool able to detect silent malicious beaviours and to localize the malicious payload in Android application. Evaluating real-world malware samples we obtain an accuracy equal to 0.94.

## 1. INTRODUCTION

In recent years, more and more mobile devices are using Android systems. As a result, the Android system becomes the major attack target to the so-called smart platforms, as phone and wearable devices. Mobile malware has various spreading methods and it is very able to hide itself. On the other hand, according to Antiy security experts, more than 10% users do not even know that there is phone malware, and more than 30% do not worry about phone malware[1].

Usually users are able to realize the infection of aggressive malware, for instance ransomware that locks the user interface and encrypts the data on the device [1]. Instead, the silent malware, typically able to send sensitive information and to generate network traffic, has as primary objective to be stealth. For this reason, users do not notice if their device generates abnormal traffic by sending sensitive information or by click-jacking: this happens because the every day user experience of the device is not significantly different. This type of malware is running for a long time without the user's awareness.

Users' carelessness makes it much easier for malware to spread; for this reason in the mobile malware landscape the *ADRD* malware family, embodying these silent behaviors, was able to infect almost a million users.

[1]http://www.antiy.net/media/reports/android_adrd_analysis.pdf



*ASIA CCS '17 April 02-06, 2017, Abu Dhabi, United Arab Emirates*



DOI: http://dx.doi.org/10.1145/3052973.3055157

In details, it can open several system services. It can also upload infected phone information (i.e., IMEI, IMSI, and version) to the control server every 6 hours and then it receives its commands. In addition, it can obtain 30 URLs from the data server and access them individually. Furthermore, it can download an installation file (.apk) to a specified directory of the SD card. Infected phones will generate lots of network traffic and cause users a lot of extra expenses.

*ADRD* needs to obtain the following system privileges: read the contact data, fully access the Internet, modify/delete the contents of the SD card, read and modify the call state, write the APN (Access Point Name) settings, check the network and WiFi state and auto-start. Once installed, *ADRD* executes itself if one of the following conditions is met: (i) 12 hours have passed since the OS was started; (ii) a change in network connectivity; (iii) the device lost ans reestablished connectivity to a network; (iv) the device received a phone call. *ADRD* samples are also able to upload device-specific information to remote servers using DES-encrypted communication. Most interesting, the malware also received search parameters from a set of URLs; these parameters are used to silently issue multiple HTTP search request. The purpose of these search requests was to increase site rankings for a web site via fraudulent clicks. *ADRD* represents the unique Android malware family in using multiple infected devices to increase the site ranking for a given web site: its main purpose is the search engine manipulation and it focused on the search engine Baidu. This happens because Baidu would pay the affiliate who brought them the search traffic a share of any revenue generated from clicks on the advertisements. As explained, the *ADRD* malicious behaviors do not affect directly the mobile user experience and for this reason the infection is not easy to detect by the device end user and, unfortunately, the anti-malware currently based on signatures are not effective in the fast detection of new threats and zero-day attacks [2]. For these reasons, in this paper we propose FormalDroid, a tool for the identification of stealth malicious behaviours in Android application. Our tool is based on formal methods and it performs a static analysis, i.e., it does not require the run of the application under analysis. Our tool is not purely syntactic and it performs behavioural-based analysis. This leads to a methodology that is also resistant to common obfuscations used by hackers. We experiment FormalDroid on the malicious samples belonging to *ADRD* family, an in the wild widespread family that exhibits stealth malicious payload, obtaining encouraging results.

The paper proceeds as follows: Section 2 explains our tool, Section 3 discusses the evaluation, finally, conclusions are drawn in Section 4.

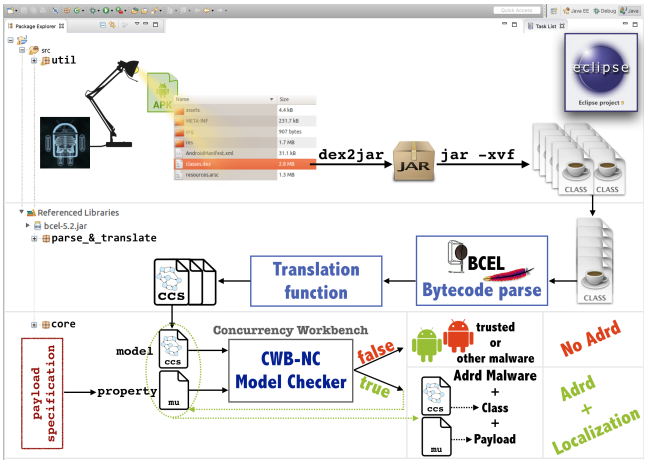


**Figure 1: The FormalDroid Workflow.**

## 2. THE FORMALDROID APPROACH

A tool for the detection of malware family has been implemented. Figure 1 shows the FormalDroid workflow, basically based on two main steps. The first step generates a Calculus of Communicating Systems (CCS) [3] specification starting from *.class* files written in Java Bytecode extracted by Android Dalvik Executable using the dex2jar tool. The Milner's CCS is one of the most well known process algebras and it is largely used for modeling complex systems. We define a Java Bytecode-to-CCS transformation function. This is defined for each instruction of the Java Bytecode. It directly translates the Java Bytecode instructions into CCS process specifications. The second step aims to investigate malicious *ADRD* family behaviours which are successively expressed using mu-calculus logic [4], which is a branching temporal logic. Starting from current literature and with the manual inspection of few samples we specify the set of properties. The CCS processes obtained in the first step are then used to verify the properties. Codes described as CCS processes are first mapped to labelled transition systems and a model checker is used. In our approach, we invoke the Concurrency Workbench of New Century (CWB-NC) [5] as formal verification environment. When the result of the CWB-NC model checker is *true*, we consider that the sample under analysis belonging to the *ADRD* family, otherwise the sample cannot be recognized as belonging to the *ADRD* family. We formulated logic rules from following characterizing *ADRD* behaviours: (i) information gathering as: device id, operator name, device model, latitude, longitude, IMSI and IMEI and (ii) opening multiple connections to HTTP addresses.

In order to highlight the novelty of FormalDroid we compare our tool only with other static tools. Moreover, there are plenty of possible criteria for such comparison. Our tool provides some features that no other tools presently offer. This can be seen by formulating seven essential criteria:

- $C_1$: does the tool support the localization of the payload?
- $C_2$: does the tool capture malicious behaviours even when they are divided in small actions apparently not harmful?
- $C_3$: does the tool performs behavioural analysis?
- $C_4$: does the tool is resilient to obfuscation?
- $C_5$: does the tool completely automated?
- $C_6$: is the tool scalable?
- $C_7$: does the tool detects Android Native Code?

**Table 1: Family Android Malware detection tools.**

| TOOL | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ |
|---|---|---|---|---|---|---|---|
| *DroidLegacy* [6] | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ |
| *Dendroid* [7] | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ |
| *Apposcopy* [8] | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ |
| Canfora et al. [9] | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ |
| *DroidClone* [10] | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ |
| FormalDroid | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |

As shown in Table 1, only FormalDroid satisfies criteria from $C_1$ to $C_4$. To the best of our knowledge, we introduce the first method supporting the payload localization. Our method is not completely automated because it needs the analyst involvement during the properties specification. We perform a manual inspection of a few samples to identify the malicious behaviours and then we specify them through logic formulae. Thus, our tool performs behavioral analysis catching malicious behaviours splitted in several small actions. FormalDroid is resilient to the common code obfuscations. Also *DroidLegacy* and *DroidClone* works assert to be resilient to code obfuscations but they do not provide any example of this. Conversely, *Apposcopy* is resilient to code obfuscations and provides an analysis testing the tool with obfuscated samples obtained using ProGuard tool[2].

### The use of formal methods

The advantages of using formal methods derive from the power of automated model checking to search a state space and the generation of the formal model is automated too. However, as shown in Table 1, our tool is not completely automated since the definition of the logic formulae expressing the malicious behavior is handmade. Verifying branching temporal logic formula allow us to recognize a malicious behaviour non just checking the presence of given sequence, like for example pattern-matching approaches. In fact, the fundamental drawback of a pattern-matching approach to malware detection is that it ignores the behaviour of a program. Commercial malware detectors use simple pattern matching approaches to malware detection. A code is considered malware if it contains a sequence of instructions that is matched by a regular expression. Recently, it has been shown that such malware detectors can be easily defeated using simple program obfuscations that are already being used by the hackers.

Another feature of our approach is that we try to reuse existing model checkers avoiding the design of custom-made model checker. Model checkers, especially the most widely used ones, are extremely sophisticated programs that have been crafted over many years by experts in the specific techniques employed by the tool. A re-implementation of the algorithms in these tools could likely yield worst performance.

### The detection on Java Bytecode and not on the source code

Performing Android malware families detection on the Bytecode and not directly on the Java code has several advantages: (i) independence of the source programming language; (ii) detection of malware families without decompilation even when source code is lacking; (iii) ease of parsing a lower-level code; (iv) independence from obfuscation.

[2]http://proguard.sourceforge.net/

# 3. THE EXPERIMENT

In this section we demonstrate the effectiveness of FormalDroid in *ADRD* malicious payload identification testing real-world malicious and legitimate applications. We gathered 861 malware involved in the experiment from the Drebin [11] dataset, a collection of freely available Android malware family labelled (91 as *ADRD* while the remaining belonging to the most widespread families).

We express the FormalDroid effectiveness in terms of Accuracy. This metric compares the number of obtained positive results with the total number of evaluated samples. The Accuracy formula is the following: $Acc = (TP+TN)/(TP+FP+FN+TN)$, where $TP$ (True Positive) and $TN$ (True Negative) are the number of sample correctly identified as belonging to *ADRD* family ($TP$) or not ($TN$). $FP$ (False Positive) represents the number of samples incorrectly labeled as belonging to *ADRD* family by our tool. $FN$ (False Negative) is the number of samples that our tool incorrectly not identifies as *ADRD* malware. $FP$ and $FN$ take into account the mismatch between our prediction and the right category which a sample belongs to. Instead, $TP$ and $TN$ take into account the correct predictions, in these cases our prediction is in according with the sample label. Table 2 shows the result achieved in *ADRD* recognizing by FormalDroid. FormalDroid obtains an accuracy equal to 0.94.

**Table 2: FormalDroid Performance Evaluation**

| Family | Samples $\in$ *ADRD* | Samples $\notin$ ADRD | TP | FP | FN | TN | Acc |
|---|---|---|---|---|---|---|---|
| *ADRD* | 91 | 770 | 84 | 48 | 7 | 722 | **0.94** |

In order to demonstrate the resilience to obfuscation, we applied to the *ADRD* malware samples evaluated a set of well-known code transformations techniques using the implementation provided in [2, 12]: (i) disassembling & reassembling, (ii) repacking, (iii) changing package name, (iv) identifier renaming, (v) data encoding, (vi) transform manifest, (vii) call indirections, (viii) code reordering, (ix) defunct methods, and (x) junk code insertion.

Detection performance resulted unchanged in testing morphed samples, as matter of fact FormalDroid marked as belonging to *ADRD* family 84 malware on 91. In order to measure FormalDroid performances, we used the *System.currentTimeMillis()* Java method that returns the current time in milliseconds. Table 3 shows the performance of our method. In particular, we consider the overall time to analyse a sample as the sum of two different contributions: the average time in seconds required to extract the class files of the application using the dex2jar tool ($t_{dex2jar}$) and the time required to obtain the response from FormalDroid ($t_{response}$). These two values are the average times, i.e., they are computed as the total time employed by FormalDroid to process the samples divided the number of samples evaluated.

The machine used to run the experiments and to take measurements was an Intel Core i5 desktop with 4 gigabyte RAM, equipped with Microsoft Windows 7 (64 bit).

# 4. CONCLUSIONS AND FUTURE WORKS

In this paper we propose FormalDroid, a tool able to detect Android malware behaviours, localizing the malicious payload in the code of the application. We evaluated our tool analyzing the *ADRD* family obtaining an accuracy equal to 0.94. As future works, we plan to extend the experiments to other widespread malware families in order to enforce the rule set we evaluated in this work.

**Table 3: FormalDroid Time Performance Evaluation.**

| $t_{dex2jar}$ | $t_{response}$ | Total Time |
|---|---|---|
| 9.8471 s | 242.1838 s | 252.0309 s |


# Acknowledgements

This work has been partially supported by H2020 EU-funded projects NeCS and C3ISP and EIT-Digital Project HII.


# 5. REFERENCES


[1] F. Mercaldo, V. Nardone, A. Santone, and C. A. Visaggio, "Ransomware steals your phone. formal methods rescue it," in *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pp. 212–221, Springer, 2016.

[2] 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*, pp. 329–334, ACM, 2013.

[3] R. Milner, *Communication and concurrency*. PHI Series in computer science, Prentice Hall, 1989.

[4] C. Stirling, "An introduction to modal and temporal logics for ccs," in *Concurrency: Theory, Language, And Architecture* (A. Yonezawa and T. Ito, eds.), LNCS, pp. 2–20, Springer, 1989.

[5] R. Cleaveland and S. Sims, "The ncsu concurrency workbench," in *CAV* (R. Alur and T. A. Henzinger, eds.), vol. 1102 of *Lecture Notes in Computer Science*, Springer, 1996.

[6] L. Deshotels, V. Notani, and A. Lakhotia, "Droidlegacy: Automated familial classification of android malware," in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, PPREW'14, (New York, NY, USA), pp. 3:1–3:12, ACM, 2014.

[7] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco, "Dendroid: A text mining approach to analyzing and classifying code structures in android malware families," *Expert Syst. Appl.*, vol. 41, pp. 1104–1117, Mar. 2014.

[8] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysisâĹŮ."

[9] G. Canfora, F. Mercaldo, and C. A. Visaggio, "An hmm and structural entropy based detector for android malware," *Comput. Secur.*, vol. 61, pp. 1–18, Aug. 2016.

[10] S. Alam, R. Riley, I. Sogukpinar, and N. Carkaci, "Droidclone: Detecting android malware variants by exposing code clones," in *2016 Sixth International Conference on Digital Information and Communication Technology and its Applications (DICTAP)*, pp. 79–84, July 2016.

[11] D. Arp, M. Spreitzenbarth, M. Huebner, H. Gascon, and K. Rieck, "Drebin: Efficient and explainable detection of android malware in your pocket," in *Proceedings of 21th NDSS*, IEEE, 2014.

[12] M. Zheng, P. P. Lee, and J. C. Lui, "Adam: an automatic and extensible platform to stress test android anti-virus systems," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 82–101, Springer, 2012.