Evaluating Analysis Tools for Android Apps: Status Quo and Robustness Against Obfuscation

Johannes Hoffmann[*]
Ruhr-University Bochum

Teemu Rytilahti[*]
Ruhr-University Bochum

Davide Maiorca[†]
University of Cagliari

Marcel Winandy[*]
Ruhr-University Bochum

Giorgio Giacinto[‡]
University of Cagliari

Thorsten Holz[*]
Ruhr-University Bochum

ABSTRACT

The recent past has shown that Android smartphones became the most popular target for malware authors. Malware families offer a variety of features that allow, among the others, to steal arbitrary data and to cause significant monetary losses. This circumstances led to the development of many different analysis methods that are aimed to assess the absence of potential harm or malicious behavior in mobile apps. In return, malware authors devised more sophisticated methods to write mobile malware that attempt to thwart such analyses. In this work, we briefly describe assumptions analysis tools rely on to detect malicious content and behavior. We then present results of a new obfuscation framework that aims to break such assumptions, thus modifying Android apps to avoid them being analyzed by the targeted systems. We use our framework to evaluate the robustness of static and dynamic analysis systems for Android apps against such transformations.

1. INTRODUCTION

Malicious software for mobile devices became prevalent in the last few years. While the first such samples were rather simple and unsophisticated, the complexity of today's malicious apps is steadily increasing. Especially malware for Android-based smartphones has significantly advanced in the recent past.

To counter this development, analysis tools must keep up with the constant evolution of malware. The typical arms race in computer security between attackers and defenders is especially distinctive in this area. Researchers from both academia and industry developed a large number of analysis methods for Android apps in recent years. These tools explored many different approaches and they took into account lessons learned from analyzing malware for desktop computers. However, existing analysis methods cannot simply be ported to Android due to many intricacies of the platform.

In this paper, we evaluate the robustness of state-of-the-art analysis tools for Android apps. Such an empirical evaluation is needed to assess how reliable existing tools are.

In summary, we make the following contributions in this short paper: (1) We briefly describe basic assumptions underlying existing analysis methods (*e. g.*, ability to reconstruct a CG) and explore how these assumptions can be thwarted (*e. g.*, flattening the CG). Our goal is to produce an obfuscated app with the same semantics as the original one, but which withstands existing analysis tools. (2) We develop a comprehensive framework that employs obfuscation techniques that aim to thwart the assumptions on which the aforementioned tools base their analysis. Our framework is available upon request. (3) We empirically evaluate our obfuscations by using our framework, and found that they severely hamper state-of-the-art tools, thus successfully evading both static and dynamic systems.

An accompanying technical report is available [10]. This technical report additionally includes a thorough survey of existing analysis tools, the assumptions they rely upon as well as more thorough evaluation results.

2. PROGRAM ANALYSIS ASSUMPTIONS

Analysis tools have several basic assumptions on which their analysis is based. For example, to perform an analysis a tool could rely on (1) having access to the call graph (CG) of the app, (2) being able to follow data flows, or (3) being able to find API function names in the code. We found that the number of underlying assumptions for all existing methods is rather small.

To produce an obfuscated app, a viable strategy is thus to systematically thwart these assumptions by means of obfuscation. For example, we can degenerate the CG such that it does not contain meaningful information, hide all literals and strings to conceal methods' semantics and invoked API methods. Additionally, types can be hidden such that bytecode only refers Java's most generic type `j.l.Object` where possible which further conceals the semantic. To this end, we systematically replace method calls, field and array accesses with indirect calls by leveraging the Reflection API.

[*]firstname.lastname@rub.de
[†]davide.maiorca@diee.unica.it
[‡]giacinto@diee.unica.it

CODASPY'16 March 09-11, 2016, New Orleans, LA, USA
© 2016 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-3935-3/16/03.
DOI: http://dx.doi.org/10.1145/2857705.2857737

The resulting app has the same semantics as the unobfuscated one, but it withstands tools which base their analysis on those basic assumptions as they are not fulfilled anymore.

3. EVALUATING THE ROBUSTNESS OF ANALYSIS TOOLS

We implemented a framework that is capable of obfuscating arbitrary Android apps with techniques briefly introduced in the previous section (for a more detailed description see our technical report [10]). Our system directly obfuscates DEX code without converting it into an intermediate format (such as JAR), and performs automatic obfuscations. In this section, we present results produced by static and dynamic analysis systems against apps obfuscated with our framework.

We have produced self-written samples that exhibit characteristics that should be detected and analyzed by the target systems. All our samples are based on a "Hello World" app but contain additional functions:

- *Direct*: creates three threads performing suspicious actions: 1) sends an SMS; 2) sends Browser's search terms over a socket; 3) like the second, but IMEI instead of searches.
- *Sleep*: calls `Thread.sleep()` to sleep 5 mins before sending an SMS.
- *Alarm*: same as *Sleep*, but uses *AlarmManager* for delaying the sending.
- *EmuDetect (ED)*: detects whether it runs in an emulator before sending an SMS and the IMEI.

We also tested whether anti-tainting techniques were able to thwart dynamic analyzers which make use of taint tracking. The aforementioned actions are often used by malware and should therefore be reported by analysis systems. Our emulator detection is rather straightforward and it is well-detectable. In general, all tests are implemented in a straightforward fashion by using standard APIs, and should thus be easily detectable.

3.1 Evaluation of Dynamic Analysis Systems

In these experiments, we tested the capabilities of dynamic systems to detect evasive behaviors under obfuscation. Because of samples' properties, such behaviors should easily be detectable.

To test such systems, we wrote the four above described applications that exhibit malicious and evasive behavior. Such applications sum up attacks that can be easily developed to thwart dynamic analysis. If a dynamic system fails at detecting such attacks, it would most likely also fail with more complex strategies. We analyzed our obfuscated samples with five well known dynamic analysis systems. We did not specifically evaluate *DroidBox* [2] as Mobile Sandbox is based on it.

We show a summary of the results of our tests in Table 1. Satisfying analysis results—meaning the analysis system was resistant to our modifications—are marked with an "✓". If provided results for that application do not contain hints for suspicious behavior (such as simply marking it as "unsuspicious"), we mark them in the table with a "ϟ". If the system does not support an analysis of a tested feature, we mark it as "*n. a.*".

The analyzed services base their taint tracking on *TaintDroid* [7], which should detect possible leaks and report them. If leaked information is being sent back to us and the service report does not provide information about it, but does so when the original and not obfuscated application is analyzed, we know that our obfuscation techniques are successfully evading taint analyses. Lost taints are marked with a "ϟ" in the "Taints" section of the table. If taint tracking is not supported we mark it as "*n. a.*".

Table 1: Results for dynamic analysis services.

Vendor	Direct	Sleep	Alarm	ED	Taints
Andrubis	✓	ϟ	✓	ϟ	ϟ
ForeSafe	ϟ	ϟ	ϟ	ϟ	*n. a.*
Mobile Sandbox	✓	✓	✓	ϟ	ϟ
NVISO	✓	ϟ	ϟ	ϟ	ϟ
Tracedroid	✓	ϟ	✓	ϟ	*n. a.*

The results obtained by the tested analysis systems presented in Table 1 demonstrate many shortcomings of existing analysis methods. Next, we provide details about them.

Andrubis [1] provides information about all network activity and rates tested apps with a maliciousness value, which were in our case always towards malicious. It failed to detect malicious actions for the *Sleep* and *EmuDetect* tests, and no emulator identifiers apart from the IMEI were changed.

Mobile-Sandbox [16] checks for malware, determines required permissions, and identifies possible entry points. It is the only analyzer that is able to correctly analyze both delaying samples, but fails the emulator detection and anti-taint test.

NVISO [13] only marked the *Direct* sample as malicious. In its provided report, all relevant information about the samples' intentions can be found, although some only in the provided PCAP file. All other samples were ranked as non-malicious and as the emulator is detectable, the ability to send an SMS goes undetected.

ForeSafe [8] provided a screenshot of apps, meaning that they weresuccessfully started, but the reports did not detect malicious activities. Our server was contacted thrice, indicating that the app was run multiple times. The IMEI and other identifiers were unchanged. ForeSafe was the only system to fail to detect maliciousness even from non-obfuscated samples, except for the SMS sending.

Tracedroid [17] reports contain a lot of information and provide a complete execution trace, including the calls (with parameters) done reflectively, completely revealing what has happened. It however fails at detecting activities performed by *Sleep* and *EmuDetect* samples.

3.2 Evaluation of Static Analysis Systems

We continue our evaluation with publicly available static analyzers. All these systems are from academia and are free to download. As our framework flattens the call graph almost completely, we expect that static tools cannot properly analyze the program's control and data flows. They additionally see almost no types, literals, nor strings.

Most public static analyzers focus on Inter-Component Communication vulnerabilities. All these tools search for corresponding sinks and sources, *i. e.*, Intents, Receivers and Content Providers. *Epicc* [14], *ComDroid* [5] and *FlowDroid* [9] were unable to properly analyze data being passed around after obfuscation. The same was true for, *Amandroid* [18], *DIDFAIL* [12] and presumably with other static flow analyzers. Because of the implicit control flows, we stop information leaks that might be detected by the precise control flow handling of *EdgeMiner* [3]. All tested tools generated no results on our obfuscated test samples. The only information available to these tools was the one defined in the Manifest file.

All the other tested public static analyzers failed at gathering information from our obfuscated test apps. For example, *SAAF* [11] was not able to retrieve meaningful information from generated program slices. The obfuscations also broke tools that rely on Java decompilation, such as *DroidChecker* [4]. We provide more details about this topic in our accompanying technical report [10]. *StaDynA* [19] was able to construct call-graphs for obfuscated applications as expected, but due to a bug in AndroGuard [6] it failed to do it for some samples. In general, their approach might be used to form proper call-graphs for further analysis, but we could not provide detailed results.

Our results show that automatically applied obfuscation to programs often completely defeats static analyzers. Most information accessible by them is only of generic value, and does not lead to informative analysis reports. The heavy use of reflections can be flagged suspicious, but it should not be used solely for tagging apps as malicious due to its wide-spread use.

4. CONCLUSION

In this short paper, we evaluated how current state-of-the-art analysis tools can cope with heavily obfuscated apps. We have developed a framework for automated obfuscation of Android apps. Our framework implements fine-grained obfuscation strategies that can be used as test benches for evaluating the robustness of analysis tools.

In our analysis, we targeted both static and dynamic analyzers, including also decompilers (which we present among other topics and much more details in our extended technical report [10]). Our test results let us conclude that many analysis tools are not capable of analyzing obfuscated apps in a satisfying manner.

The worrying aspect is that the code modifications as described in Section 2 can be automatically applied on arbitrary apps without needing access to source code. Malicious software can easily piggyback obfuscated apps, and clandestinely execute their payload while most detection systems remain blind. To this end, the recent work by Zhauniarovich *et al.* [19] and Rasthofer *et al.* [15] shows a great promise, helping to bring static analysis methods again usable even against heavily obfuscated apps.

5. REFERENCES

[1] Anubis – Malware Analysis for Unknown Binaries. https://anubis.iseclab.org/.

[2] DroidBox. http://code.google.com/p/droidbox/.

[3] 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 *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[4] P. P. Chan, L. C. Hui, and S. M. Yiu. DroidChecker: Analyzing Android applications for capability leak. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, 2012.

[5] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Intern. Conf. on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 2011.

[6] A. Desnos and G. Gueguen. Android: From reversing to decompilation. *In Proc. of Black Hat Abu Dhabi*, 2011.

[7] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*. USENIX Association, 2010.

[8] ForeSafe. ForeSafe Online Scanner. http://www.foresafe.com/scan.

[9] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Highly precise taint analysis for Android applications. Technical Report TUD-CS-2013-0113, TU Darmstadt, 2013.

[10] J. Hoffmann, T. Rytilahti, D. Maiorca, M. Winandy, G. Giacinto, and T. Holz. Evaluating Analysis Tools for Android Apps: Status Quo and Robustness Against Obfuscation. Technical Report TR-HGI-2016-001, Horst Görtz Institute for IT-security, 2016.

[11] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth. Slicing Droids: Program Slicing for Smali Code. In *SAC*. ACM, 2013.

[12] 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. ACM, 2014.

[13] NVISO. NVISO ApkScan – Scan Android applications for malware. http://apkscan.nviso.be/.

[14] 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 *USENIX Security Symposium*, 2013.

[15] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime data in android applications for identifying malware and enhancing code analysis. Technical Report TUD-CS-2015-0031, TU Darmstadt, 2015.

[16] M. Spreitzenbarth, F. C. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-Sandbox: Having a Deeper Look into Android Applications. In *SAC*. ACM, 2013.

[17] V. van der Veen and C. Rossow. Tracedroid. http://tracedroid.few.vu.nl.

[18] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.

[19] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci. StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, 2015.