COMPUTERS & SECURITY 73 (2018) 235–248

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Toward a more dependable hybrid analysis of android malware using aspect-oriented programming

Aisha I. Ali-Gombe [a,*], *Brendan Saltaformaggio* [b], *J. "Ram" Ramanujam* [c], *Dongyan Xu* [d], *Golden G. Richard III* [c]

[a] *Department of Computer and Information Science, Towson University, RM 447, 7800 York Road, Towson, MD 21252, USA*
[b] *School of Electrical and Computer Engineering, Georgia Institute of Technology, Klaus Advanced Computing Building, 266 Ferst Dr NW, Atlanta GA 30332, USA*
[c] *Center for Computation and Technology, Louisiana State University, 2027-C Digital Media Center, Baton Rouge, LA 70803, USA*
[d] *Department of Computer Science, Purdue University, 305 N. University Street, West Lafayette, IN 47907, USA*

ARTICLE INFO

Article history:
Received 24 March 2017
Received in revised form 2 November 2017
Accepted 5 November 2017
Available online 21 November 2017

Keywords:
Hybrid analysis
Bytecode weaving
Instrumentation
Dynamic execution
Android
Malware
Dataflow

ABSTRACT

The growing threat to user privacy by Android applications (app) has tremendously increased the need for more reliable and accessible analysis techniques. This paper presents *AspectDroid*[1]—an offline app-level hybrid analysis system designed to investigate Android applications for possible unwanted activities. It leverages static bytecode instrumentation to weave in analysis routines into an existing application to provide efficient dataflow analysis, detection of resource abuse, and analytics of suspicious behaviors, which are then monitored dynamically at runtime. Unlike operating system or framework dependent approaches, *AspectDroid* does not require porting from one version of Android to another, nor does it depend on a particular Android runtime, making it a more adaptable and easier to use technique. We evaluate the strength of our dataflow algorithm on 105 apps from the DroidBench corpus, with experimental results demonstrating that *AspectDroid* can detect tagged data with 94.68% accuracy. Furthermore, we compare and contrast the behavioral patterns in 100 malware samples from the Drebin dataset (Arp et al., 2014) and 100 apps downloaded from Google Play. Our results showed more traces of sensitive data exfiltration, abuse of resources, as well as suspicious use of programming concepts like reflection, native code, and dynamic classes in the malware set than the Google Play apps. In terms of runtime overhead, our experiments indicate *AspectDroid* can comprehensively log relevant security concerns with an approximate overhead of 1 MB memory and a 5.9% average increase in CPU usage.
© 2017 Published by Elsevier Ltd.

1. Introduction

Android malware and over-privileged applications are well-known for privacy violations and data leakage (Gibler et al., 2012). For instance, they transfer personal data outside the devices of end-users without their consent. In a report published by GDATA (GDATA Software, 2016), the Android platform is estimated to account for 97% of all malware on mobile devices in 2014. Over 2 million trojan applications have been detected

* *Corresponding author.*
E-mail address: aaligombe@towson.edu (A.I. Ali-Gombe).
[1] A poster version of this paper appears in CODASPY 2016 (Ali-Gombe et al., 2016).
https://doi.org/10.1016/j.cose.2017.11.006
0167-4048/© 2017 Published by Elsevier Ltd.

in 2015, representing a 50% increase from 2014. Modern malware is in use on an industrial scale by crime organizations and its development is often highly professional. In another report, Andrubis (Weichselbaum et al., 2014) performed an analysis on over a million (malicious and benign) apps, and found that 38.79% of the apps have data leakage. The percentage further increases from 13.45% in 2010 to 49.78% in 2014, and is also noted by Zhou and Jiang (2012). In many respects, this presents an even greater threat to users than before, as mobiles are entrusted with the most private of information and mobile malware can very effectively spy on users in real time. Overall, the security and privacy concerns surrounding these revelations increase the need for reliable and accessible app analysis systems.

Traditionally, Android apps are analyzed using either static or dynamic approaches. Static analysis involves the use of predetermined signatures and/or other semantic artifacts such as API calls, strings, etc. Enck et al. developed Kirin (Enck et al., 2009) which evaluates privacy risks based on the set of permissions requested, while Felt et al. (2011) and Zhou et al. (2012) analyzed Android applications by evaluating fine-grained API calls in addition to the permissions set. Other semantic-based analysis tools (Feng et al., 2014; Wu et al., 2012) examine components and intents in addition to the permissions and API calls made within the application binary.

Dynamic analysis on the other hand executes a target application in a contained environment (APIMonitor, 2012; Backes et al., 2013; Bartel et al., 2012; DroidBox, 2011; Enck et al., 2010; Falcone et al., 2013; Karami et al., 2013; Rastogi et al., 2013; Zhang and Yin, 2014a, 2014b). In general, static analysis has the advantage of high performance and coverage. Conversely, simple obfuscation can hinder the extraction of important data such as API names. Dynamic analysis on the other hand provides a better view of an app's behavior, although it is usually limited in scope to observed execution paths.

Most comprehensive dynamic analysis techniques either require instrumentation of the underlaying operating system code (DroidBox, 2011; Enck et al., 2010; Rastogi et al., 2013) or involve virtual machine introspection (Yan and Yin, 2012). They provide effective sandboxing for the analysis of the target applications, but unfortunately, such techniques are heavily dependent on OS versions and the Android runtime. Porting and flashing a new build on real devices for various versions of Android are not an easy task, which can limit the number and kind of applications that can be analyzed. Existing application-level techniques like those of APIMonitor (2012), Backes et al. (2013), Bartel et al. (2012), Falcone et al. (2013), and Karami et al. (2013) are mostly constrained to performing only API monitoring. Although systems like Capper (Zhang and Yin, 2014a, 2014b) can perform app-level taint analysis, their heavy reliance on static analysis for the extraction of taint slices makes it equally vulnerable to simple obfuscation.

In this paper, we present *AspectDroid*, a hybrid analysis system for Android applications based on the AspectJ instrumentation framework. *AspectDroid* performs static bytecode instrumentation at the application level, and does not require any particular support from the operating system or the Dalvik virtual machine. It weaves in monitoring code at compile time using a set of predefined security concerns, such as data/resource abuse and other non-traditional behaviors like reflective calls and native code execution. The target application is then executed on any Android platform of choice for which behavioral patterns are monitored and logged dynamically.

In summary, *AspectDroid* is a new hybrid analysis system for Android applications that has the following salient features:

Android Platform Independent: *AspectDroid* does not introduce code at the operating system level. Instrumented applications can run without any restrictions on both emulators and physical Android devices.

Adaptable to all Android Runtimes: *AspectDroid* is not restricted to the Dalvik virtual machine or Android runtime (ART).

Explicit Data Exfiltration: *AspectDroid* uses an efficient algorithm to track data propagation dynamically from source to sink.

Behavioral Tracing: We monitor applications for possible unwanted activities like telephony abuse, use of reflection, dynamic classes, and native code execution.

To determine the effectiveness and efficiency of *AspectDroid*, we carry out two different tests. In the first experiment, we analyze 105 Android apps from the DroidBench project for possible data exfiltration. The results show that the *AspectDroid* dataflow algorithm can accurately follow the propagation of target data from source to sink with 94.68% F-score accuracy. The second experiment analyzes the dynamic behavior of 100 malware samples from Drebin's dataset (Arp et al., 2014) and 100 apps downloaded from Google Play. Our findings are itemized based on data exfiltration, use of reflection, dynamic class loading, native code, and telephony abuse. The results of our analysis indicate that while phone-related data like IMEI are equally exfiltrated in both malware and the Google Play apps, that's not the case for user-related data like contacts where leak traces were more common in malware samples than the Google Play apps. Five malware samples use reflection for malicious purposes, such as invoking the methods of a background service to spoof user accounts and passwords. On the other hand, all the reflective call invocations in the Google Play samples did not result in any sensitive API call. Furthermore, we have seen more telephony abuse in malware than the Google Play apps, e.g., SMS was sent to all contacts on the phone without the user's consent. Nine malware samples invoke native processes 72 times, as compared to 6 for the Google Play apps.

We further use the malware dataset to measure the instrumentation overhead for dynamic execution. The results show that *AspectDroid* has limited memory overhead of around 1 MB and a reasonable 5.9% average CPU usage overhead.

The rest of the paper is organized as follows: Section 2 presents background on the AspectJ instrumentation framework; Section 3 provides an overview of *AspectDroid*'s design and associated algorithms; Section 4 presents the implementation of *AspectDroid*; Section 5 contains testing and evaluation of results; Section 6 enumerates some challenges and discusses limitations and future work; Section 7 reviews the related literature followed by Section 8 that concludes the paper.

2. Background

Instrumentation is the process of analyzing programs by adding trace code to their source code, binary code, or execution

Download English Version:

https://daneshyari.com/en/article/6884058

Download Persian Version:

https://daneshyari.com/article/6884058

Daneshyari.com