Poster: Protecting Android Apps from Repackaging by Self-Protection Code

Fumihiro Kanei, Yuta Takata, Mitsuaki Akiyama, Takeshi Yagi and Takeshi Yada
NTT Secure Platform Laboratories
{kanei.fumihiro, takata.yuta, yagi.takeshi, yada.takeshi}@lab.ntt.co.jp, akiyamam@acm.org

***Abstract*—Android malware and pirated apps created by repackaging have become a serious problem. To prevent attackers from repackaging, it is important to harden apps by using self-protection methods before distributing them. However, these countermeasures are taken by individual application developers. Thus, it depends on the developer's security awareness and implementation skills. In fact, most apps are not protected, or attackers can easily defeat an app's protection scheme. Therefore, we proposes a self-protection method that is robust against evasion attacks. The proposed method automatically builds the capability of repackaging detection into apps. It randomly splits detection code into several blocks, which are directly inserted into the bytecode of apps. Evaluation results indicate that the robustness score, which is calculated based on false positives from viewpoints of attackers, is 3.5 times higher than that with the existing method. The proposed method can also easily protect apps because it only requires their bytecode.**

I. Introduction

Application tampering, called repackaging, is a serious problem on Android apps. Repackaging is a way to create Android malware. Attackers usually inject malicious code into legitimate apps to create malware. This kind of malware steals privacy information, connects to a command and control server, and attacks other devices while the apps behave the same way as original apps from the user's viewpoint. To prevent repackaging, it is important to harden apps by using tamper-proofing techniques. For instance, countermeasures that prevent repackaged apps from working on user devices by using self-protection techniques are efficient for increasing the difficulty of repackaging. Application developers should proactively take these countermeasures during implementation. However, most apps are not protected at all. Moreover, the robustness of protection depends on the developer's security awareness and implementation skills. Therefore, we have to consider automated countermeasures with effective robustness.

II. Attack and Defense Model

In the area of self-protection for Android apps, an approach that verifies the integrity of apps is usually taken. If any tampering is found by the detection code implemented in apps, they refuse to provide their functionalities to prevent working on user devices. However, we assume that an attacker carries out evasion attacks to bypass or disable the detection code. To achieve this, an attacker would analyze protected apps by combining static and dynamic analysis techniques to locate the implementation of the detection code and disable it. Thus, we propose a self-protection method that is robust against these evasion attacks.

Permission to freely reproduce all or part of this paper for noncommercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author's employer if the paper was prepared within the scope of employment.
NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA
Copyright 2017 Internet Society, ISBN 1-891562-41-X
http://dx.doi.org/10.14722/ndss.2017.23xxx

III. Self-Protection Methods

A. Existing Method

Several self-protection methods for android apps have been proposed. Luo et al. [5] proposed an method, called *Stochastic Stealthy Network* (SSN), that automatically injects detection code into the source code of apps. As they randomly inject the detection code, it is difficult to eliminate injected code by using a static approach. Also, their method makes it difficult to debug protected apps because detection code works stochastically. Protsenko et al. [6] proposed an on-demand code decryption method. They breaks up a Dalvik executable file into individually encrypted segments. During execution, the decryption routine implemented in the native code decrypts each segment on-demand and re-encrypts after execution.

We focus on the following problems with existing methods:
Ease of deployment: A method should be easy to deploy in any situation. However, some methods require source code as input. Though a method can use the information removed during compile by using the source code, it increases the difficulty of deployment. We address this problem by only requiring a Android application package (APK) file as input.

Robustness against evasion attacks: There are several approaches to address evasion attacks. Introducing randomness when an developer implements protection is efficient way because an attacker would be forced to analyze an individual implementation. In this case, high randomness is expected for increasing the cost of analysis. We discuss improving randomness in this paper.

Side effects: The functionalities of an app should remain after being protected, and runtime overhead should be sufficiently small. Most of exiting methods meet the former requirement, but the latter remains to be met in practice.

B. Proposed Method

We propose a method for building the capability of self-protection into apps. The proposed method automatically injects randomized detection code directly into the bytecode of an app. Figure 1 shows the architecture of proposed method.

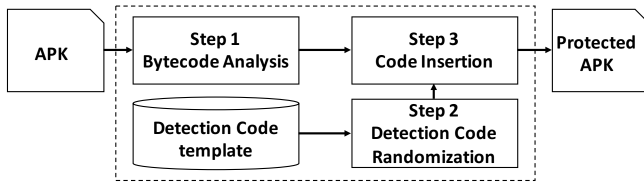

Fig. 1. Architecture of proposed method

Step 1: Bytecode Analysis
First, the proposed method conducts static and dynamic analysis to determine where to inject the detection code. It dynamically analyzes an APK file by using the Monkey tool [4] to extract methods that are not called frequently at runtime. The proposed method also leverages a debugging API (`startMethodTracing`) to trace method calls. Our method can reduce the runtime overhead to inject code into these extracted methods. After that, our method statically analyzes the extracted method in terms of access control (e.g., `public` or `private`) and exception handling (e.g., `try-catch` statement). The results of this analysis will enable the prevention of breaking the consistency of the original bytecode.

Step 2: Detection Code Randomization
Next, the proposed method randomly splits the predefined detection code template into several parts. We introduce a similar strategy with SSN [5] on this randomization step. The main difference between the proposed method and SSN is the size of the smallest unit of separation. Our method splits the detection code per instruction of the Dalvik bytecode, while SSN splits it per line of source code. Because of this *fine-grained* randomization, our method has better robustness on the bytecode, and makes it difficult to find injected detection code.

Step 3: Code Insertion
Finally, the proposed method randomly inserts randomized detection code into the methods extracted in step 1. It properly adds virtual registers and exception handling so that injected code will not break the original functionality.

IV. Evaluation

To evaluate our method, we developed a prototype system and conducted experiments. Our prototype was built upon the Soot Framework [7]. We evaluated our prototype from the viewpoint of *ease of deployment*, *robustness against evasion attacks* and *side effects*. We randomly collected 27 apps from GooglePlay [2] and F-Droid [1] for our dataset. We used Nexus 5X physical device with Android 6.0 as our dynamic analysis environment.

Ease of Deployment: To confirm ease of deployment, we applied our method to a dataset of apps. We then repackaged the protected apps by replacing their certification. After that, we installed these repackaged apps on our device and launched them. We confirmed all the repackaged apps did not work properly on the device. This means that our method properly protect apps from repackaging without requiring source code.

Robustness against evasion attacks: We compared the robustness of the detection code injected using our method and SSN [5]. This time, we evaluated robustness against static analysis in terms of *false positives from the viewpoint of attackers*. In other words, we calculated the false-positive scores that will occur when an attacker tries to find the detection code by using known parts of the detection code as a signature. First, we prepared a detection code template, and split it using both method. We then obtained the sequence of bytecode instructions contained in the separated code blocks. After that, we searched the same bytecode sequence from sample apps, and calculated the average number of exact matches. The false-positive score with our method was 3.5 times higher than that with SSN. This means that our method has better robustness against evasion attacks.

Side Effects: We evaluated the side effects from the following viewpoints; *violations of original functionalities* and *runtime overhead*. First, we manually compared the functionalities of original and protected apps by running both apps on the device. We confirmed that there was no functional difference. Next, we measured the runtime overhead by leveraging Traceview [3]. We calculated the runtime overhead based on the execution time of each method and the number of method calls. In summary, we observed 0.1% – 17% runtime overhead caused by using our method. We also observed that runtime overhead is not simply proportional to the amount of inserted detection code.

V. Conclusion and Future Work

We proposed a method for injecting randomized detection code into the bytecode of apps. The injected code detects repackaging and prevents apps from working properly on a device. The evaluation results indicate that the proposed method improves the robustness against static analysis. The proposed method can also easily protect apps because it only requires APK file as input. For future work, we will extend our method by (1) introducing multiple integrity-checking methods to prevent attackers from extracting specific API calls, such as `getPackageInfo()`, by using the dynamic monitoring approach, (2) considering a more sophisticated code-injection algorithm, for example, injecting one detection procedure into multiple methods. An attacker would leverage more advanced analysis techniques, such as dataflow analysis and program slicing for distinguishing detection code; therefore, we have to compete with these techniques.

References

[1] F-Droid. https://f-droid.org/.

[2] Google Play. https://play.google.com/store.

[3] Profiling with Traceview and dmtracedump — Android Studio. https://developer.android.com/studio/profile/traceview.html.

[4] UI/Application Exerciser Monkey — Android Studio. https://developer.android.com/studio/test/monkey.html.

[5] Lannan Luo, Yu Fu, D. W. S. Z., and Liu, P. Repackage-proofing Android Apps. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2016).

[6] Mykola Protsenko, Sebastien Kreuter, T. M. Dynamic Self-Protection and Tamperproofing for Android Apps using Native Code. In *Proceedings of the 10th International Conference on Availability, Reliability and Security (ARES)* (2015).

[7] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research (CASCON)* (1999).