Protecting Android Apps from Repackaging by Self-Protection Code  
Fumihiro Kanei, Yuta Takata, Mitsuaki Akiyama, Takeshi Yagi and Takeshi Yada  
NTT Secure Platform Laboratories  
E-mail: {kanei.fumihiro, takata.yuta, yagi.takeshi, yada.takeshi}@lab.ntt.co.jp, akiyamam@acm.org

1. Introduction

- Repackaging is a severe problem on Android  
- 80% of malware families are created by repackaging  
- Financial loss caused by pirated apps

- Countermeasures
  1. Detecting repackaged apps on the market  
  - Code-similarity approach
  2. Hardening apps by using tamper-proofing techniques  
  - Obfuscation, anti-debug, integrity-checking

- Developers should proactively protect their apps before distributing them, but:  
  - The robustness of protection depends on developer’s security awareness and implementation skills

2. Attack and Defense Model

- Self-protection for Android apps  
  - Verifying integrity of an app  
  - Repackaged apps refuse to provide their functionalities to prevent working on user devices

- Evasion attacks against self-protection mechanism  
  - An attacker uses static and dynamic analysis techniques to locate and disable the detection code  
  - Static signature matching, dynamic API monitoring, etc

3. Proposed Method

- Automatically build the capability of repackaging detection into the bytecode  
- Randomize the implementation of the detection code for improving robustness  
  - An attacker would be forced to analyze individual implementation

- Step 1: Bytecode Analysis  
  - Determine where to inject detection code  
  - Extract and analyze methods that are called few times  
  - Count the number of method calls by a debugging API, startMethodTracing()  
  - Input user event by Monkey tool  
  - Dynamic Analysis

- Step 2: Detection Code Randomization  
  - Randomly split the predefined detection code template into several parts  
  - The size of smallest unit of separation is one instruction of Dalvik bytecode  
  - Fine-grained randomization compared with existing method[1]

- Step 3: Code Insertion  
  - Insert respective parts of detection code into extracted method  
  - Fix partial code so as not to break original functionalities  
  - Add virtual registers  
  - Add Exception handling code

4. Evaluation

- Experiment 2. Robustness against static analysis  
  - Evaluating robustness against static signature matching in terms of false positives from viewpoint of attackers  
  - Idea: If original bytecode contains sequence of instructions similar to detection code, an attacker will meet false positives when they try to find detection code

- Runtime overhead is NOT simply proportional to the amount of inserted detection code

- Result  
  - False-positive score (Average number of exact matches)  
  - Proposed method: 15.66  
  - Existing method[1]: 4.392

5. Conclusions and Future work

- Conclusions  
  - Improve robustness against static signature matching  
  - Reducing runtime overhead still remaining

- Future work  
  - Introducing multiple integrity-checking methods  
  - An attacker would dynamically monitor specific API calls, such as getPackageManager(), to extract detection code

- Considering more sophisticated code injection strategy  
  - We have to compete with advanced analysis techniques such as dataflow analysis and program slicing

- Considering other evaluation methodologies  
  - How to evaluate “difficulty of repackaging” quantitatively?

References