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

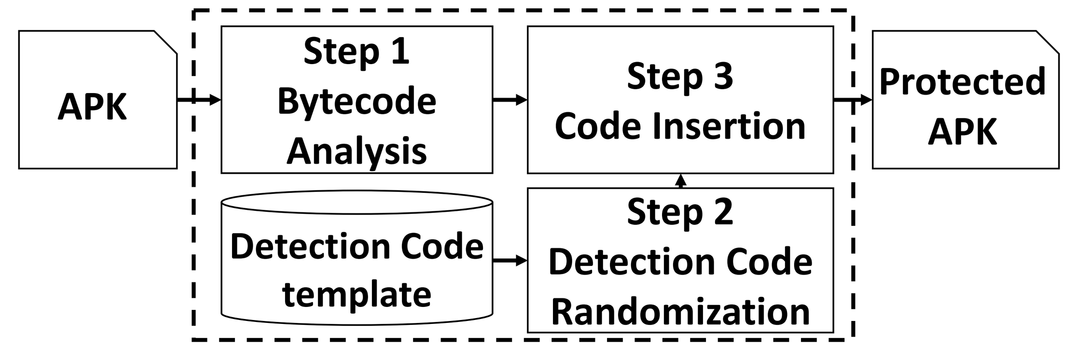
- **Repackaging is a severe problem on Android**
  - 80 % of malware families are create by repackaging
  - Financial loss caused by pirated apps
- **Countermeasures**
  1. Detecting repackaged apps on the market
     - Code-similarity approach
  2. Hardening apps by using tamper-proofing techniques
     - 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

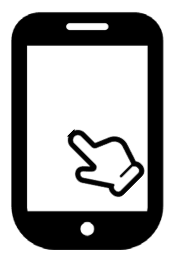
- **Self-protection for Android apps**
  - Verifying integrity of an app
    - Repackaged apps refuse to provide their functionalities to prevent working on user devices
- **Evasion attacks against self-protection mechanism**
  - An attacker uses static and dynamic analysis techniques to locate and disable 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

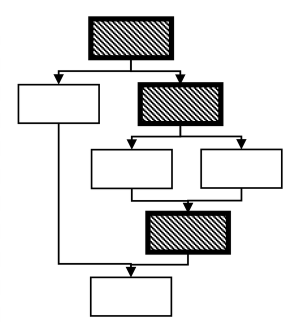


- **Step 1: Bytecode Analysis**
  - ✓ Determine where to inject detection code
    - Extract and analyze methods that are called few times



- Count the number of method calls by a debugging API, `startMethodTracing()`
- Input user event by Monkey tool

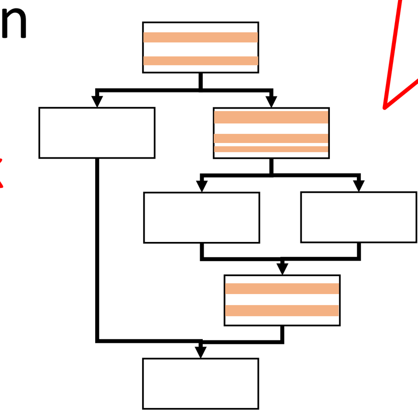
Dynamic Analysis



- Construct control flow graph(CFG)
- Find all dominator nodes of a randomly selected basic block

Static Analysis

- **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



An example of code insertion

- **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]

```
// Get certificate information
PackageManager pm = context.getPackageManager();
String pname = context.getPackageName();
PackageInfo pi = pm.getPackageInfo(pname,
                 PackageManager.GET_SIGNATURES);
Signature signature = pi.signatures[0];
// Calculate hash value of certification
int sigHash = signature.hashCode();
// Compare with hard-coded value
if (sigHash != 283418959) {
    detected();
}
```

An example of predefined detection code
(Simplified for the sake of readability)

```
invoke-virtual {p0}, ... getPackageManager() ...
move-result-object v2
invoke-virtual {p0}, ... getPackageName() ...
move-result-object v1
const/4 v0, 0x0
const/16 v5, 0x40
invoke-virtual {v2, v1, v5}, ... getPackageInfo(...) ...
move-result-object v0
iget-object v5, v0, ... signatures:[ ...
const/4 v6, 0x0
aget-object v3, v5, v6
invoke-virtual {v3}, ... hashCode() ...
move-result v4
const v5, -0x10e4a14f
if-eq v4, v5, :cond_0
invoke-static {}, ... detected() ...
:cond_0 return-void
```

Compiled detection code

Some instructions are not separable
- `invoke` and `move-result`
- `if-XX` and corresponding basic blocks

```
invoke-virtual {p0}, ... getPackageManager() ...
move-result-object v2
invoke-virtual {p0}, ... getPackageName() ...
move-result-object v1
---
const/4 v0, 0x0
---
const/16 v5, 0x40
invoke-virtual {v2, v1, v5}, ... getPackageInfo(...) ... move-result-object v0
---
iget-object v5, v0, ... signatures:[ ...
const/4 v6, 0x0
---
aget-object v3, v5, v6
---
invoke-virtual {v3}, ... hashCode() ...
move-result v4
---
const v5, -0x10e4a14f
if-eq v4, v5, :cond_0
invoke-static {}, ... detected() ...
:cond_0 return-void
```

An example of split detection code

4. Evaluation

- **Experimental Setup**
  - **Data Set**: 27 apps from GooglePlay and F-Droid
  - **Device**: Nexus 5X (physical Device), Android 6.0
- **Experiment 1. Feasibility and Side effects**
  - Dynamically analyze following apps: (1) original, (2) protected, (3) repackaged after protection
    - ✓ No functional difference between original and protected apps
    - ✓ All repackaged apps could not run successfully on the device
    - ✓ 0.1 – 17 % of runtime overhead occurred

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

- **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

Randomized by proposed method:
```
1 : new-instance …
2 : invoke-virtual …
---
3 : iget-object …
4 : invoke-static …
---
5 : sget-object …
---
6 : invoke-virtual …
7 : move-result …
---
8 : invoke-static …
9 : if-neq …
10: invoke-static …
```

Randomized by existing method[1]:
```
1 : new-instance …
2 : invoke-virtual …
3 : iget-object …
4 : invoke-static …
---
5 : sget-object …
6 : invoke-virtual …
7 : move-result …
---
8 : invoke-static …
9 : if-neq …
10: invoke-static …
```

split detection code

Bytecode sequence contained in each separated part are used for signature

- Result

	Proposed method	Existing method[1]
False-positive score (Average number of exact matches)	15.66	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 `getPackageInfo()`, 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
[1] Lannan Luo, Yu Fu, Dinghao Wu, Sencun Zhu and Peng Liu "Repackage-proofing Android Apps," in Proceedings of the International Conference on Dependable Systems and Networks (DSN), 2016.