Tackling runtime-based obfuscation in Android with TIRO

Michelle Wong and David Lie

University of Toronto

Usenix Security 2018
Android malware and analysis

• Mobile devices are a valuable target for malware developers
  ◦ Access to sensitive information and functionality

• Arms race between malware developers and security analyzers
Java obfuscation

- Most Android applications written in Java
- Obfuscation using Java features
  - Reflection
  - Dynamic code loading
  - Native methods

Language-based obfuscation

I do [?], where [?] might be X

Does it do X? Is it malware?

Does it do X?

Is it malware?

Application

DEX Code

JNI

Native Code

Framework APIs

ART/DVM Runtime

Linux

Device
Native obfuscation

• Can avoid runtime entirely by using native code
  ◦ No Java code or invocations to Java methods
• Seems very little malware do this
  ◦ Framework APIs mostly in Java
  ◦ Requires access to undocumented low-level interfaces of system services
Obfuscation via runtime tampering

Language-based obfuscation

ease of use, reliability

difficulty of analysis

Full-native code obfuscation

Runtime-based obfuscation

I do Y and only Y (I mean X)

Not malware! Doesn't do X

Application
DEX Code

Framework APIs

JNI

Application Native Code

ART/DVM Runtime

Linux

Device
Unexpected code behavior

**DEX D:**

```python
...  
class A:  
    method B:  
      <abc>  
...  
<native>
```

I'm loading class A from DEX D

I'm invoking method B from class A

I'm executing instrs <abc> from method B

**ART/DVM Runtime**

```
Actually...  
Loading class E from DEX V
```
```
Actually...  
Invoking method I from class L
```
```
Actually...  
Executing instrs <hac> from method K
```

Java

Unexpected classes  
Unexpected methods  
Unexpected instructions
Android RunTime (ART)

- Investigated how code is loaded and executed within ART

```
DEX D:
...
class A:
    method B:
        <abc>
    ...
```

ART
ART code loading

DEX D:
...

class A:
  method B:
    <abc>
...

1 2 DEX file hooking

ART

java.lang.
DexFile

mCookie

art::
DexFile

DEX file
(mmap)

class E:
  method V:
    <bad>

1 2
ART code loading

DEX D:
...  
class A:
  method B:  
  <abc>  
...  

1 2  DEX file hooking  
3  Bytecode overwriting  

ART

DEX file (mmap)  
3  

class E:  
  method V:  
  <bad>  

mirror::  
Class  

Unexpected classes  
and methods  

art::  
ArtMethod  

T I R O  
9  
U N I V E R S I T Y O F T O R O N T O
ART code execution

DEX D:
...  
class A:
    method B:
      <abc>
...  

4 ArtMethod hooking

Invoke B() in class A  
(inherited from class O)

DEX file (mmap)

class A:
  method B:
    <abc>

mirror::
Class

vtable_ 4

art::
ArtMethod

Unexpected methods
ART code execution

DEX D:
... class A: method B: <abc> ...

ART

DEX file (mmap)
class A: method B: <abc>

mirror:: Class

art:: ArtMethod

Unexpected instructions

4 ArtMethod hooking
5 Method entry-point hooking
6 Instruction hooking/ modification

4 ArtMethod hooking
5 Method entry-point hooking
6 Instruction hooking/ modification

<trampoline>
Runtime state tampering in ART

```
DEX D:
...
class A:
  method B:
    <abc>
...
```

1. DEX file hooking
2. Bytecode overwriting
3. ArtMethod hooking
4. Method entry-point hooking
5. Instruction hooking/modification

ART
Deobfuscation

- Unified framework to handle language-based and runtime-based obfuscation
- Pure static analysis: imprecise, no run-time information to deobfuscate
  - Reflection targets, dynamically loaded code, etc.
- Pure dynamic analysis: lack of code coverage
Targeted execution

Dealing with obfuscation

- Static
- Dynamic
- Inject inputs
- Path constraints
- Obfuscation locations
TIRO: A hybrid iterative deobfuscator

APK file ➔ Target ➔ Observe ➔ dynamic ➔ Run ➔ Instrument ➔ static ➔ deobfuscated application

run-time values, extracted code

instrumented obfuscation locations

security analysis
onCreate() {
    ...
    Method method = klass.getMethod(decrypt("wzjg...");
    method.invoke(receiver, args);
}
onCreate() {
...

7    Method method = klass.getMethod(decrypt("wzjg..."));
8    method.invoke(receiver, args);

Target (Reflection)
onCreate() ➞ ... ➞ Method::invoke()
onCreate() {

    ...

    Method method = klass.getMethod(decrypt("wzjg..."));

    method.invoke(receiver, args);

    Log: refl, onCreate, 8, "foo"

    Instrument

    log(..., method.getName())

    Run

    • Generate inputs from targeting
    • Inject inputs to run obfuscation locations
onCreate() {
    ...
    
    7 Method method = klass.getMethod(decrypt("wzjg..."));

    8 method.invoke(receiver, args);

    Log: refl,onCreate,8,"foo"

    Observe
    oncCreate() ➞ foo()
Handling runtime-based obfuscation

Record original ART state

Check ART state

modifies runtime state
Runtime-based deobfuscation

• Example: Instruction hooking

    onCreate() {

        ...

        7 nativeFoo();

        8 bar();

        ...

    }
Runtime-based deobfuscation

• Example: Instruction hooking

```java
onCreate() {
    nativeFoo();
    bar();
}
```

Instrument (ART runtime)

Run
Log: onCreate, 7, bar[code_item], xyz
Extracted DEX: <xyz>

Observe
onCreate() → method_xyz()
Iterative deobfuscation

- Example: 2nd iteration

```java
onCreate() {
    ...
    7 nativeFoo();
    8 bar();
    ...
}
```

```java
method_xyz() {
    ...
    11 Method method =
    klass.getMethod(decode("vbs"));
    12 method.invoke(receiver, args);
    ...
}
```

Target (Reflection)
Implementation

- Static: Soot framework\textsuperscript{2} for analysis and instrumentation

- Dynamic:
  - Modified AOSP with instrumented ART runtime
    - Android 4.4, 5.0, 6.0
  - Monitoring process to parse deobfuscation log and extract bytecode

Evaluation

• Ability to detect and deobfuscate techniques in modern Android malware

• Investigate use of language-based and runtime-based obfuscation in malware

• Deobfuscation performance (in paper)
**TIRO: Detection and deobfuscation**

- Labeled obfuscated samples, categorized by obfuscator/packer

<table>
<thead>
<tr>
<th>Language-based</th>
<th>Runtime-based</th>
<th>TIRO</th>
<th>Sensitive APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reflection</td>
<td>Dynamic loading</td>
<td>Native methods</td>
<td>DEX file hooking</td>
</tr>
<tr>
<td>aliprotect</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>baiduprotect</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>dexprotector</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>ijiampacker</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>naga_pha</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>qihoopacker</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>sechshell</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

100% 53% 2.3 +30
Obfuscation usage in malware

- Obfuscated malware samples from VirusTotal

<table>
<thead>
<tr>
<th>Method</th>
<th>Language-based</th>
<th>Runtime-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reflection</td>
<td>58.5%</td>
<td>DEX file hooking</td>
</tr>
<tr>
<td>Dynamic loading</td>
<td>79.9%</td>
<td>Class data overwriting</td>
</tr>
<tr>
<td>Direct invocation</td>
<td>52.2%</td>
<td>ArtMethod hooking</td>
</tr>
<tr>
<td>Reflected invocation</td>
<td>0.1%</td>
<td>Method entry-point hooking</td>
</tr>
<tr>
<td>Native invocation</td>
<td>49.2%</td>
<td>Instruction hooking</td>
</tr>
<tr>
<td>Native methods</td>
<td>96.8%</td>
<td>Instruction overwriting</td>
</tr>
</tbody>
</table>
Conclusion

- New category of obfuscation techniques in Android: runtime-based obfuscation

- **Tiro**: A hybrid iterative deobfuscation framework
  - Handles both language-based and runtime-based techniques
  - Deobfuscates modern malware and uncovers sensitive behaviors

- 80% of samples from VirusTotal dataset use runtime-based obfuscation