Tackling Android’s Native Library Malware with Robust, Efficient and Accurate Similarity Measures

Anatoli Kalysch (Speaker), Oskar Milisterfer, Mykolai Protsenko, Tilo Müller
August 29, 2018
Friedrich-Alexander-Universität Erlangen-Nürnberg
Department of Computer Science
IT Security Infrastructures Lab
Software Security Research Group
Outline

Android Malware is Going Native
  Android Obfuscation in Context
  Why Native Libraries?

Introducing Dimensional Encoding
  Centroids
  Comparison Procedure

Bringing it All Together
  Accurate, Efficient, and Robust?
  Hunting Malware

Conclusion
### Obfuscation on Android

<table>
<thead>
<tr>
<th>Year</th>
<th>Obfuscation Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>Symmetric Encryption (partly custom) String Encoding</td>
</tr>
<tr>
<td>2012</td>
<td>Proguard, Steganography, Dalvik Level Encryption</td>
</tr>
<tr>
<td>2013</td>
<td>Protector (Dexguard), Non-dalvik Encryption</td>
</tr>
<tr>
<td>2014</td>
<td>Packers, Protectors, and Native Code</td>
</tr>
<tr>
<td>2015</td>
<td>Packers, Protectors, and Native Code comb. w/ Obfuscators</td>
</tr>
</tbody>
</table>

**Table:** Malware obfuscation chronology (excerpt) [4].
## Solutions for your obfuscation needs

<table>
<thead>
<tr>
<th>Packer/Protector</th>
<th>Obfuscation Techniques</th>
<th>Native Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dexguard</td>
<td>obfuscation, hooking, anti-dynamic</td>
<td>✓</td>
</tr>
<tr>
<td>Aliprotect</td>
<td>native and dex obf.</td>
<td>✓</td>
</tr>
<tr>
<td>Tencent</td>
<td>native and dex obf., MLU</td>
<td>✓</td>
</tr>
<tr>
<td>Qihoo</td>
<td>native and dex obf., MLU</td>
<td>✓</td>
</tr>
<tr>
<td>Bangcle</td>
<td>native and dex obf., hooking, MLU</td>
<td>✓</td>
</tr>
<tr>
<td>Ijiami</td>
<td>native and dex obf., MLU</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table: Protection Measures in Packers and Protectors (excerpt) [3].
What makes native code so popular?

- Written in C/C++ and compiled, meaning no smali byte code is available.
- Direct usage of system resources (permission model still applies) and ability to manipulate own process components.
- Breaks most Android reverse engineering tools, and less meta data is available compared to smali byte code making reverse engineering harder.
Tackling malicious native libraries on Android

- Need for a solution to the threat posed by malicious native libraries. And ideally this solution is
  - automated,
  - accurate,
  - efficient, and
  - robust (regarding code obfuscation).

- Currently we see wide employment of code-similarity measures to detect known malicious code, e.g., hash and signature-based solutions.
Outline

Android Malware is Going Native
    Android Obfuscation in Context
    Why Native Libraries?

Introducing Dimensional Encoding
    Centroids
    Comparison Procedure

Bringing it All Together
    Accurate, Efficient, and Robust?
    Hunting Malware

Conclusion
Process Overview

- Create a 3D vector for every function in the native library based on the control-flow graph (CFG)
- From the 3D vectors we create a centroid from the sum of its edge weights
- The centroids only differ if the underlying functions differ as well
- This encoding introduces an abstraction layer that disregards certain obfuscation techniques
Creation of a 3D vector

Each basic block (BB) in the CFGs is given a coordinate in the three dimensions

- sequence,
  - defining the order in which basic blocks (BB) of the CFG are executed
- selection, and
  - represents the number of outgoing edges for each BB
- repetition.
  - reflecting the loop depth of the current basic block

After all the BBs were assigned coordinates in the 3D system a unifying vector can be created.
Creating Centroids

A Centroid of a 3D-CFG vector is defined as

$$\vec{c} = \langle c_x, c_y, c_z, \pi \rangle,$$

with

$$c_x = \frac{\sum_{e(p,q) \in 3D-CFG} (\pi p x_p + \pi q x_q)}{\pi},$$

and $c_y$ and $c_z$ accordingly [1].

The $\pi$ coordinate is encoded as $\pi = \sum_{e(p,q) \in 3D-CFG} (\pi p + \pi q)$ where $e(p, q)$ refers to an edge in the 3D-CFG, which connects the two nodes $p$ and $q$. 
Comparison

- Due to monotonicity properties of centroids [2] the same methods will be mapped to the same centroid
- Centroids are sortable [2], enabling a faster comparison
- Comparison of two centroids is performed through the computation of the Centroid Difference Degree (CDD)

**Definition (Centroid Difference Degree)**

Given two centroids, \( \vec{c} \) and \( \vec{d} \), the CDD is computed as

\[
CDD(\vec{c}, \vec{d}) = \max\left(\frac{|c_x - d_x|}{c_x + d_x}, \frac{|c_y - d_y|}{c_y + d_y}, \frac{|c_z - d_z|}{c_z + d_z}, \frac{\pi_c - \pi_d}{\pi_c + \pi_d}\right).
\]
Putting Centroids to Work on ARM Libraries

- Before application to ARM the right combination of variables and weights needs to be found
- Heuristics for a sane CDD need to be found / defined
- Equally, a Library Similarity Degree (LSD) needs to be defined
Outline

Android Malware is Going Native
  Android Obfuscation in Context
  Why Native Libraries?

Introducing Dimensional Encoding
  Centroids
  Comparison Procedure

Bringing it All Together
  Accurate, Efficient, and Robust?
  Hunting Malware

Conclusion
The Dataset and Malware Families

- 3rd party APK Stores
  - 18 different app stores
  - 508,745 apps
  - 2,346,005,582 methods
- 29 Malware-Families including
  - Bios.A
  - DroidDream
  - Godless
  - KungFu
  - OldBoot
  - Rootnik
  - TatooHack
  - VikingHorde
  - Ycchar
Accuracy

- Detection of library versions
  - comparisons of 1,500 unrelated library pairs
  - testing different pairs of CDD / LSD yielded false positive rates (FPR) as low as 1% for libraries with more than 100 functions
  - significantly small libraries with less than 100 functions performed worse with FPR around 10%

- Database clustering
  - 146,264 native libraries from 40 size-based clusters were categorized into 4,201 clusters
  - A name-based library comparison and in some cases a method level CFG comparison concluded FPRs of less than 2%
Efficiency - Computation

Figure: Computation of a centroid with and without database access.
Efficiency - Comparison

Figure: Comparison between related and unrelated libraries.
## Robustness to Obfuscation

<table>
<thead>
<tr>
<th>Obfuscation Technique</th>
<th>Category</th>
<th>Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>modified APK meta data</td>
<td>string-based</td>
<td>✓</td>
</tr>
<tr>
<td>native library relocation</td>
<td>file hiding</td>
<td>✓</td>
</tr>
<tr>
<td>native library renaming</td>
<td>string-based</td>
<td>✓</td>
</tr>
<tr>
<td>variable name obfuscation</td>
<td>string-based</td>
<td>✓</td>
</tr>
<tr>
<td>binary stripping</td>
<td>string-based</td>
<td>✓</td>
</tr>
<tr>
<td>native library payload placement</td>
<td>code insertion</td>
<td>✓</td>
</tr>
<tr>
<td>junk function insertion</td>
<td>code insertion</td>
<td>✓</td>
</tr>
<tr>
<td>literal/arithmetic encoding</td>
<td>code insertion</td>
<td>✓</td>
</tr>
<tr>
<td>BB segment reordering</td>
<td>control flow obfuscation</td>
<td>✓</td>
</tr>
<tr>
<td>opaque predicates</td>
<td>control flow obfuscation</td>
<td>×</td>
</tr>
<tr>
<td>function in/outlining</td>
<td>control flow obfuscation</td>
<td>×</td>
</tr>
<tr>
<td>control flow flattening</td>
<td>control flow obfuscation</td>
<td>×</td>
</tr>
</tbody>
</table>
### Findings

<table>
<thead>
<tr>
<th>Market Name</th>
<th>Malicious NLs</th>
<th>Detected NLs</th>
</tr>
</thead>
<tbody>
<tr>
<td>playmob</td>
<td>26</td>
<td>1089</td>
</tr>
<tr>
<td>mumayi</td>
<td>36</td>
<td>151</td>
</tr>
<tr>
<td>baidu</td>
<td>44</td>
<td>368</td>
</tr>
<tr>
<td>apkmirror</td>
<td>80</td>
<td>3393</td>
</tr>
<tr>
<td>nduo</td>
<td>128</td>
<td>396</td>
</tr>
<tr>
<td>up2down</td>
<td>219</td>
<td>4195</td>
</tr>
<tr>
<td>apkworld</td>
<td>307</td>
<td>1880</td>
</tr>
</tbody>
</table>

**Table:** Selection of detected malicious native libraries among ARM 32-bit native libraries.
Comparison to VirusTotal

- APKs from detected malicious clusters were uploaded to VirusTotal
  - Roughly half were detected as malicious

- Next we extracted the native library and uploaded it to VirusTotal as well
  - Note that we analyzed malware that actively uses native code for exploitation
  - less than 4% were considered malicious
Outline

Android Malware is Going Native
  Android Obfuscation in Context
  Why Native Libraries?

Introducing Dimensional Encoding
  Centroids
  Comparison Procedure

Bringing it All Together
  Accurate, Efficient, and Robust?
  Hunting Malware

Conclusion
Conclusion

- Improved version of the centroid similarity measure
  - Defined heuristics to use with ARM libraries
  - Increased efficiency and accuracy
  - Robustness against certain obfuscation techniques

- Large-scale study of native library malware in Android third party apps
  - 18 third party app stores checked for infection
  - 508,745 apps analyzed
  - Infection rates of up to 17.05% detected
  - Detection rates outperform VirusTotal
Thank you.

Questions?
References


