Analysis of Code Heterogeneity for High-precision Classification of Repackaged Malware

Ke Tian, Daphne Yao, Barbara Ryder
Department of Computer Science
Virginia Tech

Gang Tan
Department of Computer Science and Engineering
Penn State University
✓ Motivation:

Repackaged malware skews machine learning results

✓ Solution:

Partition + Machine learning classification

✓ Experiment:

30-fold improvement in False Negative than non-partition ML-approach!
Repackaged Malware

Android Malware writers are repackaging legitimate (popular) apps with malicious payload[1].

Conventional Machine Learning for Malware Classification

A huge Dataset

Extract Feature vectors

DroidAPIMiner: Sensitive APIs
Drebin: APIs, constant strings, URLs
Peng et al.: Permission
Is machine learning taking over the world?

No – What the specific challenges and solutions?
<table>
<thead>
<tr>
<th>Motivation</th>
<th>Code Heterogeneity</th>
<th>Challenges</th>
</tr>
</thead>
</table>

**Heterogeneous Code:**
Code with different Security behaviors in different code portions

Existing machine learning techniques extracts features from the entire app, repackaged malware skews classification results (i.e., introduce false negatives)

**Research Question:** How to recognize heterogeneity in code?
Motivation

- Code Heterogeneity
- Challenges

**Challenges**
- How to partition the code?
- How to extract efficient features?
- How to calculate the malware score?

**Ours:**

Tasks:
- How to partition the code?
- How to extract efficient features?
- How to calculate the malware score?
First Attempt: partition based on direct method call relations
First Attempt: not wok well

Missed implicit dependence relations!
2-level graph

Class-level Dependence Graph (CDG) to capture event (activity) relations.

Method-level Call Graph (MCG) for subsequent feature extraction.
Class-level Dependence Graph (CDG)

Inferring from static analysis

✓ Class-level call dependence.
✓ Class-level data dependence
✓ Class-level ICC dependence.

Solution

<table>
<thead>
<tr>
<th>Graph Generation</th>
<th>Partition &amp; Mapping</th>
<th>Feature Extraction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Class A
  - invoke
  - startActivity (explicit ICC)

- Class B
  - invoke

- Class C
  - invoke

- Class D
  - iget

- Class E
  - invoke

- Class F
  - iget

- DATA

- ICC
So far, we got code partitioned at class-level dependence graph

*Can feature extraction be done on class-level call graph?*

No. Why?

Class-level call graph is too coarse-grained, lacking useful method information. Need method-level details
Mapping Through Projection (to prepare for feature extraction)

Aggregation features in each region

| f1 | 3*f2 | f3 | f4 | ...
|----|------|----|----|-----|

Class D

Dependence Region 1

Class F

a

b

c

f1 f2

f2 f3

f2 f4
Feature Extraction for Regions

✓ Type I: User Interaction Features
  * user-related functions and the graph-related impact features

✓ Type II: Sensitive API Features.
  * sensitive Java and Android APIs

✓ Type III: Permission Request Features.
  * permissions used in each region

1. Features are used to profile the region’s behaviors.
2. Combined with traditional features, user interaction and graph properties
Classification of Apps

- Binary Classification for each dependence region.
- Computing the malware score for an app based on results from all regions.

\[ r_m = \frac{N_{mali}}{N_{total}} \]

Malware score \( r_m \)

Malicious regions

Total regions in the app

Continuous value in \([0,1]\)
Solution summaries:

- **(Partition)** Partition the app into different Regions → Class-level Dependence Graph (CDG)
- **(Feature)** Independently classify each Region → Method-Level Call Graph (MCG)
- **(Classification)** Mapping the features through projection, calculating Malware Score

Limitations :

- Graph Accuracy. -- More accurate program analysis
- Dynamic Code -- Native Libraries
- Integrated Malware – Hard to partition
Classification of non-repackaged Apps

- Each of apps contains just a single region (dependence region).
- The region is labeled as benign or malicious from dataset
- Used to evaluate the features and get trained classifiers
Classification of non-repackaged Apps

<table>
<thead>
<tr>
<th>Cases</th>
<th>FNR(%)</th>
<th>FPR(%)</th>
<th>ACC(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>KNN</td>
<td>6.43 ± 5.22</td>
<td>6.50 ± 2.67</td>
<td>93.54 ± 3.33</td>
</tr>
<tr>
<td>D.Tree</td>
<td>4.78 ± 2.90</td>
<td>3.52 ± 1.57</td>
<td>95.79 ± 2.14</td>
</tr>
<tr>
<td>R.Forest</td>
<td>3.85 ± 3.27</td>
<td>1.33 ± 0.78</td>
<td>97.30 ± 1.96</td>
</tr>
<tr>
<td>SVM</td>
<td>7.42 ± 4.85</td>
<td>1.46 ± 0.58</td>
<td>95.28 ± 2.58</td>
</tr>
</tbody>
</table>

Random Forest performs Best

All three types of features are effective

Use Random Forest as the standard classifier to test repackaged malware
Experiment
Partition & Mapping
Feature Extraction
Graph Generation

Use the Same trained Random Forest to test

Test three repackaged malware families:
1. Geinimi
2. Kungfu
3. AnserverBot

Comparison:
1. Entire-app classification (Basic)
2. Our partition classification

Our FNR gets 30-fold improvement than the non-partition!
### Case Study of Heterogeneous Properties

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Partition (ours)</th>
<th>Non-partition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DRegion1</td>
<td>DRegion2</td>
</tr>
<tr>
<td><strong>Type III</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ_PHONE_STATE permission</td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>READ_LOGS permission</td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Type II</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getDeviceId function</td>
<td>in Android/telephone/telephoneManager</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>read function in</td>
<td>Java/io/InputStream</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>write function in</td>
<td>Java/io/FileOutput</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Type I</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>onClick function occurrence</td>
<td></td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td># of distinct user-interaction</td>
<td>functions</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>onKeyDown function occurrence</td>
<td></td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Classification</th>
<th>Benign</th>
<th>Malicious</th>
<th>Benign</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctness</td>
<td>✔️(Yes)</td>
<td>✗(No)</td>
<td></td>
</tr>
</tbody>
</table>

- Malicious Region with sensitive permissions & APIs
- Benign Region with user-interaction functions

**Need to look into the code structure!**
Region analysis in popular apps

- Analyzing 1,617 free popular apps from Google Play.
- $158/1,617 = 9.7\%$ Apps contain multiple regions
- Ad Libraries introduce multiple regions in Apps.
- Some aggressive ads libraries introduce alerts in the detection.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Non-repackaged app</th>
<th>Repackaged app</th>
<th>Ads Library</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>% of Alerts</strong></td>
<td>w/o Ads</td>
<td>w/ Group 1 Ads</td>
<td>w/ Group 2 Ads</td>
</tr>
<tr>
<td>% of Alerts</td>
<td>2.96%</td>
<td>2.96%</td>
<td>5.18%</td>
</tr>
</tbody>
</table>

*Table. Alerts made by Group 2 Ads library (Group 1: admob | Group 2: adlantis)*
False Negatives:

1) Integrated benign and malicious behaviors.
2) Not enough malicious behaviors in malicious components

False Positives:

Some aggressive packages and libraries, e.g., Adlantis, results in a false alarm in our detection.
Conclusions:

- Our approach achieves 30-fold improvement than the non-partition-based approach.
- Our approach is able to identify malicious code in repackaged malware.
- Partition can be used to label malicious code or Isolate inserted code (Ads packages or dead code)

Future work:

More Effort on Partition/Detection for Code Provenance!
Thanks!