MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models

Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, Gianluca Stringhini.

NDSS 2017, 28-02-2017
Motivation: Android & Malware

• Android market share is growing
  – In 2016, 85% of smartphone sales

• At the same pace the interest by cybercriminals is growing
  – Bypassing two-factor authentication
  – Stealing sensitive information, etc.
Motivations: Current Defenses

• Can’t use complex on-device operations
  – Limited battery and memory resources

• Google’s centralized analysis
  – Previous work shown a few incidents
  – Users buy apps from third party markets

• Lots of research in the field! However
  – Permission-based models prone to false positive
  – Relying on API calls frequently used by malware
    needs constant, costly retraining
Motivations: Our Idea

Intuition: malware uses calls for different actions and in different order than benign apps

- E.g. android.media.MediaRecorder used by any app with permission to record audio
- Only using it after calls to getRunningTasks(), which allows to record conversations, may suggest maliciousness

Rely on the sequence of abstracted calls

1. Sequence captures the behavioral model
2. Abstraction provides resilience to API changes
Overview

1. Call Graph Extraction
2. Sequence Extraction
3. Markov Chain Modeling
4. Classification
Call Graph Extraction

• Based on static analysis
  – Given an apk, extract call graphs

• Tools
  – Soot (Java optimization and analysis framework)
  – FlowDroid
Call Graph

com.fa.c.RootCommandExecutor: Execute()

android.util.Log: d()

java.lang.Throwable: getMessage()


com.stericson.RootShell.execution.Shell: add()
Sequence Extraction

• Soot gives the call graph from which we extract the sequence of functions that are potentially called by the program, but...

• When running example multiple times...
  – Execute() may be followed by different calls, e.g., getShell() only in try or getShell() + getMessage() in catch
Sequence Extraction

• **We proceed as follows:**
  1. Identify set of entry nodes
  2. Enumerate paths
  3. Output set of all paths as the sequences of API calls

• **But we said we were using abstracted calls!**
Abstraction

Package

android.text.style.CharacterStyle: void <init>()

Family

Package

android.os.Bundle: void <init>()

Package

java.lang.throwable: String getMessage()
Abstraction

• Packages
  – Using the list of 243 packages (as of API level 24) + 95 from the Google API
  – Packages defined by developers → “self-defined”
  – If we can’t tell what its class implements → “obfuscated”

• Families
  – 9 families: android, google, java, javax, xml, apache, junit, json, dom
  – Plus self-defined and obfuscated
Example

com.f.a.c.RootCommandExecutor:
  Execute()
  [self-defined, self-defined]

com.stericson.RootTools.RootTools:
  getShell()
  [self-defined, self-defined]

com.stericson.RootShell.
  execution.Shell: add()
  [self-defined, self-defined]

com.f.a.c.RootCommandExecutor:
  Execute()
  [self-defined, self-defined]

android.util.Log:
  d()
  [android.util, android]

com.f.a.c.RootCommandExecutor:
  Execute()
  [self-defined, self-defined]

java.lang.Throwable:
  getMessage()
  [java.lang, java]

com.stericson.RootShell.execution.Shell: add()
Overview

1. Call Graph Extraction (1)
2. Sequence Extraction (2)
3. Markov Chain Modeling (3)
4. Classification (4)
Markov Chain

• Memoryless models
  – Probability of transition from a state to another only depends on the current state

• Represented as a set of nodes
  – Each corresponding to a different state, and a set of edges labeled with the probability of transition.

• Sum of all probabilities associated to all edges from any node is exactly 1
Building the Markov Chains

- Nodes / States
- Features set
- Edges / Transition Probabilities
- Sequence of abstracted API calls
Example

```
com.f.a.c.RootCommandExecutor: Execute()
  [self-defined, self-defined]

  [self-defined, self-defined]

com.stericson.RootShell. execution.Shell: add()
  [self-defined, self-defined]
```

```
com.f.a.c.RootCommandExecutor: Execute()
  [self-defined, self-defined]

android.util.Log: d()
  [android.util, android]

java.lang.Throwable: getMessage()
  [java.lang, java]
```

```
0.5

Self-defined

0.25  0.25

Java  Android
```
Feature Extraction

• For each app:
  – Feature vector = probabilities of transition from one state to another in the Markov chain
  – With families, 11 possible states $\rightarrow$ 121 possible transitions in each chain
  – With packages, 340 states $\rightarrow$ 115,600 transitions
Overview

1. Call Graph Extraction
2. Sequence Extraction
3. Markov Chain Modeling
4. Classification

Diagram showing the flow from Call Graph Extraction to Classification.
Classification

• Build a classifier using the extracted features
  – Each app labeled as benign or malware

• We tested our idea using:
  – Random Forests
  – 1-NN, 3-NN
  – SVM

• SVM was less efficient than the other systems
Dataset
## Datasets

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Date Range</th>
<th># Samples</th>
<th># Samples (API Calls)</th>
<th># Samples (Call Graph)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benign</td>
<td>OldBenign</td>
<td>Apr 13 – Nov 13</td>
<td>5,879</td>
<td>5,837</td>
<td>5,572</td>
</tr>
<tr>
<td></td>
<td>NewBenign</td>
<td>Mar 16 – Mar 16</td>
<td>2,568</td>
<td>2,565</td>
<td>2,465</td>
</tr>
<tr>
<td></td>
<td><strong>Total Benign</strong></td>
<td></td>
<td><strong>8,447</strong></td>
<td><strong>8,402</strong></td>
<td><strong>8,037</strong></td>
</tr>
<tr>
<td>Malicious</td>
<td>Drebin</td>
<td>Oct 10 – Aug 12</td>
<td>5,560</td>
<td>5,546</td>
<td>5,538</td>
</tr>
<tr>
<td></td>
<td>2013</td>
<td>Jan 13 – Jun 13</td>
<td>6,228</td>
<td>6,146</td>
<td>6,123</td>
</tr>
<tr>
<td></td>
<td>2014</td>
<td>Jun 13 – Mar 14</td>
<td>15,417</td>
<td>14,866</td>
<td>14,827</td>
</tr>
<tr>
<td></td>
<td>2015</td>
<td>Jan 15 – Jun 15</td>
<td>5,314</td>
<td>5,161</td>
<td>4,725</td>
</tr>
<tr>
<td></td>
<td>2016</td>
<td>Jan 16 – May 16</td>
<td>2,974</td>
<td>2,802</td>
<td>2,657</td>
</tr>
<tr>
<td></td>
<td><strong>Total Malicious</strong></td>
<td></td>
<td><strong>35,493</strong></td>
<td><strong>34,521</strong></td>
<td><strong>33,870</strong></td>
</tr>
</tbody>
</table>
How many API calls?

![API Call Distribution CDF](image)

- 2013
- 2014
- 2015
- 2016
- drebin
- newbenign
- oldbenign
Evaluation
Evaluation

• Accuracy of classification on benign and malicious samples developed around the same time

• Robustness to the evolution of malware as well as of the Android framework (using older datasets for training and newer ones for testing and vice-versa)
Training on older samples

Families abstraction
MaMaDroid Vs DroidAPIMiner

DroidAPIMiner is the previous work operating detection of malware samples from benign ones based on sequences of API calls.

<table>
<thead>
<tr>
<th>Tests</th>
<th>DroidAPIMiner</th>
<th>MaMaDroid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same Year</td>
<td>0.56</td>
<td>0.96</td>
</tr>
<tr>
<td>Older samples Training</td>
<td>0.42</td>
<td>0.68</td>
</tr>
<tr>
<td>Newer samples Training</td>
<td>0.50</td>
<td>0.96</td>
</tr>
</tbody>
</table>
Discussion and Limitations
Case Studies (2016/newbenign)

- **False Positives (164 samples)**
  - Most of them “dangerous permissions”
  - E.g., SMS permissions not clear why requested

- **False Negatives (114 samples)**
  - Actually not classified as malware by VirusTotal, might be legitimate
  - Most of them adware
Evasion

• Repackaging benign apps
  – Difficult to embed malicious code while keeping similar Markov chain, vice versa is also hard

• Imitating Markov chains
  – Likely ineffective

• Obfuscation/Mangling
  – Still captured by the [obfuscated] abstraction

• More in the paper…
Limitations

• Classification is memory hungry
• Soot is buggy, we lose ~4% of the samples
• Limits of static analysis only methods
Future Work

• **Further investigate resilience to evasion**
  – Focus on repackaged malicious apps
  – Injection of API calls to mess with Markov chains

• **Enhancements**
  – Fine-grained abstractions (e.g., class)
  – Seed with dynamic analysis

• **Releasing**
  – MaMaDroid’s python code
  – The list of used samples and their hashes
  – Parsed dataset
Conclusions

• We created MaMaDroid, a system for android malware detection

• Static analysis only, based on Markov Chain modeling of sequences of API calls

• Up to 0.99 F-measure in tests, resilient to changes over time

Enrico Mariconti

Thanks for listening