

## Malware Analysis for Android Apps (Guest Lecture)

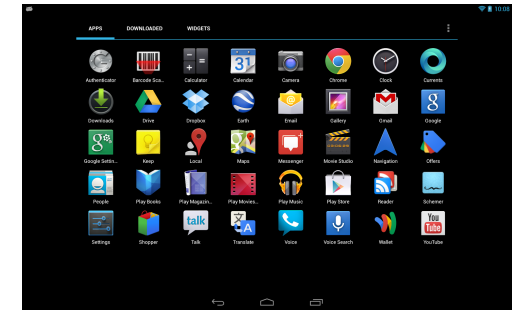
Speaker: Wei Chen  
LFCS, School of Informatics  
University of Edinburgh, UK

April 2, 2018

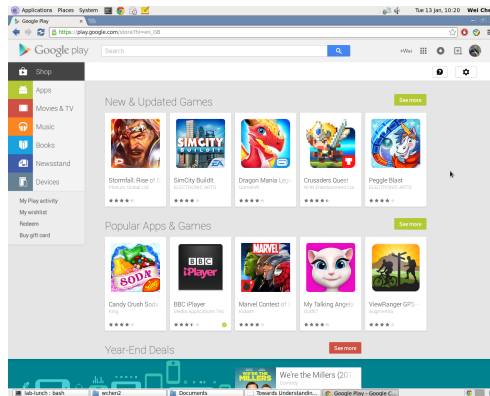
### Outline

- 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

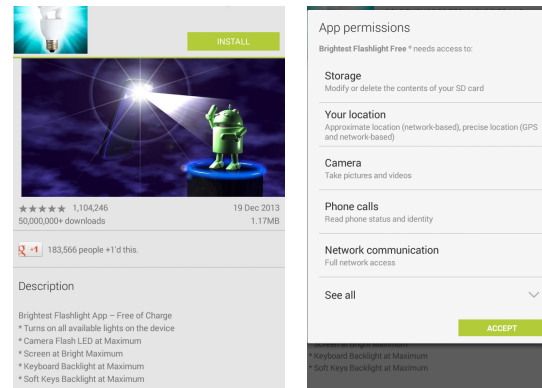
### Android apps and markets



### Android apps and markets



### Example: Flashlight

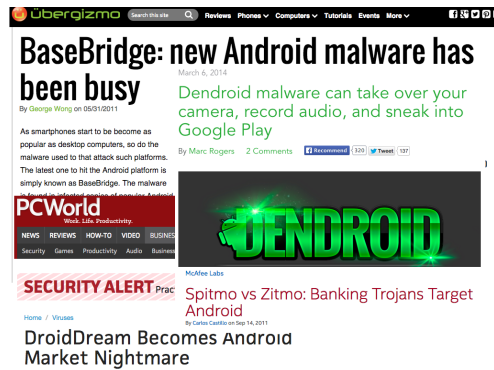


### Example: Flashlight

*"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."* 😊

## Android malware & families



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

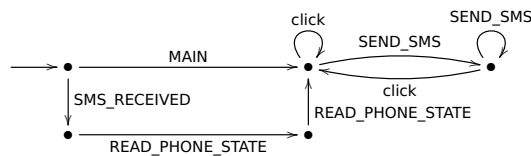
## Android Architecture



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:

```
<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>
```

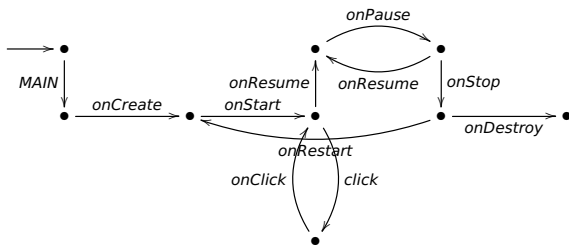
## Example: construct behavioural models

Activity:

```
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; }}}}
```

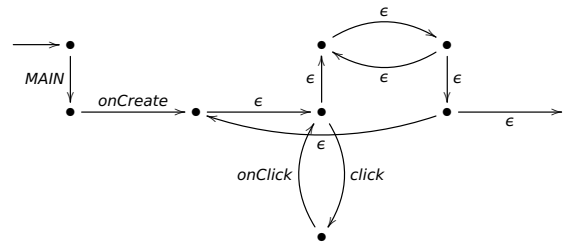
### Example: construct behavioural models

Activity's lifecycle:



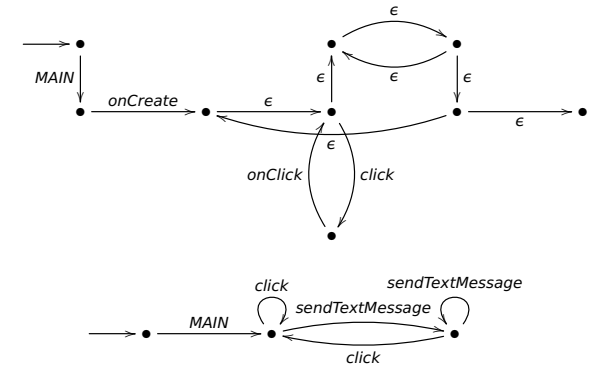
### Example: construct behavioural models

Activity's lifecycle:



### Example: construct behavioural models

Activity's behaviour automaton:

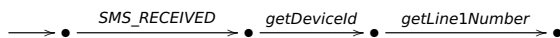


### Example: construct behavioural models

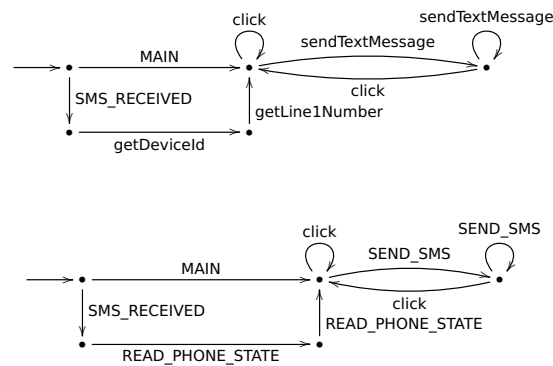
Receiver:

```

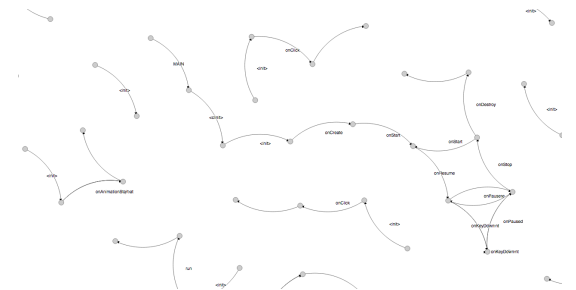
public class Receiver extends BroadcastReceiver {
    public void onReceive(Context context, Intent intent) {
        Intent intent = new Intent();
        intent.setAction("com.main.intent");
        TelephonyManager tm = (TelephonyManager)
        getBaseContext().getSystemService(Context.TELEPHONY_SERVICE);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



### Example: construct behavioural models

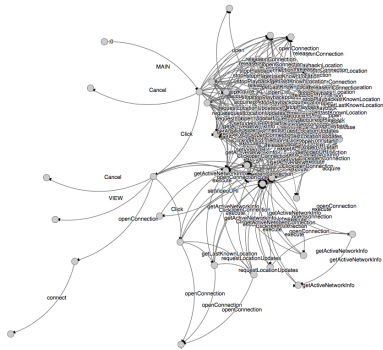


### Example: Flashlight



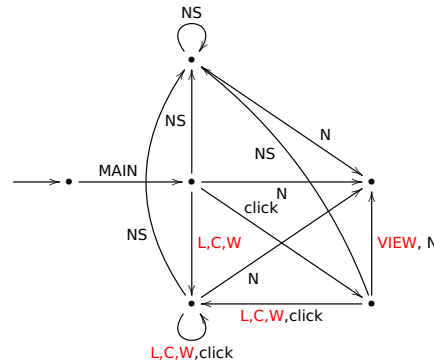
The extended call graphs for Flashlight only considering inter-procedural calls.

## 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

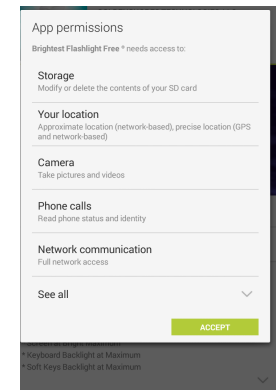
Training (2011-13)	Validation (2011-13)		Testing (2014)	
	precision	recall	precision	recall
permissions	89%	99%	55%	23%
apis	93%	98%	62%	13%
<b>all</b>	<b>95%</b>	<b>98%</b>	<b>65%</b>	<b>15%</b>

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.

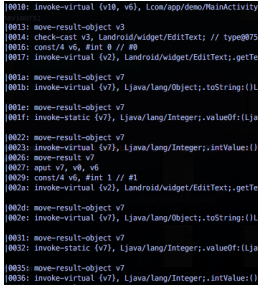
## Syntax-Based Features — Permissions

- ▶ over 200 permissions;
- ▶ coarse and lightweight;
- ▶ good on validation but poor on testing (new malware): precision 89%  $\rightsquigarrow$  55%, recall 99%  $\rightsquigarrow$  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%  $\rightsquigarrow$  62%, recall 98%  $\rightsquigarrow$  13%;
- ▶ **API calls might appear in dead code and most of them are trivial;**
- ▶ **malicious behaviour appears in order.**



```

0018: invoke-virtual {v18, v6}, Lcom/app/demo/MainActivity;
0019: move-result-object v3
001a: check-cast v3, Landroid/widget/EditText; // type=0F5
001b: const/4 v6, #int 0 // #0
001c: invoke-virtual {v2, Landroid/widget/EditText;}, getTe
001d: move-result-object v7
001e: invoke-virtual {v7, Ljava/lang/Object;}, toString()IL
001f: move-result-object v7
0020: move-static {v7}, Ljava/lang/Integer;.valueOf()Lja
0021: move-result-object v7
0022: invoke-virtual {v7, Ljava/lang/Integer;.intValue()IL
0023: move-result v7
0024: aput v7, v6, v6
0025: const/4 v6, #int 1 // #1
0026: invoke-virtual {v2, Landroid/widget/EditText;}, getTe
0027: move-result-object v7
0028: invoke-virtual {v7, Ljava/lang/Object;}, toString()IL
0029: move-result-object v7
002a: invoke-static {v7, Ljava/lang/Integer;.valueOf()Lja
002b: move-result-object v7
002c: invoke-virtual {v7, Ljava/lang/Integer;.intValue()IL
002d: move-result-object v7
002e: invoke-static {v7, Ljava/lang/Integer;.intValue()IL
002f: move-result-object v7
0030: invoke-virtual {v7, Ljava/lang/Integer;.intValue()IL
0031: move-result-object v7
0032: invoke-static {v7, Ljava/lang/Integer;.intValue()IL
0033: move-result-object v7
0034: invoke-virtual {v7, Ljava/lang/Integer;.intValue()IL
0035: move-result-object v7
0036: invoke-static {v7, Ljava/lang/Integer;.intValue()IL

```

- ## Syntax-Based Features — API Calls
- ▶ over 50,000 APIs;
  - ▶ precise and lightweight;
  - ▶ good on validation but poor on testing (new malware): precision 93%  $\rightsquigarrow$  62%, recall 98%  $\rightsquigarrow$  13%;
  - ▶ **API calls might appear in dead code and most of them are trivial;**
  - ▶ **malicious behaviour appears in order.**
- ```
0010: invoke-virtual {v10, v6}, Lcom/app/demo/MainActivity;
0011: move-result-object v3
0012: check-cast v3, Landroid/widget/EditText; // type@0015
0013: const/4 v5, #int 0 // #0
0017: invoke-virtual {v2, Landroid/widget/EditText; getter
0018: move-result-object v7
001b: invoke-virtual {v7, Ljava/lang/Object; toString()IL
001c: move-result-object v7
001f: invoke-static {v7, Ljava/lang/Integer; valueOf(Lj
0020: move-result-object v7
0023: invoke-virtual {v9, Ljava/lang/Integer; intValue()IL
0026: move-result v7
0027: aput v7, v6, v6
0029: const/4 v5, #int 1 // #1
002a: invoke-virtual {v2, Landroid/widget/EditText; getter
002d: move-result-object v7
002e: invoke-virtual {v7, Ljava/lang/Object; toString()IL
0031: move-result-object v7
0032: invoke-static {v7, Ljava/lang/Integer; valueOf(Lj
0035: move-result-object v7
0036: invoke-virtual {v7, Ljava/lang/Integer; intValue()IL
```

## Syntax-Based Features — API Calls

- ▶ over 50,000 APIs;
- ▶ precise and lightweight;
- ▶ good on validation but poor on testing (new malware): precision 93%  $\rightsquigarrow$  62%, recall 98%  $\rightsquigarrow$  13%;
- ▶ **API calls might appear in dead code and most of them are trivial;**
- ▶ **malicious behaviour appears in order.**

```
0010: invoke-virtual {v10, v6}, Lcom/app/demo/MainActivity;
0011: move-result-object v3
0012: check-cast v3, Landroid/widget/EditText; // type@0015
0013: const/4 v5, #int 0 // #0
0017: invoke-virtual {v2, Landroid/widget/EditText; getter
0018: move-result-object v7
001b: invoke-virtual {v7, Ljava/lang/Object; toString()IL
001c: move-result-object v7
001f: invoke-static {v7, Ljava/lang/Integer; valueOf(Lj
0020: move-result-object v7
0023: invoke-virtual {v9, Ljava/lang/Integer; intValue()IL
0026: move-result v7
0027: aput v7, v6, v6
0029: const/4 v5, #int 1 // #1
002a: invoke-virtual {v2, Landroid/widget/EditText; getter
002d: move-result-object v7
002e: invoke-virtual {v7, Ljava/lang/Object; toString()IL
0031: move-result-object v7
0032: invoke-static {v7, Ljava/lang/Integer; valueOf(Lj
0035: move-result-object v7
0036: invoke-virtual {v7, Ljava/lang/Integer; intValue()IL
```

## Semantics-Based Features — Reachables

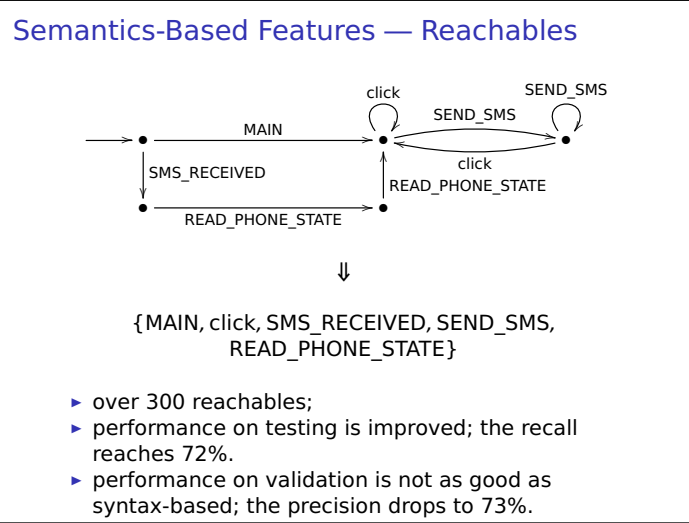
Diagram illustrating a state transition system (Reachables) for a mobile application:

- States: Represented by black dots.
- Transitions:
  - Initial state to State 1: `MAIN`
  - State 1 to State 2: `SEND_SMS`
  - State 2 to State 1: `click` (curved arrow)
  - State 2 to State 2: `SEND_SMS` (curved arrow)
  - State 1 to State 3: `SMS_RECEIVED`
  - State 3 to State 1: `READ_PHONE_STATE`

↓

{MAIN, click, SMS\_RECEIVED, SEND\_SMS, READ\_PHONE\_STATE}

- ▶ 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%.

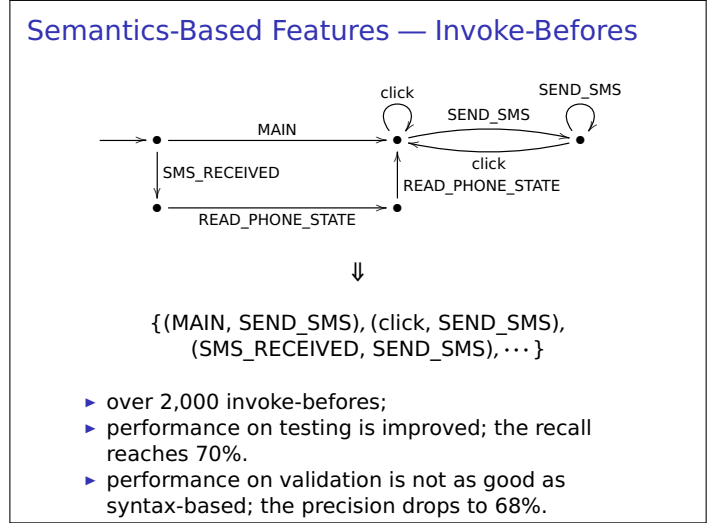


- ## Semantics-Based Features — Reachables
- 
- Diagram illustrating a state transition system (Reachables) for a mobile application:
- States: Represented by black dots.
  - Transitions:
    - Initial state to State 1: `MAIN`
    - State 1 to State 2: `SEND_SMS`
    - State 2 to State 1: `click` (curved arrow)
    - State 2 to State 2: `SEND_SMS` (curved arrow)
    - State 1 to State 3: `SMS_RECEIVED`
    - State 3 to State 1: `READ_PHONE_STATE`
- ↓
- {MAIN, click, SMS\_RECEIVED, SEND\_SMS, READ\_PHONE\_STATE}
- ▶ 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

Diagram illustrating the Invoke-Befores semantics-based feature. The diagram shows a state transition graph with three states. The first state has a self-loop labeled 'SMS\_RECEIVED' and a transition to the second state labeled 'MAIN'. The second state has a self-loop labeled 'click' and a transition to the third state labeled 'SEND\_SMS'. The third state has a self-loop labeled 'SEND\_SMS' and a transition back to the second state labeled 'click'. A transition from the second state to the first state is labeled 'READ\_PHONE\_STATE'. Below the graph, a double arrow points down to a set of invoke-before pairs:  $\{(MAIN, SEND\_SMS), (click, SEND\_SMS), (SMS\_RECEIVED, SEND\_SMS), \dots\}$ .

- ▶ 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 Features — Invoke-Befores
- 
- Diagram illustrating the Invoke-Befores semantics-based feature. The diagram shows a state transition graph with three states. The first state has a self-loop labeled 'SMS\_RECEIVED' and a transition to the second state labeled 'MAIN'. The second state has a self-loop labeled 'click' and a transition to the third state labeled 'SEND\_SMS'. The third state has a self-loop labeled 'SEND\_SMS' and a transition back to the second state labeled 'click'. A transition from the second state to the first state is labeled 'READ\_PHONE\_STATE'. Below the graph, a double arrow points down to a set of invoke-before pairs:  $\{(MAIN, SEND\_SMS), (click, SEND\_SMS), (SMS\_RECEIVED, SEND\_SMS), \dots\}$ .
- ▶ 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:

**$M_0$ :**

- States:  $SEND\_SMS, N, SMS\_RECEIVED, B, V, MAIN, C$
- Transitions:
  - $SEND\_SMS \rightarrow SEND\_SMS$  (self-loop)
  - $MAIN \xrightarrow{B, V} SMS\_RECEIVED$
  - $MAIN \xrightarrow{C} C$
  - $C \xrightarrow{R} MAIN$
  - $C \xrightarrow{R} C$  (self-loop)

**$M_1$ :**

- States:  $SMS\_RECEIVED, SEND\_SMS$
- Transitions:
  - $SMS\_RECEIVED \xrightarrow{MAIN} SEND\_SMS$
  - $SEND\_SMS \xrightarrow{R} SEND\_SMS$  (self-loop)

**$B_0$  and  $B_1$ :**

- States:  $MAIN, R, SMS\_RECEIVED$
- Transitions:
  - $B_0: MAIN \xrightarrow{MAIN} MAIN$  (self-loop)
  - $B_0: MAIN \xrightarrow{R} R$
  - $B_1: SMS\_RECEIVED \xrightarrow{SMS\_RECEIVED} SMS\_RECEIVED$  (self-loop)

**Legend:**

- 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

Example automata:

**$M_0$ :**

- States:  $SEND\_SMS, N, SMS\_RECEIVED, B, V, MAIN, C$
- Transitions:
  - $SEND\_SMS \rightarrow SEND\_SMS$  (self-loop)
  - $MAIN \xrightarrow{B, V} SMS\_RECEIVED$
  - $MAIN \xrightarrow{C} C$
  - $C \xrightarrow{R} MAIN$
  - $C \xrightarrow{R} C$  (self-loop)

**$M_1$ :**

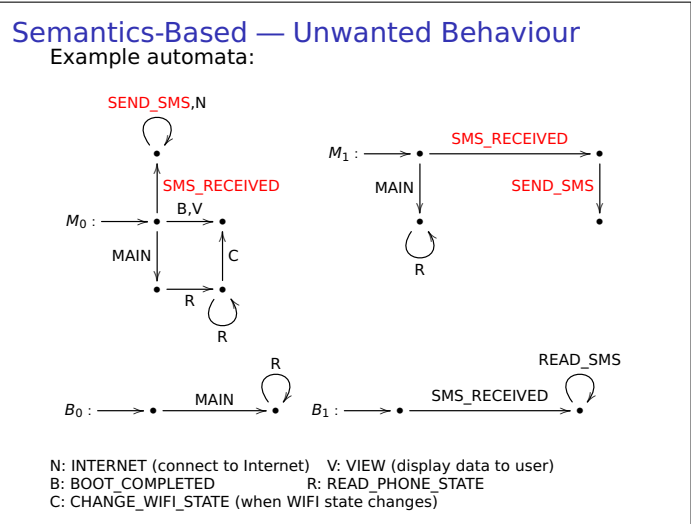
- States:  $SMS\_RECEIVED, SEND\_SMS$
- Transitions:
  - $SMS\_RECEIVED \xrightarrow{MAIN} SEND\_SMS$
  - $SEND\_SMS \xrightarrow{R} SEND\_SMS$  (self-loop)

**$B_0$  and  $B_1$ :**

- States:  $MAIN, R, SMS\_RECEIVED$
- Transitions:
  - $B_0: MAIN \xrightarrow{MAIN} MAIN$  (self-loop)
  - $B_0: MAIN \xrightarrow{R} R$
  - $B_1: SMS\_RECEIVED \xrightarrow{SMS\_RECEIVED} SMS\_RECEIVED$  (self-loop)

**Legend:**

- 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

Example automata:

**$M_0$ :**

- States:  $SEND\_SMS, N, SMS\_RECEIVED, B, V, MAIN, C$
- Transitions:
  - $SEND\_SMS \rightarrow SEND\_SMS$  (self-loop)
  - $MAIN \xrightarrow{B, V} SMS\_RECEIVED$
  - $MAIN \xrightarrow{C} C$
  - $C \xrightarrow{R} MAIN$
  - $C \xrightarrow{R} C$  (self-loop)

**$M_1$ :**

- States:  $SMS\_RECEIVED, SEND\_SMS$
- Transitions:
  - $SMS\_RECEIVED \xrightarrow{MAIN} SEND\_SMS$
  - $SEND\_SMS \xrightarrow{R} SEND\_SMS$  (self-loop)

**$B_0$  and  $B_1$ :**

- States:  $MAIN, R, SMS\_RECEIVED$
- Transitions:
  - $B_0: MAIN \xrightarrow{MAIN} MAIN$  (self-loop)
  - $B_0: MAIN \xrightarrow{R} R$
  - $B_1: SMS\_RECEIVED \xrightarrow{SMS\_RECEIVED} SMS\_RECEIVED$  (self-loop)

**Legend:**

- 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

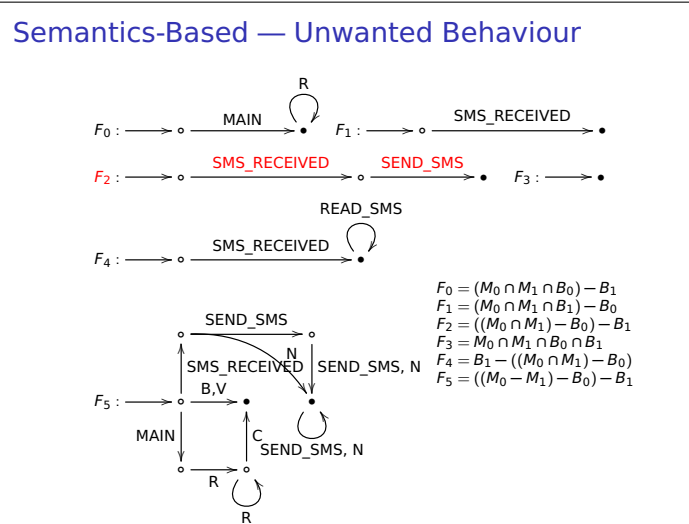
The diagram illustrates a sequence of states and transitions for a system. The states are represented by circles, and transitions are labeled with actions. The sequence of states is  $F_0, F_1, F_2, F_3, F_4, F_5$ .

Transitions and self-loops shown:

- $F_0 \xrightarrow{\text{MAIN}} F_1$  (with a self-loop  $R$  on  $F_0$ )
- $F_1 \xrightarrow{\text{SMS\_RECEIVED}} F_2$
- $F_2 \xrightarrow{\text{SEND\_SMS}} F_3$  (with a self-loop  $R$  on  $F_2$ )
- $F_3 \xrightarrow{\text{READ\_SMS}} F_4$
- $F_4 \xrightarrow{\text{SMS\_RECEIVED}} F_5$  (with a self-loop  $R$  on  $F_4$ )
- $F_5 \xrightarrow{\text{MAIN}} F_5$  (with a self-loop  $R$  on  $F_5$ )
- From  $F_5$ , there is a transition  $\xrightarrow{\text{SMS\_RECEIVED}}$  to a state with a self-loop  $R$ , and another transition  $\xrightarrow{\text{SEND\_SMS, N}}$  to a state with a self-loop  $R$ .

Summary of states and transitions:

| State | Definition                       |
|-------|----------------------------------|
| $F_0$ | $(M_0 \cap M_1 \cap B_0) - B_1$  |
| $F_1$ | $(M_0 \cap M_1 \cap B_1) - B_0$  |
| $F_2$ | $((M_0 \cap M_1) - B_0) - B_1$   |
| $F_3$ | $M_0 \cap M_1 \cap B_0 \cap B_1$ |
| $F_4$ | $B_1 - ((M_0 \cap M_1) - B_0)$   |
| $F_5$ | $((M_0 - M_1) - B_0) - B_1$      |



## Semantics-Based — Unwanted Behaviour

The diagram illustrates a sequence of states and transitions for a system. The states are represented by circles, and transitions are labeled with actions. The sequence of states is  $F_0, F_1, F_2, F_3, F_4, F_5$ .

Transitions and self-loops shown:

- $F_0 \xrightarrow{\text{MAIN}} F_1$  (with a self-loop  $R$  on  $F_0$ )
- $F_1 \xrightarrow{\text{SMS\_RECEIVED}} F_2$
- $F_2 \xrightarrow{\text{SEND\_SMS}} F_3$  (with a self-loop  $R$  on  $F_2$ )
- $F_3 \xrightarrow{\text{READ\_SMS}} F_4$
- $F_4 \xrightarrow{\text{SMS\_RECEIVED}} F_5$  (with a self-loop  $R$  on  $F_4$ )
- $F_5 \xrightarrow{\text{MAIN}} F_5$  (with a self-loop  $R$  on  $F_5$ )
- From  $F_5$ , there is a transition  $\xrightarrow{\text{SMS\_RECEIVED}}$  to a state with a self-loop  $R$ , and another transition  $\xrightarrow{\text{SEND\_SMS, N}}$  to a state with a self-loop  $R$ .

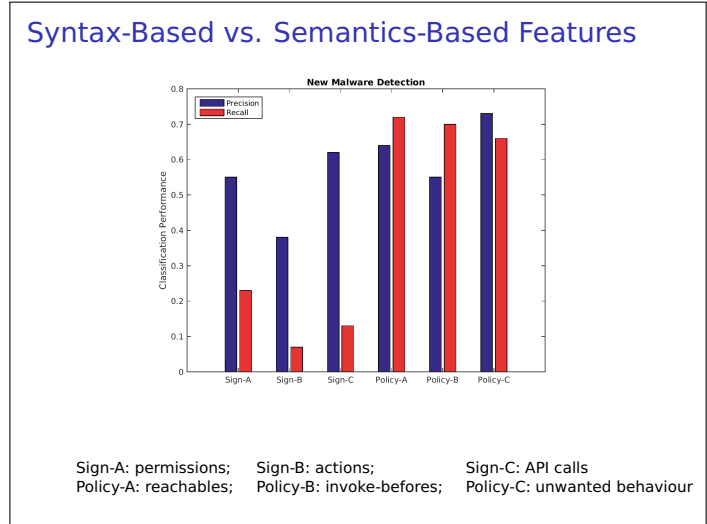
Summary of states and transitions:

| State | Definition                       |
|-------|----------------------------------|
| $F_0$ | $(M_0 \cap M_1 \cap B_0) - B_1$  |
| $F_1$ | $(M_0 \cap M_1 \cap B_1) - B_0$  |
| $F_2$ | $((M_0 \cap M_1) - B_0) - B_1$   |
| $F_3$ | $M_0 \cap M_1 \cap B_0 \cap B_1$ |
| $F_4$ | $B_1 - ((M_0 \cap M_1) - B_0)$   |
| $F_5$ | $((M_0 - M_1) - B_0) - B_1$      |

# Syntax-Based vs. Semantics-Based Features

| Model    | Precision | Recall |
|----------|-----------|--------|
| Sign-A   | 0.55      | 0.23   |
| Sign-B   | 0.38      | 0.07   |
| Sign-C   | 0.62      | 0.13   |
| Policy-A | 0.64      | 0.72   |
| Policy-B | 0.55      | 0.70   |
| Policy-C | 0.73      | 0.66   |

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



# Syntax-Based vs. Semantics-Based Features

| Model    | Precision | Recall |
|----------|-----------|--------|
| Sign-A   | 0.55      | 0.23   |
| Sign-B   | 0.38      | 0.07   |
| Sign-C   | 0.62      | 0.13   |
| Policy-A | 0.64      | 0.72   |
| Policy-B | 0.55      | 0.70   |
| Policy-C | 0.73      | 0.66   |

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

| Training method | Training feature | $\rho_1$ | $\rho_{0.5} \downarrow$ |
|-----------------|------------------|----------|-------------------------|
| NB              | actions          | 76       | 71                      |
| L1LR            | reachables ✓     | 74       | 70                      |
| NB              | reachables ✓     | 72       | 70                      |
| L1LR            | unwanted ✓       | 71       | 70                      |
| NB              | happen-befores ✓ | 70       | 67                      |
| SVM             | keywords         | 73       | 66                      |
| DT              | happen-befores ✓ | 70       | 65                      |
| AdaBoost        | keywords         | 71       | 64                      |
| KNN             | keywords         | 71       | 64                      |
| NB              | permissions      | 71       | 64                      |
| L1LR            | happen-befores ✓ | 69       | 64                      |
| RF              | happen-befores ✓ | 69       | 64                      |
| SEMI            | happen-befores ✓ | 68       | 63                      |
| NB              | keywords         | 68       | 59                      |

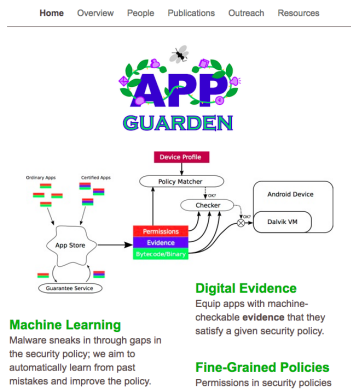
## Evaluation — Least Robust General Classifiers

| Training method | Training feature | $\rho_1$ | $\rho_{0.5} \uparrow$ |
|-----------------|------------------|----------|-----------------------|
| SEMI            | API calls        | 14       | 9                     |
| RF              | API calls        | 14       | 9                     |
| NB              | API calls        | 19       | 13                    |
| SVM             | actions          | 19       | 13                    |
| L1LR            | actions          | 21       | 14                    |
| AdaBoost        | actions          | 21       | 15                    |
| DT              | API calls        | 25       | 17                    |
| SVM             | API calls        | 26       | 18                    |
| KNN             | actions          | 27       | 19                    |
| AdaBoost        | API calls        | 27       | 19                    |
| RF              | actions          | 29       | 20                    |
| SEMI            | actions          | 31       | 22                    |
| DT              | actions          | 33       | 23                    |
| L1LR            | API calls        | 35       | 25                    |

## Questions and Discussion

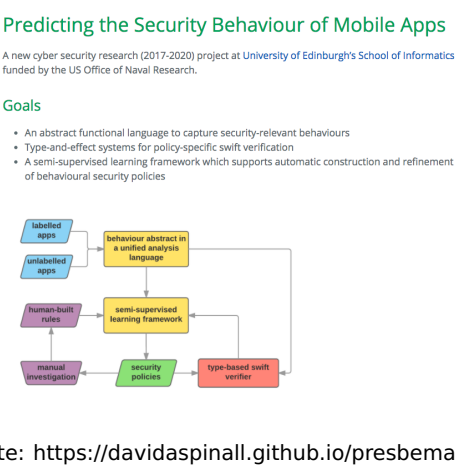
- ▶ How do you think about machine learning methods?
- ▶ How could we improve the construction of unwanted behaviour? (**on-going research**)
- ▶ Is there any other model we can learn for malware detection? (**on-going research**)

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



Website:  
<http://groups.inf.ed.ac.uk/security/appguarden/Home.html>

## More Information — PreSBema (2017 — 2020)



## 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).

## Malware analysis: techniques

**Dynamic analysis:** run, emulate, or simulate (part of) an app to produce traces and check properties.

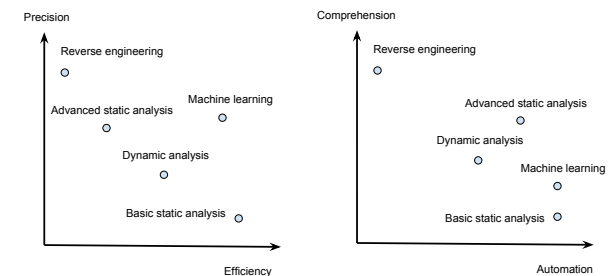
Efficient (in minutes per app) but often **under-approximated** (miss something that will happen) and hard to mimic user input.

**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



## Further Readings



Chen et al. More Semantics More Robustness: Improving Android Malware Classifiers. WISec 16.

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

Seghir et al. Certified Lightweight Contracts for Android. SecDev 16.

Arzt et al. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. PLDI 14.