Modeling the Android Platform

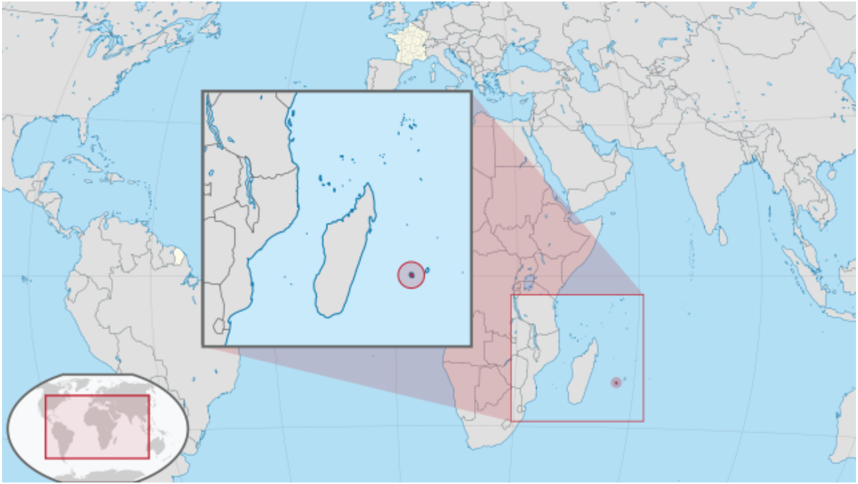
Étienne Payet

LIM-ERIMIA, université de la Réunion

BYTECODE'13

Saturday 23 March 2013

Reunion, a part of France and Europe (OMR of EU)



Outline

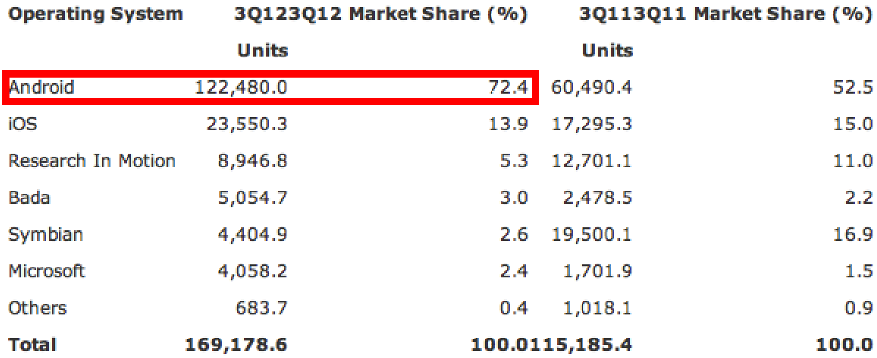
1. Analyzing Android applications
2. Operational semantics for Dalvik
3. Designing an operational semantics for Android
4. Conclusion

What is Android?

An operating system (OS) for:

- mobile devices (smartphones, tablets),
- embedded devices (televisions, car radios, . . .),
- x86 platforms (`http://www.android-x86.org`).

Worldwide mobile device sales in 3Q12 (thousands of units)



Operating System	3Q12 Units	3Q12 Market Share (%)	3Q11 Units	3Q11 Market Share (%)
Android	122,480.0	72.4	60,490.4	52.5
iOS	23,550.3	13.9	17,295.3	15.0
Research In Motion	8,946.8	5.3	12,701.1	11.0
Bada	5,054.7	3.0	2,478.5	2.2
Symbian	4,404.9	2.6	19,500.1	16.9
Microsoft	4,058.2	2.4	1,701.9	1.5
Others	683.7	0.4	1,018.1	0.9
Total	**169,178.6**	**100.0**	**115,185.4**	**100.0**

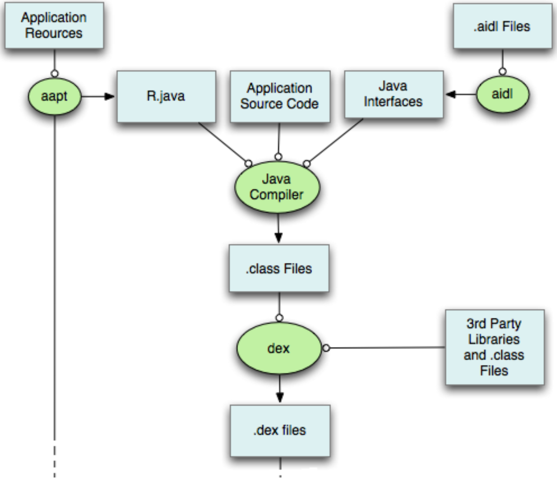
Source: Gartner (November 2012)

What is Android?

A language:

- for developping applications for the Android OS,
- Java with an extended library for mobile and interactive applications,
- based on an event-driven architecture.

Building an Android application



(http://developer.android.com/tools/building/index.html)

.dex files

- Their format is optimized for minimal memory usage: the design is driven by sharing of data,
- they contain Dalvik bytecode,
- dex stands for Dalvik executable.

Dalvik bytecode

- It is run by an instance of the Dalvik Virtual Machine (DVM),
- DVM $\neq$ JVM (register-based vs stack-based),
- register-based VMs are well-suited for devices with constrained processing power: on average, they are faster than stack-based VMs.

Android applications

- They can be downloaded from *anywhere*
  - Google play (official store),
  - Amazon, AppsApk.com, pandaapp, . . .

- They are not necessarily digitally signed.

⇒ Reliability is a major concern for users and developers.

Mobile Phone Malware

In fact, we learned that Google, tired of the malicious apps found on Play Store, has started analyzing apps before putting them in their catalog in order to detect anomalous behavior. According to their own sources, they have managed to reduce malicious app downloads by 40 percent.

Unfortunately, despite these efforts, criminals continued to target the Android mobile platform through apps not always accessible through Play Store. This was the case of Bmaster, a remote access Trojan (RAT) on the Android platform that tried to pass itself off as a legitimate application.

We also saw Trojans exclusively designed to steal data from infected devices: from call and text message records to users' contact lists. Android is potentially exposed to far more security risks than its biggest competitor (iPhone and its iOS), as it allows users to get their apps from anywhere their want. However, using the official Android marketplace is no security guarantee either, as it has also been targeted by cyber-crooks luring users into installing Trojans disguised as legitimate apps. Something which, by the way, has also happened to Apple's App Store, but to a lesser extent than to Google's Play Store.

2012 at a glance

ned primarily for mobile phones. Over the last few months,
y as a mobile browser alternative on Android smartphones,
als to trick users. In the latest attack, criminals offered the
than Google's Play store. However, installing the application
and also a Trojan that sent SMS messages to premium-rate

attempted to pass themselves off as popular mobile apps, in
d with a legitimate version of the Opera Mini mobile browser
t nothing was wrong as they could simply use the real software

FIG.02. *500 MILLION ANDROID DEVICES NOW ACTIVATED.*

Why is Android the most targeted mobile platform? Well, this is due to a number of reasons: Firstly, Android allows its users to get their apps from anywhere they want. They don't necessarily have to go to the official store, nor must applications be digitally signed as with iOS. Secondly, cyber-crooks would have never set their eyes on this platform if it weren't for the large number of users it has. In June, Google announced that 400 million Android devices had been activated, a figure that reached 500 million at the beginning of September, with 1.3 million activations per day.

Analyzing Android applications

For finding

- malicious code (*e.g.*, security and privacy vulnerabilities)
- bugs

Google's analyses

"Google has started analyzing apps before putting them in their catalog in order to detect anomalous behavior. According to their own sources, they have managed to reduce malicious app downloads by 40 percent."

(PandaLabs Annual Report 2012)

Static analyses for finding security/privacy vulnerabilities

- Barrera, Kayacik, van Oorschot, Somayaji. *A methodology for empirical analysis of permission-based security models and its application to Android*. Proc. of CCS'10.
- Chin, Felt, Greenwood, Wagner. *Analyzing inter-application communication in Android*. Proc. of MobiSys'11.
- Enck, Octeau, McDaniel, Chaudhuri. *A study of Android application security*. Proc. of SEC'11.
- Felt, Chin, Hanna, Song, Wagner. *Android permissions demystified*. Proc. of CCS'11.
- Fuchs, Chaudhuri, Foster. *SCanDroid: Automated security certification of Android applications*. Draft, 2009.
- Kim, Yoon, Yi, Shin. *ScanDal: Static analyzer for detecting privacy leaks in Android applications*. MoST'12.
- Wognsen, Karlsen. *Static analysis of Dalvik bytecode and reflection in Android*. Master's thesis, Aalborg University, 2012.

Static analyses for finding bugs

- Klocwork. `http://www.klocwork.com`.
- Payet, Spoto. *Static analysis of Android programs*. Information & Software Technology, 2012.

Dynamic analyses for finding security vulnerabilities

- Bugiel, Davi, Dmitrienko, Fischer, Sadeghi, Shastry. *Towards taming privilege-escalation attacks on Android*. Proc. of NDSS'12.
- Dietz, Shekhar, Pisetsky, Shu, Wallach. *QUIRE: Lightweight provenance for smart phone operating systems*. Proc. of USENIX Security Symposium. 2011.
- Enck, Gilbert, Chun, Cox, Jung, McDaniel, Sheth. *TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones*. Proc. of OSDI'10.
- Felt, Wang, Moshchuk, Hanna, Chin. *Permission redelegation: Attacks and defenses*. Proc. of USENIX Security Symposium. 2011.

Symbolic execution for analyzing programs

- Jeon, Micinski, Foster. *SymDroid: Symbolic execution for Dalvik bytecode*. Submitted, July 2012.

Modeling the Android platform

- Dalvik $\neq$ Android

- Some of these analyses rely on a formal operational semantics for Dalvik.

- But none of them provide a formal semantics for key specific features of the Android platform.

Outline

Dalvik registers

- Each method has a fresh set of registers.
- Invoked methods do not affect the registers of invoking methods.

Dalvik instructions

- Move between registers (`move`, `move-object`, `move-wide`, ...),
- constants to registers (`const`, `const/4`, `const/16`, ...),
- operations on int, long, float, double (`add-int`, `sub-int`, ...),
- instance creation (`new-instance`),
- read/write member fields (`iget`, `iput`, ...),
- read/write static fields (`sget`, `sput`, ...),
- array manipulation (`new-array`, `array-length`, ...),
- read/write array elements (`aget`, `aput`, ...),
- execution control (`goto`, `if-eq`, `if-lt`, ...),
- method invocation (`invoke-virtual`, `invoke-super`, ...),
- setting the result value (`return-void`, `return`, ...),
- getting the result value (`move-result`, `move-result-object`, ...),
- ...

Example (smali syntax)

```
.class public LFactorial;
.super Ljava/lang/Object;

.method public static factorial(I)I
    .registers 2

    const/4 v0, 1

    if-lez v1, :end

    sub-int v0, v1, v0
    invoke-static {v0}, LFactorial;->factorial(I)I
    move-result v0
    mul-int v0, v1, v0

    :end
    return v0
.end method
```

Operational semantics for the whole Dalvik

[WK12] Wognsen, Karlsen. *Static analysis of Dalvik bytecode and reflection in Android*. Master's thesis, Aalborg University, 2012.

$$\frac{m.instructionAt(pc) = \texttt{move}\ r_1\ r_2}{\langle S, H, \langle m, pc, R\rangle :: SF\rangle \Rightarrow \langle S, H, \langle m, pc+1, R[r_1 \mapsto R(r_2)]\rangle :: SF\rangle}$$

S is a static heap, H is a heap, SF is a call stack
m is a method, $R \in Register \rightarrow Value$ is a set of local registers

Intermediate languages

- They consist of a small set of instructions into which Dalvik can be easily translated.
- Dalvik Core:

  Kim, Yoon, Yi, Shin. *ScanDal: Static analyzer for detecting privacy leaks in Android applications*. MoST'12.
- μ-Dalvik:

  Jeon, Micinski, Foster. *SymDroid: Symbolic execution for Dalvik bytecode*. Submitted, July 2012.

μ-Dalvik vs the others

- μ-Dalvik operational semantics constructs a path condition ϕ which records which conditional branches have been taken thus far:

$$\frac{\begin{array}{c}\pi = (\Sigma[\![r_1]\!] \trianglelefteq \Sigma[\![r_2]\!]) \\ \phi_t = \pi \wedge \Sigma.\phi \qquad \mathrm{SAT}(\phi_t)\end{array}}{\langle \Sigma, \texttt{if}\ r_1 \trianglelefteq r_2\ \texttt{then}\ pc_t \rangle \Rightarrow \Sigma[\phi \mapsto \phi_t,\ pc \mapsto pc_t]}$$

- μ-Dalvik provides an instruction for checking a property of interest:

$$\frac{\neg\mathrm{SAT}(\neg\Sigma[\![r]\!])}{\langle \Sigma, \texttt{assert}\ r \rangle \Rightarrow \Sigma[pc \mapsto pc + 1]}$$

Outline

1. Analyzing Android applications

2. Operational semantics for Dalvik

3. Designing an operational semantics for Android

4. Conclusion

Goal

Provide a formal basis for the development of analyses that consider the complex flow of information inside Android applications, that usually consist of interacting components.

Android application components

(Activities) single screens with a visual user interface

(Services) background operations with no interaction with the user

(Content providers) data containers such as databases

(Broadcast receivers) objects reacting to broadcast messages

Android application components

- Each type of component has a distinct lifecycle that defines how the component changes state.

- A component can invoke another component, but

  component invocation $\neq$ method invocation.

- A component is a possible entry point into the program.

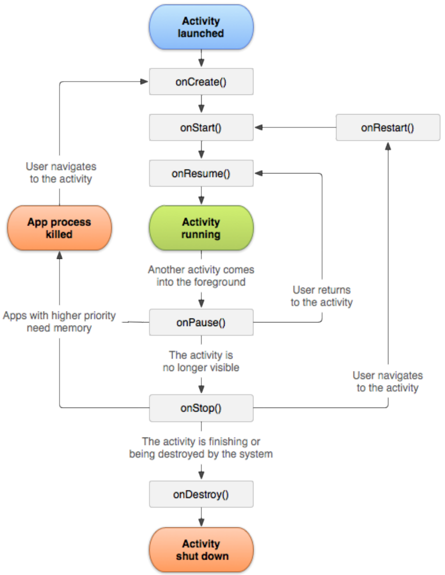
Callback methods

Callback methods are automatically invoked by the system:

- when components switch from state to state,
- in reaction to events.

Android programs do not usually call such methods explicitly.

The lifecycle of an activity



XML files

- They are used to build parts of Android applications (*e.g.*, GUI).
- They are dynamically inflated by the system to create the objects that they describe.
- Inflation makes heavy use of reflection.

An example (1/5)

res/layout/caller.xml

```xml
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TextView android:id="@+id/message"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/empty" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/launch"
        android:onClick="launchActivity" />

</LinearLayout>
```

An example (2/5)

Caller.java

```
public class Caller extends android.app.Activity {

  private TextView mMessageView;

  protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.caller);
      mMessageView = (TextView) findViewById(R.id.message);
  }

  public void launchActivity(View v) {
      ...
  }

  ...
}
```

An example (3/5)

Caller.java

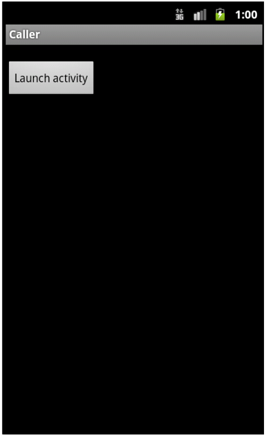
```
private final static int CALLEE = 0;

public void launchActivity(View v) {
    startActivityForResult(new Intent(this, Callee.class), CALLEE);
    System.out.println("Hello!");
}

protected void onActivityResult(int requestCode, int resultCode, ...) {
    switch(requestCode) {
    case CALLEE:
        switch(resultCode) {
        case RESULT_OK:
            mMessageView.setText("OK button clicked"); break;
        case RESULT_CANCELED:
            mMessageView.setText("Cancel button clicked"); break;
        }
    }
}
```

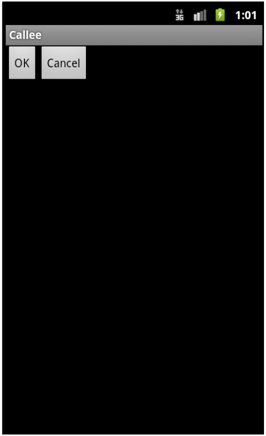
An example (4/5)

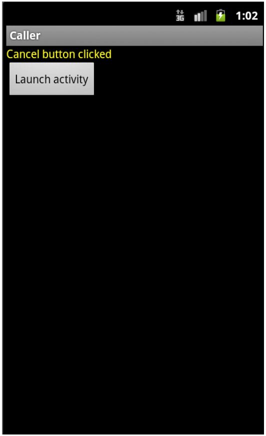
Callee.java

```
public class Callee extends android.app.Activity {

  protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.callee);
  }

  public void returnOk(View v) { // OK button clicked.
      setResult(RESULT_OK);
      finish();
  }

  public void returnCancel(View v) { // Cancel button clicked.
      setResult(RESULT_CANCELED);
      finish();
  }
}
```

An example (5/5)







Cancel button clicked

DVM state

$$\langle r \parallel \pi \parallel \mu \rangle \in \Sigma$$

- r is an array of registers (r^i denotes the ith register)
- π is a stack of pending activities
- $\mu : \mathit{Location} \rightarrow \mathit{Object}$ is a heap
- an object maps its fields into values

Semantics of Dalvik instructions

- $\texttt{const}\ d, c = \lambda\langle r \parallel \pi \parallel \mu\rangle.\langle r[d \mapsto c] \parallel \pi \parallel \mu\rangle$
- $\texttt{iget}\ d, i, f = \lambda\langle r \parallel \pi \parallel \mu\rangle. \begin{cases} \langle r[d \mapsto \mu(r^i)(f)] \parallel \pi \parallel \mu\rangle & \text{if } r^i \neq 0 \\ \textit{undefined} & \text{otherwise} \end{cases}$

Semantics of library methods (macro instructions)

- *finished* (boolean) and *res* (integer) are fields of the current activity
- `startActivityForResult` $A = \lambda\langle r \parallel \pi \parallel \mu\rangle.\langle r \parallel A :: \pi \parallel \mu\rangle$

  where A is a subclass of `android.app.Activity`
- `setResult` $i = \lambda\langle r \parallel \pi \parallel \mu\rangle.\begin{cases}\langle r \parallel \pi \parallel \mu\rangle & \text{if } \mathit{finished}\\ \langle r \parallel \pi \parallel \mu[\mathit{res} \mapsto i]\rangle & \text{otherwise}\end{cases}$
- `finish` $= \lambda\langle r \parallel \pi \parallel \mu\rangle.\langle r \parallel \pi \parallel \mu[\mathit{finished} \mapsto \mathit{true}]\rangle$

Android programs as graphs of blocks

- A program is a graph of blocks of code.
- A graph contains many disjoint subgraphs, each corresponding to a different method.
- A block with w instructions and p successor blocks is written as

$$\boxed{\begin{array}{c}\texttt{ins}_1\\ \texttt{ins}_2\\ \cdots\\ \texttt{ins}_w\end{array}} \rightrightarrows \begin{array}{c} b_1\\ \cdots\\ b_p\end{array}$$

- If m is a method, then b_m denotes the block where m starts.

Operational semantics of method execution

- (Instruction execution) $\texttt{ins} \notin \{\texttt{call}, \texttt{move-result}, \texttt{return}\}$

$$\frac{\langle r' \parallel \pi' \parallel \mu' \rangle = \texttt{ins}(\langle r \parallel \pi \parallel \mu \rangle)}{\langle \boxed{\begin{array}{c}\texttt{ins}\\ rest\end{array}} \rightrightarrows \begin{array}{c}b_1\\ \cdots\\ b_p\end{array} \parallel r \rangle :: \alpha \diamond \pi \diamond \mu \rightsquigarrow \langle \boxed{rest} \rightrightarrows \begin{array}{c}b_1\\ \cdots\\ b_p\end{array} \parallel r' \rangle :: \alpha \diamond \pi' \diamond \mu'}$$

- (Continuation)

$$\frac{1 \leq i \leq p}{\langle \boxed{} \rightrightarrows \begin{array}{c}b_1\\ \cdots\\ b_p\end{array} \parallel r \rangle :: \alpha \diamond \pi \diamond \mu \rightsquigarrow \langle b_i \parallel r \rangle :: \alpha \diamond \pi \diamond \mu}$$

Operational semantics of method execution

- (Explicit method call)

$$\dfrac{\begin{array}{c} b = \boxed{\begin{array}{c}\texttt{call}\ \{s_0,\ldots,s_w\},m \\ rest\end{array}} \rightrightarrows \begin{array}{c} b_1 \\ \cdots \\ b_p \end{array} \qquad b' = \boxed{rest} \rightrightarrows \begin{array}{c} b_1 \\ \cdots \\ b_p \end{array} \\ r' = [0 \mapsto r^{s_0}, \ldots, w \mapsto r^{s_w}] \\ \text{the lookup procedure of } m \text{ selects } m' \end{array}}{\langle b \parallel r \rangle :: \alpha \diamond \pi \diamond \mu \rightsquigarrow \langle b_{m'} \parallel r' \rangle :: \langle b' \parallel r \rangle :: \alpha \diamond \pi \diamond \mu}$$

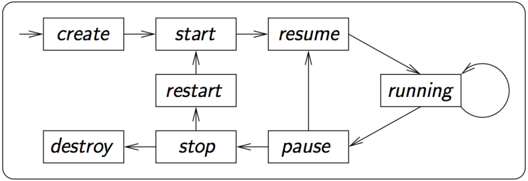
- (Method return)

$$\dfrac{b = \boxed{\begin{array}{c}\texttt{move-result}\ d \\ rest\end{array}} \rightrightarrows \begin{array}{c} b_1 \\ \cdots \\ b_p \end{array} \quad b' = \boxed{rest} \rightrightarrows \begin{array}{c} b_1 \\ \cdots \\ b_p \end{array}}{\langle \boxed{\texttt{return}\ s} \parallel r \rangle :: \langle b \parallel r' \rangle :: \alpha \diamond \pi \diamond \mu \rightsquigarrow \langle b' \parallel \langle r'[d \mapsto r^s] \rangle :: \alpha \diamond \pi \diamond \mu}$$

Operational semantics of activity execution

- Android manages activities using an activity stack (Ω).

- We formalize an activity as a tuple $\langle \ell \parallel s \parallel \pi \parallel \alpha \rangle$:
  - ℓ is the location of the activity in memory,
  - s is the lifecycle state of the activity.



Moves between lifecycle states.

Operational semantics of activity execution

- (Implicit call to a callback method)

$$\frac{\begin{array}{c} s \neq \mathit{running} \qquad (s, s') \in \mathit{Lifecycle} \\ \text{the lookup procedure of a method corresponding to } s' \text{ selects } m \end{array}}{\langle \ell \parallel s \parallel \pi \parallel \langle \boxed{\texttt{return}} \parallel \_ \rangle\rangle :: \Omega \diamond \mu \Rightarrow \langle \ell \parallel s' \parallel \pi \parallel \langle b_m \parallel [\ell] \rangle\rangle :: \Omega \diamond \mu}$$

$$\frac{\begin{array}{c} s = \mathit{running} \qquad (s, s') \in \mathit{Lifecycle} \\ \pi \neq \varepsilon \vee \mu(\ell)(\mathit{finished}) = \mathit{true} \Rightarrow s' = \mathit{pause} \\ \text{the lookup procedure of a method corresponding to } s' \text{ selects } m \end{array}}{\langle \ell \parallel s \parallel \pi \parallel \langle \boxed{\texttt{return}} \parallel \_ \rangle\rangle :: \Omega \diamond \mu \Rightarrow \langle \ell \parallel s' \parallel \pi \parallel \langle b_m \parallel [\ell] \rangle\rangle :: \Omega \diamond \mu}$$

Operational semantics of activity execution

- (Starting a new activity)

$$\frac{\begin{array}{c} s = \mathit{pause} \qquad \alpha = \langle \boxed{\texttt{return}\quad} \parallel \_ \rangle \\ \ell' \text{ is a fresh location and } a \text{ is a new object of class } A \\ \text{the lookup procedure of a method corresponding to } s' \text{ selects } m \\ s' = \mathit{create} \qquad \alpha' = \langle b_m \parallel [\ell'] \rangle \qquad \mu' = \mu[\ell' \mapsto a] \end{array}}{\langle \ell \parallel s \parallel A :: \pi \parallel \alpha \rangle :: \Omega \diamond \mu \Rightarrow \langle \ell' \parallel s' \parallel \varepsilon \parallel \alpha' \rangle :: \langle \ell \parallel s \parallel \pi \parallel \alpha \rangle :: \Omega \diamond \mu'}$$

- (Returning from an activity)

$$\frac{\begin{array}{c} \varphi' = \langle \ell' \parallel \mathit{pause} \parallel \varepsilon \parallel \langle \boxed{\texttt{return}\quad} \parallel \_ \rangle \rangle \qquad \mu(\ell')(\mathit{finished}) = \mathit{true} \\ \varphi = \langle \ell \parallel s \parallel \varepsilon \parallel \langle \boxed{\texttt{return}\quad} \parallel \_ \rangle \rangle \qquad s \in \{\mathit{pause}, \mathit{stop}\} \\ \text{the lookup procedure of } \texttt{onActivityResult} \text{ selects } m \end{array}}{\varphi' :: \varphi :: \Omega \diamond \mu \Rightarrow \langle \ell \parallel s \parallel \varepsilon \parallel \langle b_m \parallel [\ell] \rangle \rangle :: \varphi' :: \Omega \diamond \mu}$$

Outline

1. Analyzing Android applications

2. Operational semantics for Dalvik

3. Designing an operational semantics for Android

4. **Conclusion**

Analyzing Android applications

- For finding bugs and malicious code.
- Formal semantics can provide a formal basis.
- Some operational semantics have been proposed for Dalvik.
- This work is the first attempt at defining an operational semantics for Android.

Modeling the whole Android platform

- We consider a simplified situation:
  - programs only consist of activities,
  - activity interactions only occur in state *running*.

- The whole platform is very complex to model:
  - applications may consist of several kinds of components,
  - activity interactions may occur in other states than *running*,
  - there is a large number of implicitly invoked callback methods,
  - a component of another program may be invoked,
  - . . .

Thank you!

Questions?