An Operational Semantics for Android Activities

Étienne Payet
LIM, Université de la Réunion, France
etienne.payet@univ-reunion.fr

Fausto Spoto
Dipartimento di Informatica, Università di Verona, Italy
fausto.spoto@univr.it

Abstract

We define an operational semantics for a large part of the Android platform, encompassing the Dalvik bytecode but also, and more importantly, the inter-component communication mechanism used inside Android applications. This semantics is intended to provide a formal basis for the development of static analyses that consider the complex flow of information exposed by the cooperating components of Android applications.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical Verification; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational semantics, Program analysis

General Terms Languages, Theory, Verification

Keywords Operational Semantics, Android, Dalvik, Static Analysis

1. Introduction

Android is an operating system for mobile and embedded devices. It is currently the most widespread system for smartphones. Applications are written in Java and compiled to *Dalvik* bytecode [6]; they can use a large part of the standard Java library; they can be downloaded from anywhere, for instance from the official *Google Play* store, or developed by the user on a computer; moreover, they do not necessarily have to be digitally signed before installation, as it is the case, for instance, with iOS. As a consequence, reliability is a major concern for Android users and developers, since buggy applications can hang the device and malicious applications can steal private data and send them through the Internet.

Therefore, the analysis of Android applications is an important topic as it allows the developer to discover bugs in a program under development, and the user to be aware of bugs/malicious code in a downloaded program. Several different static analyses of Android programs have been presented so far; most of them have been developed to find security or privacy vulnerabilities [3, 5, 9, 10, 12, 16, 23]; others to find bugs [17, 19]. On the other hand, several dynamic analysis tools [4, 7, 8, 11] have been developed to identify security vulnerabilities. We are also aware of SymDroid [14], a symbolic executor for Dalvik, that can be used for analyzing Android programs.

It has to be noticed that Android is an event-driven system with very specific functionalities. Each application runs in its own instance of the Dalvik virtual machine, which executes the bytecode. Applications may have several entry points that are selected by the system. They are made of components, each with a specific lifecycle. Callback methods are automatically invoked by the system when components switch from state to state, or in reaction to events. As a consequence, Android programs do not usually call such methods explicitly. A specificity of Android is the use of XML to code parts of the applications, as is notably the case for user interfaces. XML files are dynamically *inflated* by the system into the objects that they describe. This inflation process makes heavy use of Java reflection. One final specificity of Android is the possibility for a component to launch another component, also belonging to a distinct program, and wait for a result. The launched component can be explicitly named by the invoker, but can also be selected dynamically at run-time on the basis of the intent of the invoker and of the list of applications installed in the device. Component invocation must not be confused with method invocation, which works differently.

The execution of Android programs is hence very complex to model, much more than Dalvik itself, whose way of working is relatively straightforward. Several operational semantics have recently been presented for Dalvik, as well as several manual implementations of some specific features of Android. In [16], the intermediate language *Dalvik Core*, together with a corresponding formal semantics, is introduced. It consists of 15 instructions that represent the whole Dalvik. It is used inside SCANDAL [20], an analyzer that also includes manual implementations of the semantics and calling conventions of many Android API methods, such as methods that add listeners, initialize threads or launch special kinds of components. SCANDAL handles simple cases of reflection by analyzing string values. In [14], the authors introduce μ-Dalvik, a language consisting of 16 instructions into which Dalvik can be easily translated. An operational semantics for μ-Dalvik is presented and it is used in the core of SymDroid, a symbolic executor for Dalvik written in OCaml. SymDroid includes manual implementations of several parts of important system classes. But it is up to the user of this tool to model behaviors of the Android system, by using client-oriented specifications [13], in order to drive the application under test as desired. In [23], an operational semantics is specified for the whole Dalvik, except instructions related to concurrency. An operational semantics for central parts of the Java reflection API is also presented. In [19], the Julia static analyzer [15] for Java bytecode is extended to perform formally correct analyses of Android programs, provided as `.jar` files. Several Android key specific features are considered, such as the potential existence of many entry points to a program and the inflation, through reflection, of XML graphical views. These features are simulated through synthetic code that mimics the behavior of Android.

As far as we know, there is currently no formal definition of an operational semantics for Android, not just Dalvik. Hence, in this article, we consider a substantial part of the platform, that goes well beyond the simple Dalvik bytecode, and formally present an operational semantics for it. In particular, we formalize the intercomponent communication mechanism of Android, which is essential in order to analyze applications made up of more cooperating components, for instance, of many *activity* screens; each screen represents a unit of interaction with the user and screen swapping is implemented through a stack of activities, that are allowed to communicate and exchange data. It is very important to take into account this feature of Android and model it with an operational semantics. For instance, this allows one to consider information flows where data travels from activity to activity, which is essential for the assessment of confidentiality. More generally, it allows one to link the input received from an activity to the data passed by another activity and prove that the expected input is passed correctly.

The rest of this paper is organized as follows. Section 2 discusses concrete situations when a semantics like ours is needed as the basis for static analysis. Section 3 introduces some key Android features, highlights the scope of this work, and provides notations. Section 4 defines the syntax of a small but non-trivial subset of the Dalvik bytecode that we use in our definitions. Section 5 presents an operational semantics for Android. Section 6 concludes the paper.

2. Motivation

We discuss why a new semantics, modeling the activity lifecycle and their interprocess communication mechanism, is needed for static analysis and which kind of analyses would benefit from it.

The key observation is that, without an explicit modeling of the lifecycle and communication mechanism of activities, the only sound assumption for a static analysis is to assume that the entry points of activities are called non-deterministically, with parameters bound to any value compatible with their declared types. This might be acceptable for some static analyses, such as class analysis, where the concrete class of the parameters of the entry points would end up being approximated with all possible instances of their declared types, an imprecise but still sound and acceptable compromise, that corresponds to *rapid type analysis* [22].

The same approach might be acceptable, although more problematic, for other static analyses such as nullness analysis, that determines which variables or fields might hold `null` at run-time. In this case, the extra complexity is due to the kind of worst-case assumption that must be made for the parameters of the entry points to the activities. Namely, they must always be assumed to be nullable, a correct but imprecise assumption that results in many false alarms. For this reason, not relying on a semantics such as that of this article, the nullness analysis for Android implemented in the Julia static analyzer [15] currently relies on manual annotations of the entry points, provided once and for all with the analyzer, that specify which parameters are really nullable. In other words, those annotations strengthen the worst-case assumption but must be provided *by hand*, a tedious and error-prone task.

A traditional static analysis, not based on our semantics, shows all its weakness when it comes to properties somehow related to the flow of data inside the program. For instance, information flow analysis determines if tainted (or *secret*) information can ever be accessed from a given variable. Without a semantics such as that of this article, the only sound assumption would be to consider all parameters of the entry points to the activities as potentially tainted. This imprecision would rapidly propagate to conclude that all variables are potentially tainted. That is, the static analysis would be so imprecise to be of no practical interest. In this specific case, one needs a semantics that models, explicitly, the lifecyle of the activities and their intercommunication, so that information is tracked while it flows through the code, in a much more precise way as the worst-case scenario.

Another situation, when a static analysis should be based on our semantics, is when the analysis must use precise information on how data is exchanged between activities. Namely, in Android data is passed from activity to activity through *bundles*, that are objects mapping string keys to tokens of information. When an activity starts, it assumes that data is passed in a way defined by the programmer: he specifies which string keys are used and which is the type of the tokens bound to the keys. If this convention is violated, activities stop at run-time with an exception. A static, compile-time verification of how information is stored inside the bundles would help the programmer and give him evidence that the program will not go wrong at run-time. This can only be achieved with a static analysis that models the communication mechanism between activities and is able to match caller activities with callee activities. Such a static analysis can then verify that the caller activities provide the expected bundles to the callee activities.

3. Preliminaries

3.1 The Android Platform

We assume the reader familiar with the Android platform (see [2] for a detailed description). Below, we recall some basic elements that will be used in this article.

Android programs are written in Java and compiled to the Google's Dalvik Virtual Machine (DVM) bytecode format before installation on a device. They can contain four types of 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) and *broadcast receivers* (objects reacting to broadcast messages). The components of a program can interact with each other and with components of other programs installed on the device. They are distinct entry points of the system into the program. During its lifetime, a component can have different successive states; the *lifecycle* of a component is the set of all its possible states together with the paths between the states. The interactions between components and between the system and the components lead to very complex situations whose semantics is hard to model. For simplicity, the scope of this paper is restricted to programs that only consist of activities.

The lifecycle of an activity [2] is depicted in Fig. 1, where the square rectangles represent the callback methods called by the system when the activity moves between states. The Android system manages activities in an *activity stack*. When a new activity starts, it is placed on top of the stack and comes to the foreground *i.e.,* it is visible to the user and has focus. The new activity is then said to be *running*. When the new activity completes (because it gets destroyed by the system that needs memory or because its `finish` method is called), it gets removed from the stack. The class `android.app.Activity` is provided by the Android API to implement the activities by subclassing. It contains several methods for modelling interactions *e.g.,* `startActivityForResult` launches a new activity and waits for a result, `setResult` sets the result that an activity will return to its caller and `finish` finishes an activity *i.e.,* indicates that it has completed and should be closed.

A hierarchy of *views* (visual objects) is associated to each activity. It provides the visual interface of the activity and is usually set up by *inflating* XML layout files *via* the method `setContentView`. The inflated views can be accessed from their *identifier* using the method `findViewById`.

As an example, consider the activity in Fig. 2. The `onCreate` event handler gets called when the activity is first created, after

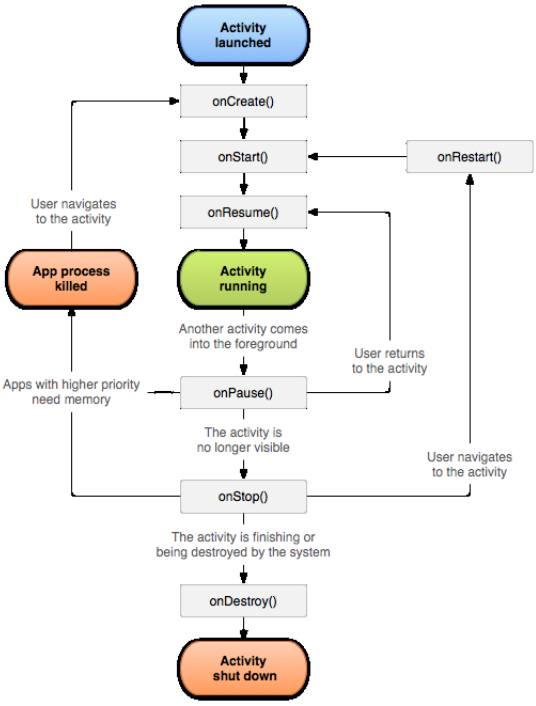

Figure 1. The lifecycle of an activity.

its constructor has been implicitly invoked by the Android system. The `setContentView` method calls the layout inflator; its integer parameter uniquely identifies the XML layout file shown in Fig. 3. From line 3 of the file in Fig. 3, it is clear that the view identified as `message` at line 10 of Fig. 2 belongs to the library view class `TextView`. The cast at line 10 in Fig. 2 is hence correct. Constants `R.layout.caller` and `R.id.message` are automatically generated at compile-time from the XML layout file names and from the view identifiers that they contain, respectively. The developer can call `setContentView` many times and everywhere in the code; he can pass the value of any integer expression to it and to `findViewById`, although the usual approach is to pass the compiler-generated constants.

Lines 5–6 of Fig. 3 describe a button that, when clicked, invokes a method with name `launchActivity`. When installed in the view hierarchy of an instance of `Caller`, the button invokes the method defined at lines 13–16 of Fig. 2. This method uses an *intent* for starting a new activity which is an instance of the class defined in Fig. 4. When the launched activity completes, the system invokes the method `onActivityResult` of the caller. This method uses its `requestCode` parameter to identify the kind of activity that sent a result. The integer constants `R.string.ok_clicked`, `R.string.cancel_clicked` and `R.string.unknown` used inside `onActivityResult` are generated at compile-time and they uniquely identify strings defined in the resource file `string.xml` of the application.

The layout of the activity in Fig. 4 is defined in the file shown in Fig. 5. It consists of two buttons, with labels `OK` and `Cancel`, that invoke methods `resultOk` and `resultCancel` at lines 9–12 and 14–19 of Fig. 4, respectively. These set the result of the activity and *finish* it.

In Fig. 2 and Fig. 4, we use `startActivityForResult` and `finish` for launching and finishing an activity, respectively. It is important to notice that these methods do not necessarily take effect *immediately*, at the point where they are invoked, since the method that calls them is allowed to reach its end before starting or finishing the activity. So, `launchActivity` in Fig. 2 outputs `Hello!` before launching the new activity and `resultCancel` in Fig. 4 outputs `Goodbye!` before stopping the activity. Another important point is the call to `setResult` at line 17 of Fig. 4: it does not modify the result set at line 15 above as it occurs after a call to `finish`.

Figure 6 shows the Dalvik bytecode of the `onActivityResult` method defined in Fig. 2. This method uses 6 registers (line 2) denoted by `v0`, `v1`, ..., `v5` (the number of registers used by a method is statically known [6]). It has 4 parameters (including `this`) whose values at the beginning of an execution are stored in the last 4 registers: `v2` stores the value of `this`, `v3` that of `requestCode`, `v4` that of `resultCode` and `v5` that of `data`.

3.2 Scope of this Article

We consider Android programs compiled into the DVM bytecode format. We assume that programs consist of activities only (no services, nor broadcast receivers, nor content providers). The activities can interact *via* methods `startActivityForResult`, `setResult` and `finish`, but we suppose that an activity can invoke one of these methods only when it is running (see the oval in the middle of Fig. 1); therefore, the calls to these methods cannot happen in the square rectangles of Fig. 1 (`onCreate`, `onStart`, ...) Moreover, the programs that we consider do not launch activities defined in other programs and name the called activities explicitly; that is, they explicitly specify the class of the activity to be started, as in line 14 of Fig. 2. We do not model the concurrency aspects of the Android platform. Note that sound static analyses for concurrent Java (and hence Android) are a complex topic of active research and solving this problem is well outside the scope of this article.

3.3 Notations

Throughout the paper, we use the underscore character '_' to denote any, non-significant, value. For any binary relation $\rightarrow$, we let $\rightarrow^*$ denote its reflexive and transitive closure. The *domain* and *codomain* of a function f are $dom(f)$ and $rng(f)$, respectively. We denote by $[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$ the function f where $dom(f) = \{v_1, \ldots, v_n\}$ and $f(v_i) = t_i$ for $i = 1, \ldots, n$. Its *update* is $f[w_1 \mapsto d_1, \ldots, w_m \mapsto d_m]$, where the domain may be enlarged (it is never reduced). An *array* r is a function whose domain is a finite subset of $\mathbb{N}$. An *index* is an element of $dom(r)$. For any index k, we write r^k instead of $r(k)$ and we say that r^k is the kth element, or the element at index k, of r. We let r^{min} denote the element of r at the least index. We write a stack in the form $x :: y :: z :: s$ meaning that x is the topmost value on the stack, y is the underlying element and z the element still below it; s is the remaining portion of the stack and might be empty. The empty stack is written ε. When s is empty, we may omit it and write $x :: y :: z$ instead of $x :: y :: z :: \varepsilon$.

4. Our Simplified Dalvik Virtual Machine

We start from the notion of activity *i.e.,* a user interface screen including interaction logic with the user.

DEFINITION 1. *A subclass of* `android.app.Activity` *is called an* activity class. *An* activity *is an instance of an activity class. Any activity has a field root, that contains the view hierarchy of the activity; a field finished, that states whether the activity is finished or not; a field result, that contains the result that the activity will return to its caller. An activity of an Android program P is an instance of an activity class defined in P. Any view object has a field id that stores the view identifier.*

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

  private final static int CALLEE = 0;
  private TextView mMessageView;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.caller); // tell system to use the layout defined in our XML file
      mMessageView = (TextView) findViewById(R.id.message); // get handles to the TextView from XML
  }

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

  @Override
  protected void onActivityResult(int requestCode, int resultCode, Intent data) {
      if (requestCode == CALLEE)
          if (resultCode == RESULT_OK) mMessageView.setText(R.string.ok_clicked);
          else if (resultCode == RESULT_CANCELED) mMessageView.setText(R.string.cancel_clicked);
          else mMessageView.setText(R.string.unknown);
      else  mMessageView.setText(R.string.unknown);
  }
}
```

Figure 2. A portion of the source code Android file `Caller.java`.

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

Figure 3. The XML layout file `caller.xml`.

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

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

  public void returnOk(View v) {
    setResult(RESULT_OK);
    finish();
  }

  public void returnCancel(View v) {
    setResult(RESULT_CANCELED);
    finish();
    setResult(RESULT_OK);
    System.out.println("Goodbye!");
  }
}
```

Figure 4. A portion of the source code Android file `Callee.java`.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent" >
    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/ok" android:onClick="returnOk" />
    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/cancel" android:onClick="returnCancel" />
</LinearLayout>
```

Figure 5. The XML layout file `callee.xml`.

```
protected onActivityResult(int, int, Intent):void
    .registers 6

    const v1, R.string.unknown

    if-nez v3, :end
    const/4 v0, 1
    add-int v0, v4, v0

    if-nez v0, :cancel
    const v1, R.string.ok_clicked
    goto :end

    :cancel
    if-nez v4, :end
    const v1, R.string.cancel_clicked

    :end
    iget-object v0, v2, mMessageView
    invoke-virtual {v0,v1},TextView.setText(int):void
    return-void
```

Figure 6. The Dalvik bytecode of the `onActivityResult` method defined in Fig. 2.

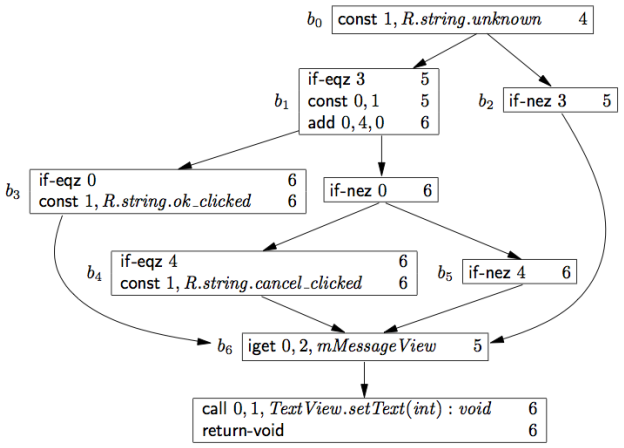

Figure 7. Our simplified Dalvik bytecode for the method `onActivityResult` in Fig. 6. On the right of each instruction, we report the number of registers at that program point, just before executing the instruction.

The state of the computation contains the registers in scope at a particular program point and the memory of the system. Inside the memory, there are objects, connected through pointers. To simplify the notation, we do not consider arrays nor interfaces and only allow integers as values of basic type.

DEFINITION 2. *The set of* values *is* $\mathbb{Z} \cup \mathbb{L}$*, where* $\mathbb{L}$ *is the set of* memory locations*. A* state *of the DVM is a triple* $\langle r \parallel \pi \parallel \mu \rangle$ *where* r *is an array of values, called* registers*,* π *is a stack of activity classes, called* pending activities*, and* μ *is a* memory*, or* heap*, that maps* locations *into* objects*. An object is a function that maps its fields (identifiers) into values and that embeds a class tag* κ*; we say that it* belongs to *class* κ *or that it is* an instance of *class* κ *or* has class κ*. We require that there are no dangling pointers i.e.,* $rng(r) \cap \mathbb{L} \subseteq dom(\mu)$ *and* $rng(\mu(\ell)) \cap \mathbb{L} \subseteq dom(\mu)$ *for every* $\ell \in dom(\mu)$*. We write* $o(f)$ *for the value of the field* f *of an object* o*; we write* $o(\kappa)$ *for the class of an object* o*. The set of all classes is denoted by* $\mathbb{K}$*. The set of all DVM states is denoted by* Σ*. When we want to fix the exact number* $\#r \in \mathbb{N}$ *of registers, we write* $\Sigma_{\#r}$*.*

EXAMPLE 1. *Let* $\mathbb{K} = \{A, B, C\}$*. Consider a memory* $\mu = [\ell_1 \mapsto o_1, \ell_2 \mapsto o_2, \ell_3 \mapsto o_3]$ *where* $o_1 = [f \mapsto \ell_2]$*,* $o_2 = [f \mapsto \ell_1]$ *and* $o_3 = [g \mapsto 0, h \mapsto 3]$*. Then a state is*

$$\sigma = \langle [0 \mapsto 5, 1 \mapsto \ell_2, 2 \mapsto \ell_3] \parallel C :: A \parallel \mu \rangle \, .$$

We have $\sigma \in \Sigma_3$ *since* σ *has* 3 *registers.*

The Dalvik bytecode is strongly typed. Each value has a type and registers are statically typed.

DEFINITION 3. *The set of* types *of our simplified DVM is* $\mathbb{T} = \mathbb{K} \cup \{\texttt{int}, \texttt{void}\}$*. The* `void` *type can only be used as the return type of methods. A method signature is denoted by* $\kappa.m(t_1, \ldots, t_p) : t$ *standing for a method named* m*, defined in class* κ*, expecting* p *explicit parameters of type, respectively,* $t_1, \ldots, t_p$ *and returning a value of type* t*, or returning no value when* $t =$ `void`*.*

In object-oriented languages, a non-static method $\kappa.m(t_1, \ldots, t_p) : t$ also has an *implicit* parameter of type κ called `this` inside the code of the method. Hence the actual number of parameters is $p+1$. We do not distinguish between methods and constructors. A constructor is just a method named `<init>` and returning `void`. We do not consider static fields and methods, but the extension of our definitions to them is not difficult.

Dalvik bytecode instructions work over states. Their execution affects the registers or the memory in the states. There are more than 200 Dalvik bytecode instructions [6]. Many are similar or only differ in the type or size of their operands. Hence we concentrate here on a restricted set of 12 instructions, which exemplify the operations that the DVM performs: register manipulation, arithmetics, object creation and access and method call. Below, when we say that the *computation stops*, we mean that an exception would be raised at run-time. Exception handling could be accomodated in our framework exactly as done for Java bytecode, by distinguishing normal from exceptional states (see for instance [21]). However, this would only complicate our semantics without adding anything to the semantics of activities that we want to provide in this article.

DEFINITION 4. *The set of instructions of our simplified Dalvik bytecode is the following (a formalisation of their semantics will be given later).*

const d, c *Move constant* c*, which can be an integer or a location, into the* d*th register.*

move d, s *Move the content of the* s*th register into the* d*th register.*

new-instance d, κ *Move a reference to a new object of class* κ *(which is properly initialised) into the* d*th register.*

add d, s_1, s_2 *Store the sum of the content of the* s_1*th register and of the* s_2*th register into the* d*th register.*

iget d, i, f *The content* r^i *of the* i*th register must be* 0 *(the equivalent of* `null` *in Java) or a reference to an object* o*. If* r^i *is* 0*, the computation stops. Otherwise,* $o(f)$ *is stored into the* d*th register.*

iput s, i, f *The content* r^i *of the* i*th register must be* 0 *or a reference to an object* o*. If* r^i *is* 0*, the computation stops. Otherwise, the content of the* s*th register is stored into* $o(f)$*.*

if-eqz i *Check if the content of the* i*th register is* 0*. If this is not the case, the computation stops.*

if-nez i *Check if the content of the* i*th register is* 0*. If this is the case, the computation stops.*

call $S, m_1, \ldots, m_n$ *where* $S = s_0, s_1, \ldots, s_p$ *is a sequence of register indexes and, for each* $1 \leq i \leq n$*,* m_i *has the form* $\kappa_i.m(t_1, \ldots, t_p) : t$*. The content* r^{s_0} *of the* s_0*th register, . . . ,* r^{s_p} *of the* s_p*th register are the* actual parameters *of the call.*

Value r^{s_0} is called receiver *of the call and must be* 0 *or a reference to an object o. In the former case, the computation stops. Otherwise, a lookup procedure is started from the class of o upwards along the superclass chain, looking for a method called m, expecting p* formal parameters *of type $t_1, \ldots, t_p$, respectively, and returning a value of type t. It is guaranteed that such a method is found in a class belonging to the set $\{\kappa_1, \ldots, \kappa_n\}$. That method is run from a state having an array of registers bound to $r^{s_0}, r^{s_1}, \ldots, r^{s_p}$.*

move-result d *Move the result of the most recent called method into the dth register. This instruction must immediately follow a* call *instruction.*

return-void *Return from a void method.*

return s *Return from a non-void method with the content of the sth register as result.*

We suppose that each move-result instruction immediately follows a call. We let ret denote an instruction in {return-void, return}. As Definition 4 states, the call instruction is decorated with an overapproximation of the set of its run-time targets. This is because Dalvik is an object-oriented language and the target of a method call is determined at run-time from the run-time class of the receiver. This overapproximation is not available in the original bytecode and must be computed through any kind of class analysis [18]. The situation here is not different from that of the static analysis of any object-oriented language, where late binding must be overapproximated by a class analysis before any interprocedural static analysis of the code can be performed. In particular, we have already used and implemented the same technique for Java bytecode [19]. Note that the exact algorithm used to lookup for the method implementation called at a call site is not relevant to this article. The class analysis that computes this overapproximation will know and use the detail of the lookup procedure of the language.

DEFINITION 5. *We also consider the following set of macro instructions which correspond to library methods typically used in Android applications.*

setContentView xml *Remove all the views from the view hierarchy of the current activity and replace them with new views created from the description contained in file xml.*

findViewById d, i *Search a view with identifier i in the view hierarchy of the current activity. If such a view exists, then move a reference to it into the dth register. Otherwise, move value* 0 *into the dth register.*

startActivityForResult A *Start a new activity of class A.*

setResult i *Set the result code of the current activity to i.*

finish *Set the finished field of the current activity to true.*

In the sequel of this article, we consider Android programs written with the instructions of Definitions 4–5. In order to reason about the control flow in the code, we assume that *flat* code, like that in Fig. 6, is given a structure in terms of blocks of code linked by arrows expressing how the flow of control passes from block to block. These might be for instance the *basic blocks* of [1], but we also require that a call instruction can only occur at the beginning of a block, a ret instruction can only occur as the last instruction of a block with no successor and blocks with no successor end with a ret instruction. For instance, Fig. 7 shows the blocks derived from the code of the method `onActivityResult` in Fig. 6.

From now on, an *Android program* will be a graph of blocks; inside each block there is one or more instructions among those described above. This graph typically contains many disjoint subgraphs, each corresponding to a different method. The ends of a method, where the control flow returns to the caller, are the end of the blocks with no successor. A conditional branch results in a block followed by two blocks, each starting with a conditional bytecode that selects the right branch of execution on the basis of the state of the DVM.

DEFINITION 6. *We write a block containing w bytecode instructions and having x immediate successor blocks $b_1, \ldots, b_x$, with $w > 0$ and $x \geq 0$, as*

$$\boxed{\begin{array}{c}\mathsf{ins}_1\\ \mathsf{ins}_2\\ \cdots\\ \mathsf{ins}_w\end{array}} \rightrightarrows \begin{array}{c}b_1\\ \cdots\\ b_x\end{array} \qquad \textit{or just as} \qquad \boxed{\begin{array}{c}\mathsf{ins}_1\\ \mathsf{ins}_2\\ \cdots\\ \mathsf{ins}_w\end{array}} \qquad \textit{when } x = 0.$$

An Android program *P is a graph of such blocks.*

DEFINITION 7. *We let $b_{\kappa.m(t_1,\ldots,t_p):t}$ denote the block of code where the method $\kappa.m(t_1, \ldots, t_p) : t$ starts.*

We assume that Dalvik bytecode is verified. For instance, instructions accessing a field may only occur over 0 or over a receiver that actually contains that field. As for Java bytecode, real Dalvik bytecode must pass a verification check before being run on the DVM [6].

5. An Operational Semantics

We define here our operational semantics for Dalvik. The main point is that each bytecode or block of code is modelled as a state transformer. This is important to pave the way to static analysis and in particular to abstract interpretation, since there will be only a single concrete domain to abstract, that of states, in order to get an abstract semantics of Dalvik. State transformers are partially defined maps, that we write through the λ-notation. They are undefined when the DVM cannot proceed, for instance because of a null-pointer exception; they are also undefined for conditional bytecodes, expressing the fact that the computation continues only on the right branch.

DEFINITION 8. *In Fig. 8 and Fig. 9, each instruction or macro instruction* ins *different from* call, move-result, return-void *and* return, *occurring at a program point q, is associated with its* semantics *$ins_q : \Sigma_{r_i} \to \Sigma_{r_o}$ at q, where r_i, r_o are the number of registers defined at q and at the subsequent program point(s), respectively (this information is statically known [6]).*

EXAMPLE 2. *Consider the state*

$$\sigma = \langle [0 \mapsto 5, 1 \mapsto \ell_2, 2 \mapsto \ell_3] \,\|\, C :: A \,\|\, \mu \rangle \in \Sigma_3$$

of Example 1. We have

$$(const\ 0, 3)(\sigma) = \langle [0 \mapsto 3, 1 \mapsto \ell_2, 2 \mapsto \ell_3] \,\|\, C :: A \,\|\, \mu \rangle \in \Sigma_3$$
$$(iget\ 0, 1, f)(\sigma) = \langle [0 \mapsto \ell_1, 1 \mapsto \ell_2, 2 \mapsto \ell_3] \,\|\, C :: A \,\|\, \mu \rangle \in \Sigma_3$$
$$(iput\ 1, 2, g)(\sigma) = \langle [0 \mapsto 5, 1 \mapsto \ell_2, 2 \mapsto \ell_3] \,\|\, C :: A \,\|\, \mu' \rangle \in \Sigma_3$$

with $\mu' = [\ell_1 \mapsto o_1, \ell_2 \mapsto o_2, \ell_3 \mapsto o'_3]$ and $o'_3 = [g \mapsto \ell_2, h \mapsto 3]$. We also have

$$(move\ 6, 2)(\sigma) = \langle [0 \mapsto 5, 1 \mapsto \ell_2, 2 \mapsto \ell_3, 6 \mapsto \ell_3] \,\|\, C :: A \,\|\, \mu \rangle \in \Sigma_4$$
$$(startActivityForResult\ B)(\sigma) = \langle [0 \mapsto 5, 1 \mapsto \ell_2, 2 \mapsto \ell_3] \,\|\, B :: C :: A \,\|\, \mu \rangle \in \Sigma_3\ .$$

5.1 Method Invocation and Execution

The state transformers of Definition 8 define the operational semantics of each single instruction and macro instruction different from call, move-result, return-void and return. The semantics of call is more difficult to define, since it performs many operations:

1. creation of a new state for the callee, copying the content of the registers of the caller that hold the actual arguments of the call into the last registers of the callee;

$$const_q\ d, c = \lambda\langle r \parallel \pi \parallel \mu\rangle.\langle r[d \mapsto c] \parallel \pi \parallel \mu\rangle$$

$$move_q\ d, s = \lambda\langle r \parallel \pi \parallel \mu\rangle.\langle r[d \mapsto r^s] \parallel \pi \parallel \mu\rangle$$

$$new\text{-}instance_q\ d, \kappa = \lambda\langle r \parallel \pi \parallel \mu\rangle.\langle r[d \mapsto \ell] \parallel \pi \parallel \mu[\ell \mapsto o]\rangle$$

where ℓ is a fresh location and o is an object of class κ whose fields hold 0

$$add_q\ d, s_1, s_2 = \lambda\langle r \parallel \pi \parallel \mu\rangle.\langle r[d \mapsto r^{s_1} + r^{s_2}] \parallel \pi \parallel \mu\rangle$$

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

$$iput_q\ s, i, f = \lambda\langle r \parallel \pi \parallel \mu\rangle.\begin{cases}\langle r \parallel \pi \parallel \mu[r^i \mapsto \mu(r^i)[f \mapsto r^s]]\rangle & \text{if } r^i \neq 0\\ undefined & \text{otherwise}\end{cases}$$

$$if\text{-}eqz_q\ i = \lambda\langle r \parallel \pi \parallel \mu\rangle.\begin{cases}\langle r \parallel \pi \parallel \mu\rangle & \text{if } r^i = 0\\ undefined & \text{otherwise}\end{cases}$$

$$if\text{-}nez_q\ i = \lambda\langle r \parallel \pi \parallel \mu\rangle.\begin{cases}\langle r \parallel \pi \parallel \mu\rangle & \text{if } r^i \neq 0\\ undefined & \text{otherwise}\end{cases}$$

Figure 8. Semantics of the Dalvik bytecode instructions that we consider.

$$setContentView_q\ xml = \lambda\langle r \parallel \pi \parallel \mu\rangle.\langle r \parallel \pi \parallel \mu[\ell \mapsto \mu(\ell)[root \mapsto \{v_x \mid x \in xml\}]]\rangle$$

where xml is a set of view descriptions
and κ_x is the class tag in a view description x
and v_x is a new instance of κ_x whose fields are set according to x

$$findViewById_q\ d, i = \lambda\langle r \parallel \pi \parallel \mu\rangle.\begin{cases}\langle r[d \mapsto \ell_v] \parallel \pi \parallel \mu\rangle & \text{if there exists a view } v \text{ in } \mu(\ell)(root)\\ & \text{held at location } \ell_v \text{ in } \mu \text{ and } v(id) = i\\ \langle r[d \mapsto 0] \parallel \pi \parallel \mu\rangle & \text{otherwise}\end{cases}$$

$$startActivityForResult_q\ A = \lambda\langle r \parallel \pi \parallel \mu\rangle.\langle r \parallel A :: \pi \parallel \mu\rangle$$

$$setResult_q\ i = \lambda\langle r \parallel \pi \parallel \mu\rangle.\begin{cases}\langle r \parallel \pi \parallel \mu\rangle & \text{if } \mu(\ell)(\textit{finished}) = true\\ \langle r \parallel \pi \parallel \mu[\ell \mapsto \mu(\ell)[result \mapsto i]]\rangle & \text{otherwise}\end{cases}$$

$$finish_q = \lambda\langle r \parallel \pi \parallel \mu\rangle.\langle r \parallel \pi \parallel \mu[\ell \mapsto \mu(\ell)[\textit{finished} \mapsto true]]\rangle$$

Figure 9. Semantics of the macro instructions that we consider. Location ℓ is a reference to the current activity.

2. lookup of the dynamic target method on the basis of the run-time class of the receiver;
3. execution of the dynamic target method and return.

We model (1) and (2) as state transformers, and (3) as the creation of a new frame for the callee, followed by the rehabilitation of the frame of the caller.

The first operation is formalised as follows.

DEFINITION 9. *Let q be a program point where a call to a method $\kappa.m(t_1, \ldots, t_p) : t$ occurs. Let r_q be the number of registers at q and r_m be the number of registers used by $\kappa.m(t_1, \ldots, t_p) : t$. Let $S = s_0, s_1, \ldots, s_p$ be a sequence of register indexes. We define*

$$args_{q,S,\kappa.m(t_1,\ldots,t_p):t} \in \Sigma_{r_q} \to \Sigma_{p+1}$$

as

$$args_{q,S,\kappa.m(t_1,\ldots,t_p):t} = \lambda\langle r \parallel \pi \parallel \mu\rangle.\langle[i \mapsto r^{s_{i-k}} \mid k \leq i < r_m] \parallel \pi \parallel \mu\rangle$$

where $k = r_m - (p + 1)$.

EXAMPLE 3. *Consider $m =$ `onActivityResult` defined in Fig. 6. We have $p = 3$ and $r_m = 6$. If q is a program point where a call to* `onActivityResult` *occurs and $S = 3, 0, 0, 2$,*

$$args_{q,S,\texttt{onActivityResult}} = \lambda\langle r \parallel \pi \parallel \mu\rangle.\langle[2 \mapsto r^3, 3 \mapsto r^0, 4 \mapsto r^0, 5 \mapsto r^2] \parallel \pi \parallel \mu\rangle\ .$$

The second operation is formalised as a *filter* state transformer, put at the beginning of the code of each method $\kappa_i.m(t_1, \ldots, t_p) : t$. It checks if that method is actually selected at run-time, on the basis of the dynamic class of the receiver and of the method lookup procedure of the language.

DEFINITION 10. *Let $\kappa.m(t_1, \ldots, t_p) : t$ be a method. We define*

$$select_{\kappa.m(t_1,\ldots,t_p):t} : \Sigma_{p+1} \to \Sigma_{p+1}$$

as

$$\lambda\langle r \parallel \pi \parallel \mu\rangle.\begin{cases}\langle r \parallel \pi \parallel \mu\rangle & \text{if } r^{min} \neq 0 \text{ and the lookup procedure of a method } m(t_1,\ldots,t_p):t \text{ from the class of } \mu(r^{min}) \text{ selects its implementation in class } \kappa \\ \mathit{undefined} & \text{otherwise.}\end{cases}$$

We do not specify the detail of the lookup procedure used in Definition 10 since it would only complicate the definition of our concrete semantics. Note that we have already assumed that call sites are decorated with an overapproximation of the set of their possible targets, hence any abstraction of our concrete semantics will not use a formal definition of the lookup procedure but will overapproximate the concrete execution paths by using the decoration available at call sites.

DEFINITION 11. *A* method frame *(or* frame*) is a pair* $\langle b \parallel r\rangle$ *of a block* b *of the program and registers* r*. It represents the fact that the DVM is going to execute* b *with registers* r*. A* method stack *is a stack* $c_1 :: c_2 :: \cdots :: c_n$ *of method frames.*

DEFINITION 12. *A* method configuration *(or* configuration*) is a triple* $\alpha \diamond \pi \diamond \mu$ *of a method stack* α*, pending activities stack* π *and memory* μ*. The (small step) operational semantics for method execution in a program* P *is the binary relation* $\rightsquigarrow_P$ *(*P *is usually omitted) over configurations defined by the rules in Fig. 10.*

In Fig. 10, rule (1a) executes an instruction or macro instruction ins, different from call, move-result, return-void and return, by calling its semantics ins. The DVM moves then forward to run the rest of the instructions. Rule (1b) calls a method. It chooses one of the possible callees, looks for the block b_{m_i} where the latter starts and builds its initial state $\sigma' = \langle r' \parallel \pi' \parallel \mu'\rangle$, by using the functions $args$ and $select$. It creates a new current frame containing b_{m_i} and r'. It removes the call from the instructions still to be executed at return time. The choice of the possible callee, as expressed in our semantics, is non-deterministic. However, only one possible callee will be selected at run-time and only one non-deterministic choice will continue, since the $select$ function of the method lookup rules of the language is deterministic in the case of Dalvik. For all other callees, σ' does not exist ($select$ is a partial function). Control returns to the caller by rules (1c) and (1d), that rehabilitate the frame of the caller. Rule (1c) corresponds to the situation when the call instruction is followed by a move-result; the return value of the callee is stored in the specified register of the caller. In contrast, rule (1d) corresponds to the situation when no move-result follows the call instruction; the return value of the callee, if any, is lost. Rule (1e) applies when all instructions inside a block have been executed; it runs one of its immediate successors, if any. This rule is normally deterministic, since if a block of the Dalvik bytecode has two or more immediate successors then they start with mutually exclusive conditional instructions and only one thread of control is actually followed.

We state now the usual *preservation* and *progress* theorems for the semantics of Fig. 10.

THEOREM 1 (Preservation). *If* $\alpha\diamond\pi\diamond\mu$ *is well-formed and* $\alpha\diamond\pi\diamond\mu \rightsquigarrow \alpha'\diamond\pi'\diamond\mu'$*, then* $\alpha'\diamond\pi'\diamond\mu'$ *is well-formed.*

Successful configurations correspond to the situations when the end of a callback method is reached. This happens when the method stack has the form $\langle \boxed{\mathsf{ret}} \parallel \_\rangle :: \varepsilon$ *i.e.,* consists of exactly one frame whose block has no successor and contains just a single ret instruction. None of the rules in Fig. 10 applies to such stacks: in rules (1a) and (1b) the block of the topmost frame does not start with a ret instruction, in rules (1c) and (1d) the method stack consists of at least two elements and in rule (1e) the block of the topmost frame is empty. Note that the semantics of Fig. 10 is *partial*: the program may get stuck. Some of this partiality is related to well-formedness of the Android programs that we consider. For instance, iget d, i, f is not defined if the content of the ith register is not 0 nor a reference to an object. This kind of errors is ruled out by the static well-formedness constraints imposed in Section 4. In other cases, partiality stands for certain failure modes for which the DVM specification says that the behavior of the program is undefined, for instance an iget d, i, f instruction where $r^i = 0$ (access to a field over a `null` reference); we model these events by stuck configurations.

DEFINITION 13. *We let* $\overline{\alpha}$*,* $\overline{\alpha}'$*,* $\overline{\alpha}_1$*, ... denote method stacks of the form* $\langle \boxed{\mathsf{ret}} \parallel \_\rangle :: \varepsilon$*. A method configuration is* successful *if it has the form* $\overline{\alpha}\diamond\pi\diamond\mu$*. A method configuration is* stuck *if it has the form*

$$\langle \boxed{\substack{\mathsf{ins}\\ rest}} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_x \end{smallmatrix} \parallel r\rangle :: \alpha \diamond \pi \diamond \mu$$

with $ins(\langle r \parallel \pi \parallel \mu\rangle) = \mathit{undefined}$ *(see Fig. 8).*

THEOREM 2 (Progress). *If* $\alpha \diamond \pi \diamond \mu$ *is well-formed, then either it is successful, or it is stuck (hence in an erroneous configuration) or there exists* $\alpha' \diamond \pi' \diamond \mu'$ *such that* $\alpha \diamond \pi \diamond \mu \rightsquigarrow \alpha' \diamond \pi' \diamond \mu'$*.*

5.2 Activities

Activities can be in any of the states depicted in Fig. 1.

DEFINITION 14. *An* activity state *is an element of the set*

$$\{\ \mathit{constructor}, \mathit{onCreate}, \mathit{onStart}, \mathit{onRestart}, \mathit{onResume}, \mathit{running}, \mathit{onActivityResult}, \mathit{onPause}, \mathit{onStop}, \mathit{onDestroy}\ \}\ .$$

We let Lifecycle be the binary relation over activity states consisting of the pairs:

$$\begin{array}{l}(\mathit{constructor}, \mathit{onCreate}),\\ (\mathit{onCreate}, \mathit{onStart}), (\mathit{onRestart}, \mathit{onStart}),\\ (\mathit{onStart}, \mathit{onResume}), (\mathit{onStart}, \mathit{onStop}),\\ (\mathit{onResume}, \mathit{onPause}), (\mathit{onResume}, \mathit{running}),\\ (\mathit{running}, \mathit{onPause}), (\mathit{running}, \mathit{running}),\\ (\mathit{onPause}, \mathit{onResume}), (\mathit{onPause}, \mathit{onStop}),\\ (\mathit{onStop}, \mathit{onRestart}), (\mathit{onStop}, \mathit{onDestroy})\ .\end{array}$$

Let a be an activity at memory location ℓ. For any activity state s different from $constructor$ and $running$, we let $\ell.s$ denote the implementation of the callback method s found by method lookup from the class of a. For instance, $\ell.onCreate$ denotes the method $onCreate$ found by method lookup from the class of a. We let $\ell.constructor$ denote the default constructor of a. We let $\ell.running$ denote any callback method, different from those in Fig. 1 and from `onActivityResult`, found by method lookup from the class of a. For instance, if a is an object of the class `Caller` shown in Fig. 2, then $\ell.running$ may denote the method `launchActivity` defined in `Caller`. If a is an object of the class `Callee` shown in Fig. 4, then $\ell.running$ may denote the method `returnOk` or `returnCancel` defined in `Callee`.

DEFINITION 15. *Let* ℓ *be the memory location of an activity and* s *be an activity state. We let* $\alpha_{\ell.s}$ *denote a method stack of the form* $\langle b_{\ell.s} \parallel r\rangle :: \varepsilon$ *where*

$$r = [r_{\ell.s} - (p+1) \mapsto \ell, r_{\ell.s} - p \mapsto u_1, \ldots, r_{\ell.s} - 1 \mapsto u_p]$$

and $r_{\ell.s}$ *is the number of registers used by the method* $\ell.s$*,* $p+1$ *is the number of parameters of* $\ell.s$ *(including* `this`*) and* $u_1, \ldots u_p$ *is a sequence of values that the system passes to* $\ell.s$*.*

We will use $\alpha_{\ell.s}$ in the rules defining our operational semantics for modelling a callback method invocation: $\ell.s$ is a callback method,

$$\frac{\mathsf{ins} \notin \{\mathsf{call}, \mathsf{move\text{-}result}, \mathsf{return\text{-}void}, \mathsf{return}\} \quad \langle r' \,\|\, \pi' \,\|\, \mu' \rangle = ins(\langle r \,\|\, \pi \,\|\, \mu \rangle)}{\langle \boxed{\begin{smallmatrix}\mathsf{ins}\\ rest\end{smallmatrix}} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_x \end{smallmatrix} \,\|\, r \rangle :: \alpha \diamond \pi \diamond \mu \rightsquigarrow \langle \boxed{rest} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_x \end{smallmatrix} \,\|\, r' \rangle :: \alpha \diamond \pi' \diamond \mu'} \tag{1a}$$

$$\frac{\begin{array}{c} b = \boxed{\begin{smallmatrix}\mathsf{call}\ S, m_1, \ldots, m_n\\ rest\end{smallmatrix}} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_x \end{smallmatrix} \quad b' = \boxed{rest} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_x \end{smallmatrix} \\ \text{the call instruction occurs at program point } q \\ \langle r' \,\|\, \pi' \,\|\, \mu' \rangle = select_{m_i}(args_{q,S,m_i}(\langle r \,\|\, \pi \,\|\, \mu \rangle)) \end{array}}{\langle b \,\|\, r \rangle :: \alpha \diamond \pi \diamond \mu \rightsquigarrow \langle b_{m_i} \,\|\, r' \rangle :: \langle b' \,\|\, r \rangle :: \alpha \diamond \pi' \diamond \mu'} \tag{1b}$$

$$\frac{b = \boxed{\begin{smallmatrix}\mathsf{move\text{-}result}\ d\\ rest\end{smallmatrix}} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_x \end{smallmatrix} \quad b' = \boxed{rest} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_x \end{smallmatrix}}{\langle \boxed{\mathsf{return}\ s} \,\|\, r \rangle :: \langle b \,\|\, r' \rangle :: \alpha \diamond \pi \diamond \mu \rightsquigarrow \langle b' \,\|\, \langle r'[d \mapsto r^s] \rangle :: \alpha \diamond \pi \diamond \mu} \tag{1c}$$

$$\frac{b \text{ does not start with a } \mathsf{move\text{-}result}}{\langle \boxed{\mathsf{ret}} \,\|\, r \rangle :: \langle b \,\|\, r' \rangle :: \alpha \diamond \pi \diamond \mu \rightsquigarrow \langle b \,\|\, r' \rangle :: \alpha \diamond \pi \diamond \mu} \tag{1d}$$

$$\frac{1 \leq i \leq x}{\langle \square \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_x \end{smallmatrix} \,\|\, r \rangle :: \alpha \diamond \pi \diamond \mu \rightsquigarrow \langle b_i \,\|\, r \rangle :: \alpha \diamond \pi \diamond \mu} \tag{1e}$$

Figure 10. Operational semantics of Dalvik.

$b_{\ell.s}$ is the block where $\ell.s$ starts and the last $p+1$ registers of $\ell.s$ are bound to ℓ (the receiver of the call) followed by some values passed by the Android system to the method.

The next definition models the notion of an activity stack used by the Android system for managing activities.

DEFINITION 16. *An* activity frame *is a tuple $\langle \ell \,\|\, s \,\|\, \pi \,\|\, \alpha \rangle$ where ℓ is a memory location, s is an activity state, π is a stack of pending activities and α is a method stack. An* activity stack *is a stack $\varphi_1 :: \varphi_2 :: \cdots :: \varphi_n$ of activity frames where at most one frame is underlined.*

An activity frame $\langle \ell \,\|\, s \,\|\, \pi \,\|\, \alpha \rangle$ is a data structure used to manage an activity a in an activity stack; ℓ is the location of a in the heap, s is the current state of a, π contains the activities that are waiting to be launched from a and α is a method stack used for managing the execution of a callback method corresponding to s. We will use underlined frames for denoting situations when the execution of a callback method has to be given high priority.

We can define now the *operational semantics* of a program.

DEFINITION 17. *An* activity configuration *(or* configuration*) is a pair $\Omega \diamond \mu$ of an activity stack Ω and memory μ. The (small step) operational semantics of an Android program P is the binary relation $\Rightarrow_P$ (P is usually omitted) over configurations defined in Fig. 11–12.*

An activity stack Ω in Definition 16 is meant to contain only activities of the program P under analysis. Hence, Ω corresponds to the portions of the DVM activity stack that consist of activities of P. Therefore, although the topmost activity of the DVM stack is always in state $running$ (it is visible to the user and has focus), the topmost element of Ω may for instance be in state $onPause$. This corresponds to the situation when an activity of another program has come into the foreground: it does not appear in Ω because it is not an activity of P. Moreover, each Android program P runs in its own process within its own virtual machine, in isolation from other applications. Consequently, P has its own heap memory which is not shared with other applications. In a step $\Omega \diamond \mu \Rightarrow_P \Omega' \diamond \mu'$ of the operational semantics, μ represents the state of the memory assigned to the process of P and μ' is the state of this memory after performing the step. Note that μ cannot be affected by other applications.

The following *preservation* and *progress* theorems hold for the semantics of Fig. 11–12.

THEOREM 3 (Preservation). *If $\Omega \diamond \mu$ is well-formed and $\Omega \diamond \mu \Rightarrow \Omega' \diamond \mu'$, then $\Omega' \diamond \mu'$ is well-formed.*

Successful configurations correspond to the situations when the program terminates successfully. This happens when all the activities of the program have run and been removed from the activity stack, which has become empty. Stuck configurations correspond to the situations when the program is stuck in the middle of a method because the next instruction to be executed is undefined on the current state (Definition 13).

DEFINITION 18. *An activity configuration is* successful *if it has the form $\varepsilon \diamond \mu$. An activity configuration is* stuck *if it has the form $\Omega :: \underline{\langle \ell \,\|\, s \,\|\, \pi \,\|\, \alpha \rangle} :: \Omega' \diamond \mu$ where $\alpha \diamond \pi \diamond \mu$ is stuck.*

THEOREM 4 (Progress). *If $\Omega \diamond \mu$ is well-formed, then either it is successful, or it is stuck (hence in an erroneous configuration) or there exists $\Omega' \diamond \mu'$ such that $\Omega \diamond \mu \Rightarrow \Omega' \diamond \mu'$.*

5.3 Lifecycle Moves

Our rules for modelling the lifecyle of an activity are presented in Fig. 11. We do not model the situations when, in order to save memory space, the system decides to kill an entire application process or an individual, non-finished, activity. In such cases, the killed application, or single activity, is restarted when needed and restored to its previously saved state. The destruction of a single activity is an extremely rare situation which may happen when the application process has a large number of activities running in it. Modulo this restriction, the rules of Fig. 11 are intended to model a *superset* of the possible concrete traces of an Android program.

Rule (2a) models the execution of a callback method in the frame which is underlined. It cannot be used when the callback

$$\frac{\alpha \diamond \pi \diamond \mu \rightsquigarrow \alpha' \diamond \pi' \diamond \mu'}{\Omega :: \underline{\langle \ell \parallel s \parallel \pi \parallel \alpha \rangle} :: \Omega' \diamond \mu \Rightarrow \Omega :: \underline{\langle \ell \parallel s \parallel \pi' \parallel \alpha' \rangle} :: \Omega' \diamond \mu'} \tag{2a}$$

$$\frac{}{\Omega :: \underline{\langle \ell \parallel s \parallel \pi \parallel \overline{\alpha} \rangle} :: \Omega' \diamond \mu \Rightarrow \Omega :: \langle \ell \parallel s \parallel \pi \parallel \overline{\alpha} \rangle :: \Omega' \diamond \mu} \tag{2b}$$

$$\frac{\begin{array}{c}(s, s') \in \mathit{Lifecycle} \qquad \pi \neq \varepsilon \Rightarrow (s, s') = (\mathit{running}, \mathit{onPause}) \\ \mu(\ell)(\mathit{finished}) = \mathit{true} \Rightarrow (s, s') \in \{(\mathit{running}, \mathit{onPause}), (\mathit{onPause}, \mathit{onStop}), (\mathit{onStop}, \mathit{onDestroy})\}\end{array}}{\langle \ell \parallel s \parallel \pi \parallel \overline{\alpha} \rangle :: \Omega \diamond \mu \Rightarrow \underline{\langle \ell \parallel s' \parallel \pi \parallel \alpha_{\ell.s'} \rangle} :: \Omega \diamond \mu} \tag{2c}$$

$$\frac{\mu(\ell)(\mathit{finished}) = \mathit{true}}{\Omega :: \langle \ell \parallel \mathit{onDestroy} \parallel \_ \parallel \overline{\alpha} \rangle :: \Omega' \diamond \mu \Rightarrow \Omega :: \Omega' \diamond \mu} \tag{2d}$$

$$\frac{\mu' = \mu[\ell \mapsto \mu(\ell)[\mathit{finished} \mapsto \mathit{true}]]}{\langle \ell \parallel \mathit{running} \parallel \varepsilon \parallel \overline{\alpha} \rangle :: \Omega \diamond \mu \Rightarrow \langle \ell \parallel \mathit{running} \parallel \varepsilon \parallel \overline{\alpha} \rangle :: \Omega \diamond \mu'} \tag{2e}$$

$$\frac{\ell' \text{ is a fresh location and } o \text{ is a new object of class } \mu(\ell)(\kappa) \quad \mu' = \mu[\ell' \mapsto o]}{\langle \ell \parallel \mathit{onDestroy} \parallel \pi \parallel \overline{\alpha} \rangle :: \Omega \diamond \mu \Rightarrow \underline{\langle \ell' \parallel \mathit{constructor} \parallel \pi \parallel \alpha_{\ell'.\mathit{constructor}} \rangle} :: \Omega \diamond \mu'} \tag{2f}$$

$$\frac{\varphi = \langle \ell \parallel s \parallel \pi \parallel \overline{\alpha} \rangle \quad s \in \{\mathit{onResume}, \mathit{onPause}\} \quad (s', s_1') \in \{(\mathit{onPause}, \mathit{onStop}), (\mathit{onStop}, \mathit{onDestroy})\}}{\varphi :: \Omega :: \langle \ell' \parallel s' \parallel \pi' \parallel \overline{\alpha}' \rangle :: \Omega' \diamond \mu \Rightarrow \varphi :: \Omega :: \underline{\langle \ell' \parallel s_1' \parallel \pi' \parallel \alpha_{\ell'.s_1'} \rangle} :: \Omega' \diamond \mu} \tag{2g}$$

Figure 11. Lifecycle moves. In rules (2c)–(2g), it is supposed that the activity stack on the left of $\Rightarrow$ contains no underlined frame.

method has been run to completion *i.e.,* when the activity stack has the form $\Omega :: \underline{\langle \ell \parallel s \parallel \pi \parallel \overline{\alpha} \rangle} :: \Omega'$, because no rule in Fig. 10 applies to $\overline{\alpha}$. Rule (2b) models the situation when the callback method in the underlined frame has been run to completion: the frame is not given high priority anymore (it loses its underline) and one of the rules (2c)–(2g) can now be applied. Rule (2c) models the transition from a state s to one of its successors s' in $\mathit{Lifecycle}$. The end of a callback method corresponding to s has been reached and now it is possible to jump to a successor state and execute a corresponding callback method (the topmost activity frame gets underlined). Note that if pending activities exist (resulting from the execution of a startActivityForResult instruction in state $\mathit{running}$), then the only possibility is to switch to state $\mathit{onPause}$; the pending activities then have to be launched, which is modelled by the rules in Sect. 5.4. Moreover, if the activity is finished (due to the execution of a finish instruction in state $\mathit{running}$), then the only possible ways are those leading to state $\mathit{onDestroy}$. Rule (2d) models the removal of a finished activity from the stack. Rule (2e) models the situation when the activity is running and the user hits the back button of the device. Then, the activity gets finished. It will reach its state $\mathit{onPause}$ with (2a)–(2c) and will transfer control to its parent with the rules in Sect. 5.4. Rule (2f) models a configuration change, such as for instance a screen orientation switch. The foreground activity has reached its state $\mathit{onDestroy}$ with (2a)–(2c); now, it gets destroyed and replaced by a new activity. Rule (2g) models the situation when a new activity (represented by the frame φ) has been started, has come into the foreground and now completely hides a previous activity (represented by the frame $\langle \ell' \parallel s' \parallel \pi' \parallel \overline{\alpha}' \rangle$) which is no longer visible. The hidden activity switches state and is ready to execute the callback method corresponding to its new state *i.e.,* `onStop` or `onDestroy`.

Note that different rules may apply to the same stack *e.g.,* (2c) and (2g), or the same rule may apply to different portions of the same stack *e.g.,* (2d), or the same rule may be applied differently to the same stack *e.g.,* (2c) allows one to switch from state $\mathit{running}$ to itself starting *any* allowed callback method, or to switch from $\mathit{running}$ to $\mathit{onPause}$. Hence, the choice and the application of the rules is highly non-deterministic. This reflects the event-driven nature of Android applications and their tight, intricate interactions with the system.

EXAMPLE 4. *Consider the program in Fig. 2–5. Suppose that the activity described in Fig. 2 appears in the launcher screen of the device. When the user taps the corresponding icon, the application starts with a memory μ and an activity stack*

$$\Omega = \underline{\langle \ell \parallel \mathit{constructor} \parallel \varepsilon \parallel \alpha_{\ell.\mathit{constructor}} \rangle}$$

where $\mu(\ell)$ is an object of class `Caller`*. Then, using* (2a)–(2c) *as many times as necessary, one gets*

$$\Omega \diamond \mu \Rightarrow^* \underbrace{\langle \ell \parallel \mathit{onResume} \parallel \varepsilon \parallel \overline{\alpha}_1 \rangle}_{\Omega_1} \diamond \mu_1$$

i.e., the default constructor in class `Caller` *and the callback methods* `onCreate`*,* `onStart` *and* `onResume` *have been run to completion. Let*

$$\alpha_2 = \langle b_{\mathit{Caller.launchActivity(View):void}} \parallel r \rangle :: \varepsilon$$

where $r = [\#r - 2 \mapsto \ell, \#r - 1 \mapsto v]$, $\#r$ is the number of registers used by $\mathit{Caller.launchActivity(View)} : \mathit{void}$ and v is the location of a view. We have $\alpha_2 \in \alpha_{\ell.\mathit{running}}$, hence, using (2c)*,*

$$\Omega_1 \diamond \mu_1 \Rightarrow \overbrace{\underline{\langle \ell \parallel \mathit{running} \parallel \varepsilon \parallel \alpha_2 \rangle}}^{\Omega_2} \diamond \mu_2$$

and, using (2a)–(2b)*,*

$$\Omega_2 \diamond \mu_2 \Rightarrow^* \underbrace{\langle \ell \parallel \mathit{running} \parallel \mathit{Callee} \parallel \overline{\alpha}_3 \rangle}_{\Omega_3} \diamond \mu_3$$

i.e., the method `launchActivity` *is run, which contains an occurrence of* startActivityForResult *for invoking an instance of class* `Callee`*. The derivation* $\Omega \diamond \mu \Rightarrow^* \Omega_3 \diamond \mu_3$ *corresponds to the situation when an instance of* `Caller` *has started and then the user has tapped the* `Launch activity` *button.*

5.4 Starting and Returning from an Activity

Let P be an Android program. We assume that each activity a of P has a field *parent* that stores 0 if a was created from another program or stores a reference to the activity that started a otherwise. The rules for modelling the invocation of an activity are presented in Fig. 12.

Rule (3a) corresponds to the launch of a pending activity. The parent activity, represented by $\langle \ell \parallel s \parallel A :: \pi \parallel \overline{\alpha} \rangle$, is losing focus, hence it has run its method $onPause$. The new activity, represented by $\langle \ell' \parallel \mathit{constructor} \parallel \varepsilon \parallel \alpha_{\ell'.\mathit{constructor}} \rangle$, comes into the foreground, so it lays on top of the activity stack and its default constructor is about to be run.

When an activity a' of the program P under analysis finishes, the next situations may occur.

- Either a' was started from another program P'. Then, it returns its result to P' which is out of the scope of the analysis. This situation is handled by (2c) and (2d): first, a' reaches the state $onDestroy$, then it is removed from the activity stack.
- Or a' was started from an activity a of P. Then, a lies in the activity stack, just below a'. This situation is handled by (3b) and (3c) where a (resp. a') is represented by the frame φ (resp. φ'). In (3b), there are still some pending activities in a, waiting to be launched; therefore, φ is back to the top of the stack, so that (3a) can be used. In (3c), there are no more pending activities in a. Hence, a runs its $onActivityResult$ method and comes into the foreground. Both in (3b) and (3c), the finishing activity a' goes into the background; there, it can reach its $onDestroy$ state with (2g) and then it is removed from the stack with (2d).

EXAMPLE 5 (Example 4 continued). *At step* $\Omega_3 \diamond \mu_3$*, the user has tapped the* `Launch activity` *button and the callback method* `launchActivity` *has run. This method invokes a new instance of class* `Callee`*, which now has to be created and launched. First, as the current instance of* `Caller` *is about to lose focus, its method* `onPause` *is run. Using* (2c) *and then* (2a)–(2b)*, one gets*

$$\Omega_3 \diamond \mu_3 \Rightarrow^* \underbrace{\langle \ell \parallel \mathit{onPause} \parallel \mathit{Callee} \parallel \overline{\alpha}_4 \rangle}_{\Omega_4} \diamond \mu_4$$

which models the execution of `onPause`*. Then, an instance of* `Callee` *is created. Using* (3a)*, one gets*

$$\Omega_4 \diamond \mu_4 \Rightarrow \varphi'_5 :: \varphi_5 \diamond \mu_4[\ell' \mapsto o]$$

where

$$\begin{array}{rcl} \varphi'_5 & = & \langle \ell' \parallel \mathit{constructor} \parallel \varepsilon \parallel \alpha_{\ell'.\mathit{constructor}} \rangle, \\ \varphi_5 & = & \langle \ell \parallel \mathit{onPause} \parallel \varepsilon \parallel \overline{\alpha}_4 \rangle, \end{array}$$

with ℓ' *a fresh location,* o *a new object of class* `Callee` *and* $o(\mathit{parent}) = \ell$*.*

5.5 Saving and Restoring the State

The relation $Lifecycle$ that we consider in Definition 14 is intended to model the lifecyle of an activity as presented in Fig. 1. Note that other callback methods than those shown in Fig. 1 may be called by the system before or after the running state of an activity. For instance, `onSaveInstanceState` is called by the system before placing in a background state an activity that may be killed for memory reasons; it is called between states $running$ and $onStop$ and is used to save away any dynamic data in an activity, to be later retrieved if the activity needs to be re-created. The callback method `onRestoreInstanceState` is called after `onStart` when the activity is being re-initialized from a previously saved state. A possibility for modelling the calls to `onRestoreInstanceState` is to define the activity state $onRestoreInstanceState$, add the pairs of states

$$(onStart, onRestoreInstanceState)$$
$$(onRestoreInstanceState, onResume)$$

to $Lifecycle$ and use (2c). On the other hand, the calls to method `onSaveInstanceState` can be handled by defining the activity state $onSaveInstanceState$ and by adding the pairs

$$(running, onSaveInstanceState)$$
$$(onSaveInstanceState, onPause)$$
$$(onPause, onSaveInstanceState)$$
$$(onSaveInstanceState, onStop)$$

to $Lifecycle$. Then, in order to avoid infinite loops of the form

$$\begin{array}{rl} onSaveInstanceState & \rightarrow onPause \\ & \rightarrow onSaveInstanceState \\ & \rightarrow \ldots \end{array}$$

a field *paused* with boolean type has to be added to activities, indicating whether state $onPause$ has been met already. Moreover, rule (2c) has to be modified so that any move between states $onPause$ and $onSaveInstanceState$ is only allowed if *paused* holds *false*; such a move also sets field *paused* to *true*.

5.6 Validation of the Semantics

We are in the process of developing a symbolic executor for Dalvik that implements the operational semantics presented in this paper. Although yet incomplete, our tool is able to run small Android applications that fall within the scope of our framework (Section 3.2). We have tested the current version on a large set of small programs that we have written ourselves and verified successfully that it produces the same output as the emulator provided by the official Android SDK.

6. Conclusion

We have defined an operational semantics for a fragment of Android that includes its Dalvik bytecode and the lifecycle and intercommunication mechanism of the activities. This provides the basis for further developments, towards the inclusion of more Android components. In particular, similar lifecycles exist for other components as well, such as services and broadcast receivers. However, the lifecycle of activities is the most complex, since it involves an intercomponents communication mechanism and scheduling, that is what we have formalised. Our semantics is meant to be the basis for static analyses that take the lifecycle of the activities into account to provide more precise results than other analyses that abstract away that aspect of Android.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles Techniques and Tools.* Addison Wesley Publishing Company, 1986.

[2] androidDev. Android developers. `http://developer.android.com/`.

$$\frac{s \in \{onPause, onStop\} \quad \ell' \text{ is a fresh location and } o \text{ is a new object of class } A \text{ with } o(parent) = \ell}{\langle \ell \parallel s \parallel A :: \pi \parallel \overline{\alpha} \rangle :: \Omega \diamond \mu \Rightarrow \underline{\langle \ell' \parallel constructor \parallel \varepsilon \parallel \alpha_{\ell'.constructor} \rangle} :: \langle \ell \parallel s \parallel \pi \parallel \overline{\alpha} \rangle :: \Omega \diamond \mu[\ell' \mapsto o]} \tag{3a}$$

$$\frac{\begin{array}{c}\varphi' = \langle \ell' \parallel onPause \parallel \varepsilon \parallel \overline{\alpha}' \rangle \quad \mu(\ell')(finished) = true \\ \varphi = \langle \ell \parallel s \parallel A :: \pi \parallel \overline{\alpha} \rangle \quad s \in \{onPause, onStop\} \quad \mu(\ell')(parent) = \ell\end{array}}{\varphi' :: \varphi :: \Omega \diamond \mu \Rightarrow \varphi :: \varphi' :: \Omega \diamond \mu} \tag{3b}$$

$$\frac{\begin{array}{c}\varphi' = \langle \ell' \parallel onPause \parallel \varepsilon \parallel \overline{\alpha}' \rangle \quad \mu(\ell')(finished) = true \\ \varphi = \langle \ell \parallel s \parallel \varepsilon \parallel \overline{\alpha} \rangle \quad s \in \{onPause, onStop\} \quad \mu(\ell')(parent) = \ell\end{array}}{\varphi' :: \varphi :: \Omega \diamond \mu \Rightarrow \underline{\langle \ell \parallel s \parallel \varepsilon \parallel \alpha_{\ell.onActivityResult} \rangle} :: \varphi' :: \Omega \diamond \mu} \tag{3c}$$

Figure 12. Starting and returning from an activity. It is supposed that the activity stack on the left of $\Rightarrow$ contains no underlined frame.

[3] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to Android. In E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, pages 73–84. ACM, 2010.

[4] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS'12)*. The Internet Society, 2012.

[5] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In A. K. Agrawala, M. D. Corner, and D. Wetherall, editors, *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys'11)*, pages 239–252. ACM, 2011.

[6] dalvikdoc. Dalvik docs mirror. `http://www.milk.com/kodebase/dalvik-docs-mirror/`.

[7] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. QUIRE: Lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Security Symposium*. USENIX Association, 2011.

[8] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In R. H. Arpaci-Dusseau and B. Chen, editors, *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, pages 393–407. USENIX Association, 2010.

[9] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *Proceedings of the 20th USENIX Security Symposium (SEC'11)*. USENIX Association, 2011.

[10] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, pages 627–638. ACM, 2011.

[11] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium*. USENIX Association, 2011.

[12] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated Security Certification of Android Applications. Available at `http://www.cs.umd.edu/~avik/papers/scandroidascaa.pdf`, 2009.

[13] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster. Specifying and verifying the correctness of dynamic software updates. In R. Joshi, P. Müller, and A. Podelski, editors, *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE'12)*, volume 7152 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2012.

[14] J. Jeon, K. K. Micinski, and J. S. Foster. SymDroid: Symbolic execution for Dalvik bytecode. Technical Report CS-TR-5022, Department of Computer Science, University of Maryland, College Park, July 2012.

[15] JuliaSRL. Julia Srl. `http://www.juliasoft.com`.

[16] J. Kim, Y. Yoon, K. Yi, and J. Shin. SCANDAL: Static analyzer for detecting privacy leaks in Android applications. Mobile Security Technologies (MoST'12), a part of the IEEE Computer Society Security and Privacy Workshops, 2012.

[17] Klocwork. Klocwork. `http://www.klocwork.com`.

[18] J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In A. Paepcke, editor, *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*, volume 26(11) of *ACM SIGPLAN Notices*, pages 146–161, Phoenix, Arizona, USA, November 1991. ACM.

[19] É. Payet and F. Spoto. Static analysis of Android programs. *Information & Software Technology*, 54(11):1192–1201, November 2012.

[20] Scandal. SCANDAL. `http://ropas.snu.ac.kr/scandal/`.

[21] F. Spoto, F. Mesnard, and E. Payet. A termination analyzer for Java bytecode based on path-length. *ACM Transactions on Programming Languages and Systems*, 32, Issue 3:70 pages, March 2010.

[22] F. Tip and J. Palsberg. Scalable Propagation-based Call Graph Construction Algorithms. In M. B. Rosson and D. Lea, editors, *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)*, pages 281–293, Minneapolis, Minnesota, USA, October 2000. ACM.

[23] E. R. Wognsen and H. S. Karlsen. Static analysis of Dalvik bytecode and reflection in Android. Master's thesis, Department of Computer Science, Aalborg University, Aalborg, Denmark, 2012.