Study, Formalisation, and Analysis of Dalvik Bytecode

Henrik Søndberg Karlsen, Erik Ramsgaard Wognsen, Mads Chr. Olesen, and René Rydhof Hansen

Department of Computer Science, Aalborg University
{hkarls07,ewogns08}@student.aau.dk
{mchro,rrh}@cs.aau.dk

Abstract. With the large, and rapidly increasing, number of smartphones based on the Android platform, combined with the open nature of the platform that allows "apps" to be downloaded and executed on the smartphone, misbehaving and malicious (malware) apps is set to become a serious problem. To counter this problem, automated tools for analysing and verifying apps are essential. Further, to ensure high-fidelity of such tools, we believe that it is essential to formally specify both semantics and analyses.
In this paper we present the first (to the best of our knowledge) formalisation of the Dalvik bytecode language and formally specified control flow analysis for the language. To determine which features to include in the formalisation and analysis, 1,700 Android apps from the Android Market were downloaded and examined.

1 Introduction

With over 100 million Android devices already activated, and in excess of 400,000 new activations every day, Android has become one of the most widespread and fastest growing computing platforms for smartphones and tablet computers. The combination of the wide distribution and the open nature of the Android platform, where *apps* can be downloaded and installed not only from the official Android Market but also from unknown, untrusted, and potentially malicious third parties makes it obvious that tools are needed to ensure, and possibly certify, that apps are well-behaved and do not access (and leak) information or functionality not explicitly allowed and intended by the user. The problem is further exacerbated by the often sensitive and private nature of information stored on a smartphone as well as the potential for apps to (ab-)use services that cost the user money, e.g., by secretly sending text messages to expensive premium numbers [11].

In order to develop tools for highly trustworthy analysis and, especially, for certification we believe it is necessary to have a formal underpinning of the target platform and to show that the analyses are sound with respect to the formalisation. In this paper we first present a study of 1,700 Android apps, carried out in order to determine what Dalvik instructions and language features are most often used in typical apps. Based on the results of this study, we develop a formal operational semantics for the Dalvik bytecode language [14]. We further abstract the operational semantics into a formal *control flow analysis* for the Dalvik bytecode language, intended both as the basis for further, more specialised analyses but also by itself for detecting potentially malicious actions, e.g., leaking prviate information or surreptitiously calling expensive phone numbers. To the best of our knowledge, this is the first such formalisation of the Dalvik bytecode language and a accompanying control flow analysis. Finally, since our study revealed that more than half the apps examined used reflection, we illustrate how the *reflection* API can be formalised and analysed.

While Android apps are generally developed in Java, compiled to Java bytecode, and only then converted to Dalvik bytecode, we focus here on Dalvik bytecode because it is the common executable format for all Android apps and, therefore, offers the best opportunity for performing analyses as close to the code actually executed as possible and we sidestep issues relating to decompiling and reverse engineering apps, cf. [4].

1.1 Related Work

In [3] the tool *ComDroid* is described as a tool that performs "flow sensitive, intraprocedural static analysis with limited interprocedural analysis" of Dalvik bytecode programs. It is designed to analyse the communication between Android applications through the so-called *Intents*, the Android equivalent of events, and to find potential security vulnerabilities in the communication patterns of applications. In [6] the ComDroid tool is used as a component of another analysis tool, called *Stowaway*, that analyses API calls in applications to determine if they are over-privileged. In order to improve the precision and efficacy of the analysis, Stowaway incorporates some analysis of the reflection features found in Dalvik bytecode (through the `java.lang.reflect` library). Both ComDroid and Stowaway are sophisticated analysis tools covering not only the Dalvik bytecode language but also important parts of the API and the Android platform itself. However, since the analyses are not actually specified in detail, neither formally nor informally, it is impossible to evaluate the exact strengths and weaknesses of the underlying analyses. Indeed, it is stated in [6] that Stowaway makes a "primarily linear traversal" and that it "experiences problems with non-linear control flow". This emphasises the need for a formalisation of both the Dalvik bytecode language as well as the control flow analysis.

In [4] Android applications are analysed by first recovering the Java source through decompilation and then using the Fortify SCA static analysis tool to detect potential security vulnerabilities. While the paper reports on impressive results using this approach, it is also noted that it was not possible to recover the source code for all the targeted applications and thus making analysis of those applications impossible. Analysing directly at the bytecode sidesteps this issue.

The approach described in [13] takes advantage of the fact that most, if not all, Android applications are developed in Java and adapts the Julia framework for Java bytecode analysis to the specificities of the Java bytecode that results from developing Android applications (before being converted to Dalvik bytecode). The Julia framework is theoretically sound and well-documented, but the described solution requires access to the Java bytecode version of an application in order to analyse it.

2 Study of Apps

In order to identify which Dalvik bytecode instructions and which Java language features are used in typical Android apps, we have collected and examined 1,700 of the most popular free apps from Android Market [7]. Notable features include code obfuscation, threading, reflection, native libraries, and dynamic class loading. The apps were collected in November 2011 using Android 2.3.3 on a Samsung Nexus S.

For efficiency reasons the Dalvik bytecode language contains several specialised variants of many common instructions, e.g., there are numerous variants of the `move` instruction. For our study we have grouped instruction variants that are semantically similar, e.g., most variants of the `move` instruction belong to the same group.

Table 1. Percentages of apps in our data set that use various features.

Feature	Used by apps	Hereof in libraries
Obfuscated source	64.82%	-
Has native libraries	20.35%	-
`java/lang/Thread`	90.18%	24.07%
`java/lang/reflect`	73.00%	55.92%
`java/lang/ClassLoader`	39.71%	81.19%
`java/lang/Runtime;->exec`	19.53%	80.44%

In the semantics (see Section 3) we use the same notion of grouping to abstract and generalise the original 218 Dalvik bytecode instructions into a set of 39 instructions.

In our study we found that, with the exception of the `filled-new-array` instruction, *all* types of Dalvik bytecode instructions are used in more than half the studied apps. In particular, the instructions `invoke-direct` and `return-void` are used in every app and even the most rare instructions, `sparse-switch` and `filled-new-array`, are used in 69.7% and 22.3% of the studied apps, respectively. The instructions that occur most frequently are `invoke-virtual` and `move-result`, which are used more than 12 million times each in total in the 1,700 apps. In comparison, `filled-new-array` is used 1,930 times. For full details, see [10].

The observations made from studying the use of Java features are summarised in Table 1 and are explained in detail below. For the study we have separated code into *developer code* and *library code*. Developer code is code that lies within the natural packages for the application. For an application `company.app` this means all classes located directly in the packages `/`, `/company/`, `/company/app/`, and any subpackages in `/company/app/`. Library code is everything else.

Code obfuscation, especially using ProGuard [5], is used to a large extent. We searched for classes named "`a`" within apps in the data set, and used this as an approximation to determine if an app contains any obfuscated code. The same approach was used in [4] which found 36% of apps to include obfuscated code. We found the class in 64.82% of the apps. Obfuscation has legitimate uses and is recommended by Google [8], but makes it harder to manually inspect the code.

Native libraries, i.e., ARM shared object (`.so`) files, were included in 20.35% of the apps we studied[1]. A previous study [4] found that of their 1,100 studied apps from September 2010, only 6.45% included shared objects. We presume the increased usage is because the Android NDK, released June 25, 2009, has gained more widespread use in 2011.

Threading, as indicated by the use of monitors, i.e., the Java `synchronized` keyword, was found in 88% of the apps. Furthermore, 90.18% of the apps include a reference to `java/lang/Thread`. These observations are not conclusive, but indicate that multi-threaded programming is wide-spread. However, further studies are needed to substantiate the results.

Reflection is used extensively in Android apps for accessing private and hidden classes, methods, and fields, for JSON and XML parsing, and for backward compatibility [6]. We confirmed these observations by manual inspections. Of the 940 apps studied in [6], 61% were found to use reflection, and using automated static analysis they were able to resolve the targets for 59% of the reflective calls. 73% of the apps in our data set use reflection. This indicates that a formalisation of reflection in Dalvik is necessary to precisely analyse most apps. We treat this in Section 5.

Class Loading Of the studied apps 39.71% contain a reference to the class loader library, `java/lang/ClassLoader`, which means that the app can load Dalvik

[1] In addition, 15 apps included the ARM executable `gdbserver`.

executable (DEX) files and JAR files at runtime. Manual inspection shows that some of these uses relate to IPC transport with the Android Parcelable interface.

The Java method `Runtime.exec()` is used to execute programs in a separate native process and is present in 19.53% of the apps. We manually inspected some of these uses. Most of these do not use a hardcoded string as the argument to `exec`, but of those that do, we found execution of both the `su` and `logcat` programs which, if successful, give the app access to run programs as the super user on the platform or read logs (with private data [4]) from all applications, respectively.

3 Operational Semantics

In this section we describe the formalisation of the Dalvik bytecode language using operational semantics. With the exception of instructions related to concurrency, we have formalised the full (generalised) instruction set and below we present the semantic rules for a few interesting instructions. The approach is inspired by a similar effort to formalising the Java Card bytecode language [15, 9].

To simplify our work, we have made a number of convenient, but minor, generalisations, including: simplified type hierarchy to avoid dealing with bit-level operations, except when absolutely necessary; "inlining" of the constant pools for easier and more direct reference of strings, types, methods, and fields; and finally an idealised program counter abstracting away the length of instructions. While none of these modifications change the expressive power of a Dalvik application, they greatly simplify presentation and formalisation.

The study described in Section 2 impacted the formalisation in two major ways: it was clear that all of the core bytecode language had to be formalised and also that the reflection API had to be formalised. In order to ensure that the formalisation correctly represents the Dalvik (informal) semantics, we based the formalisation on the documentation for Dalvik [1], inspection of the source code for the Dalvik VM in Android [2], tests of handwritten bytecode, and experiments with disassembly of compiled Java code.

3.1 Program Structure and Semantic Domains

To facilitate the development of the formal semantics for Dalvik bytecode, it is important to have a good formalisation of the program structure of apps. Here we follow [15] and use domains equipped with accessor-functions, written in an OO inspired notation. The Method domain is presented to illustrate this approach:

$$\begin{aligned}\mathsf{Method} = {} & (\mathit{name}\colon \mathsf{MethodName}) \times (\mathit{class}\colon \mathsf{Class}) \times \\ & (\mathit{argType}\colon \mathsf{Type}^{*}) \times (\mathit{returnType}\colon \mathsf{Type} \cup \{\mathtt{void}\}) \times \\ & (\mathit{instructionAt}\colon \mathsf{PC} \to \mathsf{Instruction}) \times \\ & (\mathit{kind}\colon \{\mathtt{virtual}, \mathtt{static}, \mathtt{direct}\}) \times \\ & (\mathit{maxLocal}\colon \mathbb{N}_0) \times (\mathit{handlers}\colon \mathbb{N}_0 \to \mathsf{ExcHandler})\end{aligned}$$

The return type of a method $m \in \mathsf{Method}$ is denoted $m.\mathit{returnType}$. The function *argType* is a sequence of the types of the arguments to the method. The function *instructionAt* maps to another function mapping locations in the method (program counter values) to instructions. *maxLocal* is the number of the last register used for local variables in the method (Dalvik uses registers instead of an operand stack).

The next step in our formalisation is to define the semantic domains. Since these are quite standard, and for lack of space, we only give a few examples. Local registers are modelled simply as a map from register names to values using $\perp$ to denote undefined register contents: $\mathsf{LocalReg} = (\mathbb{N}_0 \cup \{\mathtt{retval}\}) \to \mathsf{Val}_{\perp}$. Note that a special register, the `retval` register, is used to transfer return values from invoked methods.

Addresses are used to identify specific program points and are composed of a method and a program counter value: $\mathsf{Addr} = \mathsf{Method} \times \mathsf{PC}$ where $\mathsf{PC} = \mathbb{N}_0$. This gives a unique address for every instruction in a Dalvik program. We can then define stack frames to contain a method and a program counter, i.e., an address, and the local registers: $\mathsf{Frame} = \mathsf{Method} \times \mathsf{PC} \times \mathsf{LocalReg}$. This leads to the following definition of call stacks as simply a sequence of frames except that the top frame may be an *exception frame* representing an as yet unhandled exception: $\mathsf{CallStack} = (\mathsf{Frame} + \mathsf{ExcFrame}) \times \mathsf{Frame}^*$. An exception frame contains the location of its corresponding exception object on the heap and the address of the instruction that threw the exception: $\mathsf{ExcFrame} = \mathsf{Location} \times \mathsf{Method} \times \mathsf{PC}$. We shall not go into further details with the semantic domains, merely refer to [10] for the full definition.

3.2 Semantic Rules

We specify the semantics as a straightforward structural operational semantics where each configuration comprises a static heap, a heap, and a call stack (as defined above). To illustrate the semantics, we present the semantic rule for the central `invoke-virtual` instruction:

$$\frac{\begin{array}{c} m.\mathit{instructionAt}(pc) = \texttt{invoke-virtual}\ v_1 \ldots v_n\ \ \mathit{meth} \\ R(v_1) = \mathit{loc} \qquad \mathit{loc} \neq \texttt{null} \qquad o = H(\mathit{loc}) \\ n = \mathit{arity}(\mathit{meth}) \qquad m' = \mathit{resolveMethod}(\mathit{meth}, o.\mathit{class}) \\ R' = [0 \mapsto \bot, \ldots, m'.\mathit{maxLocal} \mapsto \bot, \\ \qquad m'.\mathit{maxLocal} + 1 \mapsto v_1, \ldots, m'.\mathit{maxLocal} + \mathit{arity}(m') \mapsto v_n] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', 0, R'\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

The function *instructionAt* is an access function on the Method domain that identifies the instruction at a given location in a specified method. In the new configuration, the static heap (S) and dynamic heap (H) are unchanged. The instruction receives n arguments and the signature of the method to invoke. The first argument v_1 is a reference to the object on which the method should be invoked. The location of the method is resolved using the auxiliary function *resolveMethod* as explained below and this method is put into a new frame on top of the call stack, with the program counter set to 0. A new set of local registers, R', is created, where the registers up to $m'.maxLocal$ are mapped to $\bot_{\mathsf{Val}}$ such that they are initially undefined. The arguments are then mapped into the next registers. Like Java, Dalvik implements dynamic dispatch. We define a function to search through the class hierarchy for virtual methods (all non-static methods that are overridable or defined in interfaces):

$$\mathit{resolveMethod}(\mathit{meth}, \mathit{cl}) = \begin{cases} \bot & \text{if } \mathit{cl} = \bot \\ m & \text{if } m \in \mathit{cl}.\mathit{methods} \wedge \mathit{meth} \lhd m \wedge \\ & \quad m.\mathit{kind} = \texttt{virtual} \\ \mathit{resolveMethod}(\mathit{meth}, \mathit{cl}.\mathit{super}) & \text{otherwise} \end{cases}$$

where $\mathit{meth} \lhd m$ is a predicate formalising when a method signature *meth* is compatible with a given method $m \in \mathsf{Method}$, i.e. when the names, argument types and return types match.

Exceptions can be thrown either explicitly using the `throw` instruction, or by the system in case of a runtime error, such as a null dereference. Exception handlers have a type for the exceptions it may catch, a program counter value pointing to the handler code, and program counter values defining the boundaries of the region covered by the exception handler. Exceptions that are not handled in the method where it is thrown are put on the call stack for the next method's exception handlers to try to handle. The `throw` instruction is defined as follows:

$$\frac{\begin{array}{c} m.\mathit{instructionAt}(pc) = \mathtt{throw}\ \ v \\ R(v) = loc_E \neq \mathtt{null} \qquad cl = H(loc_E).class \qquad cl \preceq \mathtt{Throwable} \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle loc_E, m, pc\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

When the top frame is an exception frame and a handler is found, the following rule applies (an analogous rule applies when no handler is found):

$$\frac{cl = H(loc_E).class \qquad \mathit{findHandler}(m, pc, cl) = pc'}{\begin{array}{l} A \vdash \langle S, H, \langle loc_E, m_E, pc_E\rangle :: \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \\ \qquad\qquad\qquad\qquad \langle S, H, \langle m, pc', R[\mathtt{retval} \mapsto loc_E]\rangle :: SF\rangle \end{array}}$$

The auxiliary function *findHandler* finds the exception handler matching the location in the method and the given exception class or $\perp$ if no appropriate handler is available.

4 Control Flow Analysis

In the following we give a very brief overview of the control flow analysis, which is specified using flow logic [12]. In this approach an analysis is defined through a number of flow logic judgements that specify what is required of an analysis result in order to be correct. Details can be found in [12]. Many other frameworks for program analysis exist, but the combination of a structural operational semantics and flow logic has proven to be flexible and easy to use for both theoretical developments as well as for implementation. A more detailed comparison with other approaches is out of scope for this paper.

The abstract domains, used in the analysis to statically represent runtime values, are based very closely on the underlying semantic domains. Similar to the *class object graphs* in [16], we map all instances of a given class to one abstract class instance. In order to achieve sufficient precision, the analysis is flow-sensitive, but only intra-procedurally. This yields an abstract domain for local registers that tracks (abstract) values for every address in the program (including a special address, denoted END, used to track return values from methods): $\widehat{\mathsf{LocalReg}} = \widehat{\mathsf{Addr}} \to (\mathsf{Register} \cup \{\mathsf{END}\}) \to \widehat{\mathsf{Val}}$ where $\widehat{\mathsf{Val}}$ is the domain for abstract values, containing abstractions of primitive types, references and classes. Similarly we can define the abstract domains for static and "ordinary" heaps: $\widehat{\mathsf{StaticHeap}} = \mathsf{Field} \to \widehat{\mathsf{Val}}$ and $\widehat{\mathsf{Heap}} = \widehat{\mathsf{Ref}} \to (\widehat{\mathsf{Object}} + \widehat{\mathsf{Array}})$ respectively. The overall domain for the analysis is then defined as $\widehat{\mathsf{Analysis}} = \widehat{\mathsf{StaticHeap}} \times \widehat{\mathsf{Heap}} \times \widehat{\mathsf{LocalReg}}$.

We now illustrate flow logic judgements, starting with the judgement for the `move` instruction: after a `move` instruction, the destination register contains the value in the source register while all others are unchanged as signified by the $\sqsubseteq_{\{v\}}$ relation:

$$\begin{array}{l} (\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\colon \mathtt{move}\ \ v_1\ \ v_2 \\ \qquad \text{iff} \quad \hat{R}(m, pc)(v_2) \sqsubseteq \hat{R}(m, pc+1)(v_1) \\ \qquad\qquad\ \ \hat{R}(m, pc) \sqsubseteq_{\{v_1\}} \hat{R}(m, pc+1) \end{array}$$

The conditions are joined by an implicit conjunction. A solution for the analysis that satisfies the ordering specified by the flow logic judgements will be a safe over-approximation of all possible values in every method. To satisfy the conditions for the `move` instruction, the value of the destination register at the following instruction must be the least upper bound of the old and the new value.

The flow logic judgement for the `invoke-virtual` instruction works as follows: For each possible object the method can be called on, the method is resolved (by dynamic dispatch using the *resolveMethod* function from the semantics), the arguments are transferred, and the `retval` register is updated with the return value unless the return type of the method is void:

$$\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\colon \texttt{invoke-virtual}\ v_1 \ldots v_n\ \ meth \\
\quad \text{iff}\quad \forall (\textsf{ObjRef}\ cl) \in \hat{R}(m,pc)(v_1)\colon \\
\qquad\qquad m' = \mathit{resolveMethod}(meth, cl) \\
\qquad\qquad \forall 1 \le i \le n\colon \hat{R}(m,pc)(v_i) \sqsubseteq \hat{R}(m',0)(m'.\mathit{maxLocal} + i) \\
\qquad\qquad m'.\mathit{returnType} \ne \texttt{void} \Rightarrow \hat{R}(m', \textsf{END}) \sqsubseteq \hat{R}(m, pc+1)(\texttt{retval}) \\
\qquad\quad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc+1)
\end{array}$$

Object references are modelled simply as classes, which is possible since these are all known statically.

Two things can happen when an exception is thrown: If a local handler exists, control is transferred to that handler with a reference to the exception object in the `retval` register. If no local handler exists, the method aborts and the exception is put on the call stack in an exception frame. The analysis will treat this situation with the following auxiliary predicate:

$$\begin{array}{l}
\textsf{HANDLE}_{(\hat{R},\hat{E})}((\textsf{ExcRef}\ cl_E), (m, pc\,)) \equiv \\
\quad \mathit{findHandler}(m, pc, cl_E) = pc' \Rightarrow \{(\textsf{ExcRef}\ cl_E)\} \sqsubseteq \hat{R}(m, pc')(\texttt{retval}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc') \\
\quad \mathit{findHandler}(m, pc, cl_E) = \bot \Rightarrow \{(\textsf{ExcRef}\ cl_E)\} \sqsubseteq \hat{E}(m)
\end{array}$$

where the exception cache $\hat{E}$ abstracts the details of exceptions on the call stack by storing them when no local handler is found. With the above predicate it is now trivial to define the analysis for the `throw` instruction:

$$\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}, \hat{E}) \models (m, pc)\colon \texttt{throw}\ \ v \\
\quad \text{iff}\quad \forall (\textsf{ExcRef}\ cl_E) \in \hat{R}(m,pc)(v)\colon \\
\qquad\qquad \textsf{HANDLE}_{(\hat{R},\hat{E})}((\textsf{ExcRef}\ cl_E), (m, pc))
\end{array}$$

5 Reflection

As shown by our study, many apps use reflection and it is therefore important to handle this in the formalisation and analysis. Below we show how a central API method in the reflection library is formalised and analysed, namely method invocation. Refer to [10] for further details.

Dynamic method invocation is done by calling the `invoke()` method on a `java.lang.reflect.Method` object. This method is used by 84.3% of the apps that use reflection. Modelling the semantics of this results in the following special case for the `invoke-virtual` instruction:

$$\frac{\begin{array}{c}
m.\mathit{instructionAt}(pc) = \texttt{invoke-virtual}\ \ v_1\ \ v_2\ \ v_3\ \ meth \\
meth.\mathit{name} = \texttt{Method.invoke} \qquad R(v_1) = loc_1 \qquad loc_1 \ne \texttt{null} \\
o_1 = H(loc_1) \qquad o_1.\mathit{class} \preceq \texttt{Method} \qquad meth' = \mathit{methodSignature}(o_1) \\
R(v_2) = loc_2 \qquad loc_2 \ne \texttt{null} \qquad o_2 = H(loc_2) \qquad R(v_3) = loc_3 \qquad a = H(loc_3) \in \textsf{Array} \\
m' = \mathit{reflectResolveMethod}(meth', o_2.\mathit{class}) \qquad R' = [0 \mapsto \bot, \ldots, m'.\mathit{maxLocal} \mapsto \bot, \\
m'.\mathit{maxLocal} + 1 \mapsto a.\mathit{value}(0), \ldots, m'.\mathit{maxLocal} + a.\mathit{length} \mapsto a.\mathit{value}(a.\mathit{length} - 1)]
\end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', 0, R'\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

Register v_1 points to the `Method` object, v_2 points to the receiver object, i.e., the object on which the method is invoked, and v_3 points to an array with the arguments for the method. The rule uses two auxiliary functions: *methodSignature* extracts the semantic method signature from a `Method` object and *reflectResolveMethod* works like *resolveMethod* except that it ignores the method kind such that it works on any method.

The flow logic judgement is similar to the one for `invoke-virtual`, with another layer of indirection because the over-approximative analysis represents sets of possible values in registers:

```
$(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)$: invoke-virtual $v_1\ v_2\ v_3\ meth$
    iff $meth.name = \texttt{Method.invoke}$
        $\forall(\mathsf{ObjRef}\ cl) \in \{cl \mid cl \in \hat{R}(m, pc)(v_1) \wedge cl \preceq \texttt{Method}\}$:
          $\forall meth' \in methodSignature(\hat{H}(cl))$:
            $\forall(\mathsf{ObjRef}\ cl') \in \hat{R}(m, pc)(v_2)$:
              $m' = reflectResolveMethod(meth', cl')$
              $\forall 1 \leq i \leq arity(meth')$:
                $\forall(\mathsf{ArrRef}\ a) \in \hat{R}(m, pc)(v_3)$:
                  $\hat{H}(\mathsf{ArrRef}\ a) \sqsubseteq \hat{R}(m', 0)(m'.maxLocal + i)$
              $m'.returnType \neq \texttt{void} \Rightarrow \hat{R}(m', \mathsf{END}) \sqsubseteq \hat{R}(m, pc + 1)(\texttt{retval})$
        $\hat{R}(m, pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc + 1)$
```

In effect, the conditions of the basic `invoke-virtual` must be satisfied for each method that can be resolved from each `Method` object reference in v_1. Furthermore, the contents of each array referenced by v_3 must be available in each argument register for the current method being invoked.

This approach presents a problem: For the *methodSignature* function to work, the `Method` object must be known statically and no classes must be loaded or created dynamically. In the studied apps, the `Method` object is typically derived from a `Class` object using the `getMethod()` method which takes the method name as a string argument. The `Class` object itself is also typically obtained from a string using the `Class.forName()` method. In many cases we have determined that the string can be found statically. Of the apps that use `Method.invoke()`, 18.9% use only local string constants for `forName()` and `getMethod()` or get the `Class` and `Method` objects from the `const-class` Dalvik instruction or simple API methods such as the `getClass()` instance method.

We discovered that many of the apps that we could not classify as using static strings only used strings of unknown origin in a single case: A Google AdMob library for install referrer tracking. This library calls a number of `BroadcastReceivers` and as a safe over-approximation, we expect to include this as a special case in the analysis by letting it call all available broadcast receivers. With this addition, we would be able to analyse the use of reflection in 36.7% of the apps.

The above numbers are based on primitive intra-procedural data tracking and with the implementation of the analysis and its inter-procedural (but flow insensitive) data flow, they should be improved.

6 Conclusion

In this paper we have shown excerpts of a formal semantics for the Dalvik bytecode language and a formally specified control flow analysis, both based on the results of a study of 1,700 Android apps. Also based on this study, we have identified reflection as a particularly important language feature (supported through the `java.lang.reflect` API) to take into account when formalising semantics and analysis.

In future work we will finish the formal proof of soundness for the analysis (work in progress, currently a few cases have been proved) and also implement a prototype of the control flow analysis, by systematically converting the flow logic judgements into Prolog terms.

References

1. Android Open Source Project: Bytecode for the Dalvik VM. http://source.android.com/tech/dalvik/dalvik-bytecode.html (December 13th 2011)
2. Android Open Source Project: Downloading the Source Tree. http://source.android.com/source/downloading.html (December 14th 2011)
3. Chin, E., Felt, A., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: Proc. of the Annual International Conference on Mobile Systems, Applications, and Services (2011)
4. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of Android application security. In: In proc. of the 20th USENIX Security Symposium (SEC'11). pp. 315–330. USENIX Association, San Francisco, CA, USA (Aug 2011)
5. Eric Lafortune: ProGuard. http://proguard.sourceforge.net (December 13th 2011)
6. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 627–638. CCS '11, ACM, New York, NY, USA (2011)
7. Google: Android Market. https://market.android.com (November 29th 2011)
8. Google Inc.: ProGuard | Android Developers. Web page available at http://developer.android.com/guide/developing/tools/proguard.html (December 13th 2011)
9. Hansen, R.R.: Flow Logic for Language-Based Safety and Security. Ph.D. thesis, Technical University of Denmark (2005)
10. Karlsen, H.S., Wognsen, E.R.: Study, Formalisation, and Analysis of Dalvik Bytecode. Master's thesis, Aalborg University (2012), forthcoming. Preliminary version available at http://students.cs.aau.dk/erikrw/report_sw9.pdf
11. Alienvault Labs: Analysis of Trojan-SMS.AndroidOS.FakePlayer.a. Web page available at http://labs.alienvault.com/labs/index.php/2010/analysis-of-trojan-sms-androidos-fakeplayer-a/ (November 29th 2011)
12. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Verlag (1999)
13. Payet, É., Spoto, F.: Static analysis of Android programs. In: Proc. of the 23rd International Conference on Automated Deduction (CADE-23). Lecture Notes in Computer Science, vol. 6803, pp. 439–445. Springer Verlag, Wroclaw, Poland (Jul/Aug 2011)
14. The Android Open Source Project: Bytecode for the Dalvik VM. Retrieved 2011-10-13 from http://source.android.com/tech/dalvik/dalvik-bytecode.html
15. Siveroni, I.: Operational Semantics of the Java Card Virtual Machine. Journal of Logic and Algebraic Programming 58(1–2), 3–25 (Jan/Mar 2004)
16. Vitek, J., Horspool, R.N., Uhl, J.S.: Compile-Time Analysis of Object-Oriented Programs. In: Proc. International Conference on Compiler Construction (CC'92). Lecture Notes in Computer Science, vol. 641. Springer Verlag (1992)