Formalisation and Analysis of Dalvik Bytecode[☆]

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

Department of Computer Science, Aalborg University, Denmark

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 are set to become a serious problem. To counter this problem, automated tools for analysing and verifying apps are essential. Furthermore, to ensure high-fidelity of such tools, it is essential to formally specify both semantics and analyses.

In this paper we present, to the best of our knowledge, the first formalisation of the complete Dalvik bytecode language including reflection features and the first formally specified control flow analysis for the language, including advanced control flow features such as dynamic dispatch, exceptions, and reflection. To determine which features to include in the formalisation and analysis, 1,700 Android apps from the Google Play app market (formerly known as Android Market) were downloaded and examined.

Keywords: Dalvik, bytecode, Android, static analysis, flow logic, reflection

1. Introduction

Since Android was first introduced, more than 100 million Android devices have been activated, and more than 400,000 new devices are activated every day. This makes Android 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 Google Play app market[1] 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 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

[☆]This work is based on the paper "Study, Formalisation, and Analysis of Dalvik Bytecode" presented at the *Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation* (BYTECODE 2012) [14] and the master's thesis of the first two authors [13].

Email addresses: `erw@cs.aau.dk` (Erik Ramsgaard Wognsen), `henrik.karlsen@gmail.com` (Henrik Søndberg Karlsen), `mchro@cs.aau.dk` (Mads Chr. Olesen), `rrh@cs.aau.dk` (René Rydhof Hansen)

[1]See `https://play.google.com`. Last accessed 11 December 2012.

the potential for apps to (ab-)use services that cost the user money, e.g., by secretly sending text messages to expensive premium numbers [3].

In order to develop trustworthy tools for analysis and especially for certification, we believe it is necessary to have a formal underpinning of the target platform allowing sound analyses to be developed with confidence. 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 [1]. Dalvik bytecode is similar to Java bytecode. The two most notable differences are: Dalvik is register based rather than stack based and the local registers are untyped. These differences are reflected in their two instruction sets. In addition, there are a number of differences in their respective runtime systems, which we will not discuss any further here. Based on the operational semantics, we develop and formally specify a *control flow analysis* for the Dalvik bytecode language. The analysis is intended both as the basis for further and more specialised analyses, but also by itself for detecting potentially malicious actions, e.g., leaking private 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 an accompanying control flow analysis. Since our study revealed that more than half of the apps examined use *reflection*, we also formalise the semantics of the reflection API and also extend the control flow analysis to cover many uses of reflection as determined by the study.

Finally, we have developed a prototype implementation of the control flow analysis in order to investigate properties and behaviour of the analysis (Section 6). The prototype is strictly meant as a proof-of-concept and not a finished production tool. Consequently the implementation has some notable limitations regarding the analysis of real apps: there is no support for libraries and APIs, limited support for reflection, and the exception analysis, although formalised, is not implemented either. Even with these limitations the prototype has been used to perform a number of useful analyses described in sections 7 and 8.

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 allows us to sidestep issues relating to decompiling and reverse engineering apps, cf. [7].

1.1. Related Work

In [4] 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. The same ComDroid tool is used as a component in another analysis tool called *Stowaway*, that analyses API calls in applications to determine if they are over-privileged [8]. 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 their exact strengths and weaknesses. Indeed, it is stated in [8] 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.

Another approach to analysing Android applications for potential security violations is discussed in [7]. Here 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 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 level sidesteps this issue.

The approach described in [24] takes advantage of the fact that most, if not all, Android applications are developed in Java and adapts the Julia framework (see [28]) for Java bytecode analysis to the specificities of the Java bytecode that results from developing Android applications (before being converted to Dalvik bytecode). As an example, the Julia-based analysis can handle the specific use of reflection for XML[2] specified graphical views prevalent on the Android platform. It does not handle the general case of reflection nor the special instances handled by our approach. The Julia framework is theoretically sound, comprehensive, and well-documented, but the described solution requires access to the Java bytecode version of an application in order to analyse it.

In [12] the authors formalise a simple language, μ-Dalvik, and translate Dalvik bytecode into μ-Dalvik, with the goal of performing symbolic execution of Dalvik programs. μ-Dalvik corresponds roughly to the core of the language that we have formalised, but with some important differences: Dalvik bytecode might be mapped to several μ-Dalvik instructions, whereas our mapping is always one to one. In addition we have formalised exceptions, the `array-length` instruction, and some common cases of reflection, neither of which are formalised in μ-Dalvik. As our study shows, reflection is used by a majority of apps making it a crucial feature to model.

Much of the work on Dalvik bytecode has been inspired by similar work on Java bytecode, including the current paper which borrows the main methodology from [11]. One of the first, possibly *the* first, formal semantics for Java bytecode language was defined by Bertelsen in [2]. Other early work on formalisation and analysis of Java bytecode includes [10, 9] where a core bytecode language is given a formal semantics and a type system for analysis is proven correct. Spoto provided one of the first formal semantics used as the basis for abstract interpretation of Java bytecode [29]. This work inspired the Julia framework for analysis [28] which is based on traditional abstract interpretation parameterised with different abstract domains. As mentioned above the Julia framework has recently been adapted to cover Dalvik bytecode [24] translated from Java bytecode. One of the earliest and most comprehensive tools for Java

[2]Extensible Markup Language, a common standard language for document encoding, data interchange, etc.

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%

bytecode analysis is the Soot framework [30]. The work on Soot has focused more on effectiveness and efficiency than on formalisation and thus lacks formal specification and proof.

2. Study of Apps

To identify which Dalvik bytecode instructions and which Java language features are used in typical Android apps, we collected and examined the 50 most popular free apps of each category of Android Market (now part of Google Play), 1,700 apps in total. 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 smartphone.

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. 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. The mapping between the original and generalised instructions is given in Appendix A.

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 of the studied apps. In particular, the instructions `invoke-direct` and `return-void` are used in every app and even the two 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 Appendix B.

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[3], 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 [7] which found 36% of apps to include obfuscated code. We found the class in 64.82% of the apps. Obfuscation is recommended by Google[4], 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[5]. A previous study [7] 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[6], 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.

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 them 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, gives the app access to run programs as the super user on the platform or read logs (with private data [7]) from all applications, respectively. Some apps also use the `pm install` command to install other apps at runtime.

Class loading Of the studied apps 39.71% contain a reference to the Java class loader library, `java/lang/ClassLoader`, or a subclass such as the `dalvik/system/DexClassLoader`. However, only 13.1% of apps use the `loadClass()` or `defineClass()` methods to actually load or define classes at runtime. Class loading allows the loading of Dalvik executable (DEX) files and JAR files while class definition allows for programmatic definition of Java classes, e.g., from scripting languages such as Javascript. If the classes being loaded are not present, e.g., if they are downloaded from the Internet, the app cannot be analysed statically before installation. Furthermore, if the classes being loaded are created dynamically from other languages, analysing the use before installation would require the analysis tool to parse/analyse these languages. A simpler solution for handling the apps that use these features would be to analyse the class just before it is being loaded, on the device. However, we consider this as out of scope for this paper.

[3]See `http://proguard.sourceforge.net`. Last accessed 13 December 2011.

[4]See `http://developer.android.com/guide/developing/tools/proguard.html`. Last accessed 13 December 2011.

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

[6]Previously "Native Development Kit", a toolset that facilitates interfacing Java/Dalvik with C/C++ and native (ARM) code.

Class Transformation allows developers to change behaviour of classes at runtime, before they are loaded by the VM [22]. It is a Java feature, and is therefore also available in Android. The transformations allowed include adding new instructions and changing control flow. We found no apps in our data set that use this feature, and will therefore not return to this subject.

Reflection has been reported to be used extensively in Android apps for accessing private and hidden classes, methods, and fields, for JSON[7] and XML parsing, and for backward compatibility [8]. We confirmed these observations by manual inspection. Of the 940 apps studied in [8], 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[8]. This indicates that a formalisation of reflection in Dalvik is necessary to precisely analyse most apps. Reflection resolves classes, methods, and fields from strings. When these are statically known, static analysis becomes possible. We treat this in Section 5.

Javascript Interfaces allow Javascript in a webpage embedded in an app to control that app. Android supports in-app loading of webpages, through the WebKit API[9] that provides a custom embedded web browser. This API includes the `addJavascriptInterface()` method whose purpose is to make the methods on a Java object available to Javascript code. The method is used in 39% of the apps in our data set. The interface allows webpages loaded by the app to call methods on the Java object. Previous studies have shown that advertisement and analytics libraries use this to give the third-party advertisement companies access to sensor information, such as location updates [18]. We confirmed this use through manual inspection, and furthermore discovered apps that were practically webpages, and where the Dalvik code merely loads the page and extends the browser functionality, e.g. by allowing the webpage to send text messages from the phone.

3. Operational Semantics

In this section we describe the formalisation of the Dalvik bytecode language using structural operational semantics [25]. With the exception of instructions related to concurrency, we have formalised the full generalised instruction set of 39 instructions, reduced from the full Dalvik instruction set comprising 218 instructions, see Appendix A. Below we present the semantic rules for a few representative instructions and refer to [13] for an exhaustive list. The approach is inspired by a similar effort to formalising the Java Card bytecode language [27, 11].

[7] JavaScript Object Notation, a standard data interchange format similar to XML.

[8] This number counts direct uses of reflection in the app, not indirect uses through Android APIs that themselves employ reflection such as `Activity.setContentView()`.

[9] See `http://developer.android.com/reference/android/webkit/package-summary.html`. Last accessed 8 May 2012.

To simplify our work, we have made three convenient, but minor, generalisations: simplification of the 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 idealising the program counter by 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[10], tests of handwritten bytecode, and experiments with disassembly of compiled Java code.

3.1. Program Structure

To facilitate the development of the formal semantics for Dalvik bytecode, it is important to have a good formalisation of the formal structure of an app. We follow the general approach of [27, 11] and use domains equipped with accessor-functions: the domain $\mathsf{D} = \mathsf{D}_1 \times \ldots \times \mathsf{D}_n$ with functions $f_i : \mathsf{D} \to \mathsf{D}_i$ is expressed in a record-like format: $\mathsf{D} = (f_1 \colon \mathsf{D}_1) \times \ldots \times (f_n \colon \mathsf{D}_n)$. The access functions are used in an object-oriented style where, for $d \in \mathsf{D}$, $f_i(d)$ is written $d.f_i$ and $f_i(d, a_1, \ldots, a_m)$ is written $d.f_i(a_1, \ldots, a_m)$. The notation $d[f \mapsto x]$ expresses the domain d where the value of access function f is updated to x. Similarly the notation $g[x \mapsto y]$ expresses the function g where value x now maps to y. In the following we use the domains for classes, interfaces, fields, and methods to illustrate our approach. The remaining domains are found in Appendix C.

A class is specified with a class name, the app in which the class is defined, the Java package it belongs to, a superclass, as well as sets of implemented interfaces, fields, access flags, implemented methods, and method declarations (for abstract classes):

$$\begin{aligned}\mathsf{Class} = {} & (\mathit{name} \colon \mathsf{ClassName}) \times (\mathit{app} \colon \mathsf{App}) \times (\mathit{package} \colon \mathsf{Package}) \times \\ & (\mathit{super} \colon \mathsf{Class}_\perp) \times (\mathit{implements} \colon \mathcal{P}(\mathsf{Interface})) \times (\mathit{fields} \colon \mathcal{P}(\mathsf{Field})) \times \\ & (\mathit{accessFlags} \colon \mathcal{P}(\mathsf{AccessFlag})) \times (\mathit{methods} \colon \mathcal{P}(\mathsf{Method})) \times \\ & (\mathit{methodDeclarations} \colon \mathcal{P}(\mathsf{MethodDeclaration}))\end{aligned}$$

We define the superclass of `java/lang/Object` to be $\perp$, hence the domain $\mathsf{Class}_\perp$ for superclasses. For name domains like ClassName we assume an unlimited supply of unique names.

An interface is similar to a class but with three important differences: in place of a superclass it has a set of super-interfaces ($\mathit{super} \colon \mathcal{P}(\mathsf{Interface})$), instead of implemented interfaces it symmetrically has its implementing classes ($\mathit{implementedBy} \colon \mathcal{P}(\mathsf{Class})$), and, finally, instead of implemented methods it possibly has one method, a class constructor ($\mathit{clinit} \colon \mathsf{Method}_\perp$). Class constructors are needed to initialise static fields on classes and interfaces and are consequently the only methods that can be implemented directly in interfaces.

[10] See `http://source.android.com/source/downloading.html`. Last accessed 14 December 2011.

A field of a class or an interface has a name, the class or interface where it is defined, a type, an indication of whether it is a static field or not, and then its access flags:

$$\begin{aligned}\mathsf{Field} = {} & (\mathit{name}\colon \mathsf{FieldName}) \times (\mathit{class}\colon \mathsf{Class} \cup \mathsf{Interface}) \times \\ & (\mathit{type}\colon \mathsf{Type}) \times (\mathit{initialValue}\colon \mathsf{Prim} \cup \{\texttt{null}\}) \times \\ & (\mathit{isStatic}\colon \mathsf{Bool}) \times (\mathit{accessFlags}\colon \mathcal{P}(\mathsf{AccessFlag}))\end{aligned}$$

A method signature specifies how a method can be called: the name of the method, the class or interface where it is declared (though not necessarily implemented), as well as the argument and return types:

$$\begin{aligned}\mathsf{MethodSignature} = {} & (\mathit{name}\colon \mathsf{MethodName}) \times (\mathit{class}\colon \mathsf{Class} \cup \mathsf{Interface}) \times \\ & (\mathit{argTypes}\colon \mathsf{Type}^*) \times (\mathit{returnType}\colon \mathsf{Type} \cup \{\texttt{void}\})\end{aligned}$$

Method *declarations* specify everything about a method except its implementation. They appear in interfaces and abstract classes and besides the method signature specify a *kind*, a set of access flags, and the checked exceptions the method can throw. The kind of a method can be `direct`, which is used for non-overridable methods, i.e., constructors and private or final methods, `static` for static methods (that are not `direct`), and `virtual` for normal, overridable methods including methods specified in interfaces. Access flags indicate accessibility and various properties and include `public`, `private`, `protected`, `final`, and `abstract`.

$$\begin{aligned}\mathsf{MethodDeclaration} = {} & (\mathit{methodSignature}\colon \mathsf{MethodSignature}) \times \\ & (\mathit{kind}\colon \{\texttt{virtual}, \texttt{static}, \texttt{direct}\}) \times \\ & (\mathit{accessFlags}\colon \mathcal{P}(\mathsf{AccessFlag})) \times \\ & (\mathit{exceptionTypes}\colon \mathcal{P}(\mathsf{Class}))\end{aligned}$$

An actual method is a method declaration plus implementation details: a mapping from locations in the method (program counter values) to instructions[11], the number of registers used for local variables[12], a set of exception handlers, and a mapping from locations of data tables in the bytecode to the contents of these tables[13]:

$$\begin{aligned}\mathsf{Method} = {} & (\mathit{methodDeclaration}\colon \mathsf{MethodDeclaration}) \times \\ & (\mathit{instructionAt}\colon \mathsf{PC} \to \mathsf{Instruction}) \times \\ & (\mathit{numLocals}\colon \mathbb{N}_0) \times (\mathit{handlers}\colon \mathbb{N}_0 \to \mathsf{ExcHandler}) \times \\ & (\mathit{tableAt}\colon \mathsf{PC} \to \mathsf{DataTable})\end{aligned}$$

For methods, i.e., elements from Method, the class found in the method signature of the method declaration specifies the class (or interface, in the case of `clinit`) where the method is *implemented*. For convenience we introduce unambiguous shorthands such as $m.\mathit{kind}$ for $m.\mathit{methodDeclaration}.\mathit{kind}$ where $m \in \mathsf{Method}$.

[11] Instruction is simply the set of generalised instructions as given in Appendix A.
[12] Dalvik uses registers instead of an operand stack.
[13] Data tables consist of hardcoded array data and switch tables.

3.2. Semantic Domains

Having defined the domains for the overall structure of an app, what remains is to define the semantic domains. The most fundamental semantic domain is that for values. In Dalvik values can be either primitive values or references:

$$\mathsf{Val} = \mathsf{Prim} + \mathsf{Ref}$$

For our purpose, it is sufficient to let primitive values consist only of numbers: $\mathsf{Prim} = \mathbb{Z}$. References are either the `null` reference or an abstract heap location that does not need to be defined in further detail[14]: $\mathsf{Ref} = \mathsf{Location} \cup \{\texttt{null}\}$.

Local registers are formalised as a map from register names to values, with $\bot$ denoting undefined register contents: $\mathsf{LocalReg} = (\mathbb{N}_0 \cup \{\texttt{retval}\}) \to \mathsf{Val}_\bot$. Note that a special register, the `retval` register, is used to transfer return values from invoked methods to the invoking methods[15].

For convenience we use two distinct domains to formalise the heap by splitting it into one for static fields and one for dynamic objects and arrays:

$$\mathsf{StaticHeap} = \mathsf{Field} \to \mathsf{Val} \qquad \text{and} \qquad \mathsf{Heap} = \mathsf{Ref} \to (\mathsf{Object} + \mathsf{Array})$$

Objects are formalised as a domain with a class and a mapping from (object) fields to values. Furthermore, in anticipation we add an annotation component, called *origin*, that records the program point at which the object was created. It is important to note that this has no effect on the semantics, i.e., the actual execution of a program, but is only needed to facilitate the formalisation and proof of correctness for the analysis developed in Section 4:

$$\mathsf{Object} = (\mathit{class}\colon \mathsf{Class}) \times (\mathit{field}\colon \mathsf{Field} \to \mathsf{Val}) \times (\mathit{origin}\colon \mathsf{Method} \times \mathsf{PC})$$

For arrays we do not track the creation point, merely the type, length, and valuation:

$$\mathsf{Array} = (\mathit{type}\colon \mathsf{ArrayType}) \times (\mathit{length}\colon \mathbb{N}_0) \times (\mathit{value}\colon \mathbb{N}_0 \to \mathsf{Val})$$

We can then define stack frames to consist of a method and a program counter value, i.e., a uniquely determined program point, and the local registers:

$$\mathsf{Frame} = \mathsf{Method} \times \mathsf{PC} \times \mathsf{LocalReg}$$

Special frames, *exception frames* containing the location of its corresponding exception object on the heap and the address[16] of the instruction that threw the exception, are introduced for tracking exceptions that are not handled locally:

$$\mathsf{ExcFrame} = \mathsf{Location} \times \mathsf{Method} \times \mathsf{PC}$$

[14] Dalvik does not support pointer arithmetic so it will suffice to know that we can model an arbitrary number of unique locations.

[15] Besides the `retval` register, Dalvik supports 2^{16} numbered regular registers that we generalise to $\mathbb{N}_0$.

[16] In general we refer to a pair consisting of a method and a program counter value ($\mathsf{PC} = \mathbb{N}_0$) as an address. In contexts where the method is given, we sometimes use address about the program counter value alone.

This leads to the following definition of call stacks as a sequence of frames except that the top frame may be an exception frame representing an as of yet unhandled exception:

$$\mathsf{CallStack} = (\mathsf{Frame} + \mathsf{ExcFrame}) \times \mathsf{Frame}^*$$

Together the heaps and the callstack comprise a semantic configuration:

$$\mathsf{Configuration} = \mathsf{StaticHeap} \times \mathsf{Heap} \times \mathsf{CallStack}$$

3.3. 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 rules for a few central instructions, first the basic `move` instruction:

$$\frac{m.\mathit{instructionAt}(pc) = \texttt{move}\ \ v_1\ \ v_2}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v_1 \mapsto R(v_2)]\rangle :: SF\rangle}$$

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, of the top frame are unchanged. Only the top frame of the call stack is affected: the program counter is incremented, and the register valuation is updated such that the destination register now maps to the value of the source register.

The `invoke-virtual` instruction is more involved:

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

It 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 "this" pointer). The location of the method is resolved using the auxiliary function *resolveMethod* as explained below and the resulting 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 first $m'.\mathit{numLocals}$ registers are mapped to $\bot_{\mathsf{Val}}$ such that they are initially undefined. The arguments are then mapped into the next registers. To handle dynamic dispatch, we define the following function to search for a method matching the given method signature in the ancestry of the given class:

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

where $\mathit{meth} \triangleleft m$ 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.

The `return` instruction pops off the top frame, advances the program counter, and passes on the return value by updating the `retval` register:

$$\frac{m.\mathit{instructionAt}(pc) = \texttt{return}\ \ v}{A \vdash \langle S, H, \langle m, pc, R\rangle :: \langle m', pc', R'\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', pc'+1, R'[\texttt{retval} \mapsto R(v)]\rangle :: SF\rangle}$$

The Dalvik instruction set also contains some specialised instructions, for example `const-class` which is a shortcut to create a `java/lang/Class` instance representing a specified class. This instruction is typically used in conjunction with reflection which we treat in Section 5. A `java/lang/Class` object has a field called `name` which refers to a `java/lang/String` object with the name of the class. The instruction therefore creates a `Class` and a `String` and sets the `String` field `value` to the (character array) name of the given class and maps the field `name` on the `Class` to the newly created string:

$$\frac{\begin{array}{c} m.\mathit{instructionAt}(pc) = \texttt{const-class}\ \ v\ \ cl \\ \mathit{findClassObject}(H, cl) = \bot \\ (H', loc_c) = \mathit{newObject}(H, \texttt{java/lang/Class}, m, pc) \qquad o_c = H'(loc_c) \\ (H'', loc_s) = \mathit{newObject}(H', \texttt{java/lang/String}, m, pc) \qquad o_s = H''(loc_s) \\ H''' = H''[loc_c \mapsto o_c[\mathit{field} \mapsto o_c.\mathit{field}[\texttt{name} \mapsto loc_s]], \\ \qquad loc_s \mapsto o_s[\mathit{field} \mapsto o_s.\mathit{field}[\texttt{value} \mapsto cl.\mathit{name}]]] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H''', \langle m, pc+1, R[v \mapsto loc_c]\rangle :: SF\rangle}$$

The auxiliary function *findClassObject* searches the heap for a class object representing the given class. It returns the object location if the object exists, and returns $\bot$ otherwise.

The new objects are created using the auxiliary function *newObject*, also used in the semantics of the `new-instance` instruction, which returns the modified heap along with a fresh reference to the newly created object:

$$\begin{array}{l} \mathit{newObject}\colon \mathsf{Heap} \times \mathsf{Class} \times \mathsf{Method} \times \mathsf{PC} \to \mathsf{Heap} \times \mathsf{Ref} \\ \mathit{newObject}(H, cl, m, pc) = (H', loc) \end{array}$$

The produced heap and location have the following properties: $loc \notin \mathrm{dom}(H)$, $H' = H[loc \mapsto o]$, $o \in \mathsf{Object}$, $o.\mathit{class} = cl$, and $o.\mathit{origin} = (m, pc)$. Note in particular, that the current program point is recorded in the *origin* field of the newly created object.

Class objects are unique[17], so if a class object with the desired class was already present on the heap, that one is returned instead of creating a new one:

$$\frac{\begin{array}{c} m.\mathit{instructionAt}(pc) = \texttt{const-class}\ \ v\ \ cl \\ \mathit{findClassObject}(H, cl) = loc_c \neq \bot \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[v \mapsto loc_c]\rangle :: SF\rangle}$$

3.4. Exceptions

Exceptions can be thrown either explicitly using the `throw` instruction, or by the system in case of a runtime error, such as a `null` pointer dereference. Both situations can be seen directly in the `throw` instruction whose semantics depends on its argument:

[17] This only holds for classes loaded by the same class loader. Since our analysis is defined only for the standard class loader, this is not a problem here.

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

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{throw}\ \ v \\ R(v) = \texttt{null} \qquad (H', loc_e) = newObject(H, \texttt{NullPointerException}) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle loc_e, m, pc\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

In both situations an exception frame is pushed on the stack and the next step depends on whether an appropriate exception handler is available in the method. If one is available, the exception frame is discarded and its exception is put in the `retval` register of the current method which also jumps to the program counter of the handler:

$$\frac{cl = H(loc_e).class \qquad \mathit{findHandler}(m, pc, cl) = pc' \neq \bot}{\begin{array}{l} A \vdash \langle S, H, \langle loc_e, m_e, pc_e\rangle :: \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \\ \qquad \langle S, H, \langle m, pc', R[\texttt{retval} \mapsto loc_e]\rangle :: SF\rangle \end{array}}$$

If no exception handler is found, the frame of the currently executing method is discarded but the exception frame retained such that the search continues recursively among the handlers of the next method:

$$\frac{cl = H(loc_e).class \qquad \mathit{findHandler}(m, pc, cl) = \bot}{A \vdash \langle S, H, \langle loc_e, m_e, pc_e\rangle :: \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle loc_e, m, pc\rangle :: SF\rangle}$$

An exception handler has 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:

$$\begin{array}{l} \mathsf{ExcHandler} = (\mathit{catchType} \colon \mathsf{Class}_\bot) \times (\mathit{handlerAddr} \colon \mathsf{PC}) \times \\ \qquad (\mathit{startAddr} \colon \mathsf{PC}) \times (\mathit{endAddr} \colon \mathsf{PC}) \end{array}$$

The auxiliary function *findHandler* finds the exception handler matching the location in the method and the given exception class, or $\bot$ if no appropriate handler is available. It is itself defined using two auxiliary predicates.

The first, *canHandle*, determines whether an exception handler (h) can handle the given exception (cl_e) for the specific location (pc) in the method:

$$\mathit{canHandle}(h, pc, cl_e) \ \equiv\ h.\mathit{startAddr} \leq pc \leq h.\mathit{endAddr} \wedge cl_e \preceq h.\mathit{catchType}$$

The catch type of a catch-all handler, corresponding to a Java `finally` clause, is $\top$. The *canHandle* predicate is used in another predicate, *isFirstHandler*, which formalises whether or not the given handler (specified by a list of handlers, η, and an index, i) is the correct handler according to indices supplied by the compiler:

$$\begin{array}{rcl} \mathit{isFirstHandler}(\eta, i, pc, cl_e) & \equiv & \mathit{canHandle}(\eta(i), pc, cl_e) \wedge \\ & & (\forall j \colon \mathit{canHandle}(\eta(j), pc, cl_e) \ \Rightarrow\ j \leq i) \end{array}$$

The indices represent an ordering of the handlers that is used to determine which of two (or more) competing handlers should be used when an exception is thrown in their common region of responsibility. The highest index corresponds to the most specific handler.

The function *findHandler* uses *isFirstHandler* to find the address (in this case only the program counter value because the method is given) of the first handler or return $\bot$ if no handler is available:

$$\begin{array}{l} \mathit{findHandler}(m, pc, cl_e) = \\ \quad \begin{cases} \eta(i).\mathit{handlerAddr} & \text{if } \eta = m.\mathit{handlers} \wedge \mathrm{dom}(\eta) \neq \emptyset \wedge \\ & \quad \exists i \colon \mathit{isFirstHandler}(\eta, i, pc, cl_e) \\ \bot & \text{otherwise} \end{cases} \end{array}$$

4. Control Flow Analysis

In the following we give an overview of the control flow analysis we have designed for Dalvik. Control flow analysis is an essential component for building more advanced and specialised analyses, e.g., access control and information-flow analyses. But control flow analysis is also useful on its own, e.g., to find methods that are never called. The control flow analysis also has to take advanced control flow concepts such as dynamic dispatch, exceptions and reflection into account since these are integral to Dalvik. The former two are briefly described below while the latter is dealt with in detail in Section 5.

Our control flow analysis is specified as a *flow logic* [20]. 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 sound. Many other frameworks for program analysis exist, but the combination of structural operational semantics and flow logic has proven to be flexible and easy to use for both theoretical developments as well as for implementation. In particular, a similar approach has been used to specify, reason about, and implement numerous analyses for the Java bytecode language [11] which is similar in nature to Dalvik. A more detailed comparison with other approaches is outside the scope of this paper.

4.1. Abstract Domains

Before we can define the flow logic judgements for the analysis itself, we first need to define the abstract domains over which the analysis works. The abstract domains are used in the analysis to represent abstractions of runtime values and they closely follow the structure of the underlying semantic domains.

We start by defining the most basic abstract domain, namely that for values. An abstract value, similar to a concrete (semantic) value, can be either an abstract primitive value or an abstract reference:

$$\overline{\mathsf{Val}} = \overline{\mathsf{Prim}} + \overline{\mathsf{Ref}} \qquad \text{and} \qquad \widehat{\mathsf{Val}} = \mathcal{P}(\overline{\mathsf{Val}})$$

For our analysis, it is sufficient to model abstract primitive values simply as integers, $\overline{\mathsf{Prim}} = \mathsf{Prim} = \mathbb{Z}$. Note that we use overlined domain names to indicate an abstract domain based on a concrete domain with the same name while domains with a "hat" (like $\widehat{\mathsf{Val}}$) indicate domains that are complete lattices. The latter property is important for ensuring that the flow logic specification is well-defined.

Dalvik has native support for string constants that we model simply as $\widehat{\overline{\mathsf{String}}} = \mathcal{P}(\mathsf{String})$ where we assume the existence of a primitive string domain String. The native strings are different from the character arrays that live in `java/lang/String` objects in the Java standard library and therefore a

Dalvik native string cannot, as such, be used in place of a "proper" Java string. However, for the analysis we may consider the (abstract) native strings to be contained in the (abstract) Java string objects. An example of this is shown in the analysis of the `const-class` instruction in Section 4.2.

Abstract references are either object references, array references, or a `null` reference:

$$\overline{\mathsf{Ref}} = \overline{\mathsf{ObjRef}} + \overline{\mathsf{ArrRef}} + \{\texttt{null}\}$$

For object references we take an approach similar to the *textual object graphs* in [31]: we model all runtime object references as the class of the runtime object combined with the syntactic creation point of the object. Array references are formalised simply as the type of the array, i.e., the analysis merges all information about arrays of the same type:

$$\overline{\mathsf{ObjRef}} = \mathsf{Class} \times \mathsf{Method} \times \mathsf{PC} \qquad \text{and} \qquad \overline{\mathsf{ArrRef}} = \mathsf{ArrayType}$$

Note that here we directly use the concrete program structure defined in Section 3 to model the abstract program structure. To enhance readability, specific instances of object and array references are written '$(\mathsf{ObjRef}\ cl, m, pc)$' and '$(\mathsf{ArrRef}\ t)$' respectively. In anticipation of later developments, we introduce the following notation for extracting all (abstract) references to a particular class:

$$X|_{cl} = \{(\mathsf{ObjRef}\ (cl', m, pc)) \in X |\ cl = cl'\}$$

In order to achieve sufficient precision, the analysis tracks the value of individual registers at every program point, including any return values produced by a method. This leads to the following abstract domains for program points and registers:

$$\overline{\mathsf{PC}} = \mathsf{Method} \times (\mathsf{PC} + \mathsf{END}) \qquad \text{and} \qquad \overline{\mathsf{Register}} = \mathsf{Register} \cup \{\texttt{retval}, \mathsf{END}\}$$

The token 'END' is introduced in the analysis as both a special program counter value and as a special register such that the set of values returned by a method m can be easily referenced.

The domain for tracking register values can then be formalised as a mapping for every (abstract) program point from registers to abstract values:

$$\widehat{\mathsf{LocalReg}} = \overline{\mathsf{PC}} \to \overline{\mathsf{Register}} \to \widehat{\mathsf{Val}}$$

For $\hat{R} \in \widehat{\mathsf{LocalReg}}$ we shall write $\hat{R}(m, \mathsf{END})$ as shorthand for $\hat{R}(m, \mathsf{END})(\mathsf{END})$, i.e., the (abstract) return value produced by method m.

Note that tracking the values of local registers at each program point leads to a control flow analysis that is intra-procedurally *flow sensitive*, i.e., flow sensitive within methods. As already mentioned, this is necessary to achieve sufficient precision for the analysis to be useful. However, it also enables analyses that are intra-procedurally state based such as tracking (within a method) which classes have been initialised.

The subset ordering on $\widehat{\mathsf{Val}}$ can be extended point-wise to an ordering on $\widehat{\mathsf{LocalReg}}$. Let $\hat{R}_1, \hat{R}_2 \in \widehat{\mathsf{LocalReg}}$, $a_1, a_2 \in \overline{\mathsf{PC}}$, and define

$$\begin{aligned} \hat{R}_1 \sqsubseteq \hat{R}_2 \quad &\text{iff} \quad \forall a \in \mathrm{dom}(\hat{R}_1) \colon \hat{R}_1(a) \sqsubseteq \hat{R}_2(a) \\ \hat{R}_1(a_1) \sqsubseteq \hat{R}_2(a_2) \quad &\text{iff} \quad \forall r \in \mathrm{dom}(\hat{R}_1(a_1)) \colon \hat{R}_1(a_1)(r) \sqsubseteq \hat{R}_2(a_2)(r) \end{aligned}$$

In anticipation of the flow logic specification, we define the following notation:

$$\hat{R}_1(a_1) \sqsubseteq_X \hat{R}_2(a_2) \quad \text{iff} \quad \forall r \in \mathrm{dom}(\hat{R}_1(a_1)) \setminus X \colon \hat{R}_1(a_1)(r) \sqsubseteq \hat{R}_2(a_2)(r)$$

The semantic heap is modelled using two abstract domains: one that tracks the static fields of objects, called the *static heap*, and one that tracks references to objects and arrays:

$$\widehat{\mathsf{StaticHeap}} = \mathsf{Field} \to \widehat{\mathsf{Val}} \qquad \text{and} \qquad \widehat{\mathsf{Heap}} = \overline{\mathsf{Ref}} \to (\widehat{\mathsf{Object}} + \widehat{\mathsf{Array}})$$

The abstract domains for objects and arrays can now be defined:

$$\widehat{\mathsf{Object}} = \mathsf{Field} \to \widehat{\mathsf{Val}} \qquad \text{and} \qquad \widehat{\mathsf{Array}} = \widehat{\mathsf{Val}}$$

We could model the structure of arrays but due to imprecise indexes it is of little benefit to the overall precision of the analysis.

The last abstract domain needed for our analysis is needed to track exceptions that are not handled locally within a method:

$$\widehat{\mathsf{ExcCache}} = \mathsf{Method} \to \mathcal{P}(\overline{\mathsf{ObjRef}})$$

Finally, we are now able to define the abstract domain for the overall analysis:

$$\widehat{\mathsf{CFA}} = \widehat{\mathsf{StaticHeap}} \times \widehat{\mathsf{Heap}} \times \widehat{\mathsf{LocalReg}} \times \widehat{\mathsf{ExcCache}}$$

In the following we specify what it means for an element of the above domain to be a sound analysis of a given program.

4.2. Flow Logic Specification

With all the relevant abstract domains defined, we can now specify the flow logic judgements that define our control flow analysis. Flow logic is a specification-oriented approach to program analysis. Analyses are specified by defining, for every instruction, a *judgement* that formalises when an analysis result is a sound approximation of (the effects of) that instruction [20].

In the remainder of this section, we illustrate the analysis by discussing the judgements for a few interesting instructions and refer to [13] for the full specification. We start with the judgement for the `move` instruction which is one of the simplest. Recall that after a `move` instruction is executed, the destination register (v_1 below) contains the value from the source register (v_2 below) while all others are unchanged. This results in the following judgement:

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

To improve readability we adopt the convention that conditions in judgements are implicitly joined by conjunction, i.e., they must all hold in order for the judgement to hold. As a further convenience, we use indentation to indicate the scope of logical variables bound by a quantifier (used in later judgements).

In words, the above judgement states that $(\hat{S}, \hat{H}, \hat{R}, \hat{E}) \in \widehat{\mathsf{CFA}}$ is a sound analysis result for the effects of the `move` instruction located in method m at

address pc if and only if the following two conditions are fulfilled. The first condition ($\hat{R}(m, pc)(v_2) \sqsubseteq \hat{R}(m, pc+1)(v_1)$) formalises that the set of abstract values in the source register v_2, at the *current* address (m, pc), must also be present in the set of abstract values for the destination register v_1 at the *following* address $(m, pc+1)$. The latter condition ($\hat{R}(m, pc) \sqsubseteq_{\{v_1\}} \hat{R}(m, pc+1)$) requires that all registers, except the destination register v_1 which was updated explicitly, contains *at least* the same abstract values at the next address as at the current address.

The judgement for the `invoke-virtual` instruction is somewhat more involved: for each possible object the method can be called on (these can be found in register v_1 containing the "this" pointer), 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). In addition, the judgement must also deal with exceptions: both the null pointer exception that the instruction may throw itself as well as those that are thrown in but not handled by the invoked method. The latter are tracked using the exception component of the analysis $\hat{E}$. Exceptions and exception handling is discussed in more detail below:

$$
\begin{array}{l}
(\hat{S}, \hat{H}, \hat{R}, \hat{E}) \models (m, pc) \colon \texttt{invoke-virtual}\ \ v_1 \ldots v_n\ \ meth \\
\quad \text{iff} \quad \forall(\mathsf{ObjRef}\ (cl, m_t, pc_t)) \in \hat{R}(m, pc)(v_1) \colon \\
\qquad m' = resolveMethod(meth, cl) \\
\qquad \forall 1 \leq i \leq n \colon \hat{R}(m, pc)(v_i) \sqsubseteq \hat{R}(m', 0)(m'.numLocals - 1 + i) \\
\qquad m'.returnType \neq \texttt{void} \Rightarrow \hat{R}(m', \mathsf{END}) \sqsubseteq \hat{R}(m, pc+1)(\texttt{retval}) \\
\qquad \forall oref_e \in \hat{E}(m') \colon \mathsf{HANDLE}_{(\hat{R}, \hat{E})}(oref_e, (m, pc)) \\
\quad\ \ \hat{R}(m, pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc+1) \\
\quad\ \ \mathsf{HANDLE}_{(\hat{R}, \hat{E})}((\mathsf{ObjRef}\ (\texttt{NullPointerException}, m, pc)), (m, pc))
\end{array}
$$

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 following auxiliary predicate, also used in the above judgement, formalises the exception handling semantics:

$$
\begin{array}{l}
\mathsf{HANDLE}_{(\hat{R}, \hat{E})}((\mathsf{ObjRef}\ (cl_e, m_e, pc_e)), (m, pc)) \equiv \\
\quad findHandler(m, pc, cl_e) = pc' \neq \bot \Rightarrow \\
\qquad \{\mathsf{ObjRef}\ (cl_e, m_e, pc_e)\} \subseteq \hat{R}(m, pc')(\texttt{retval}) \\
\qquad \hat{R}(m, pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc') \\
\quad findHandler(m, pc, cl_e) = \bot \Rightarrow \\
\qquad \{\mathsf{ObjRef}\ (cl_e, m_e, pc_e)\} \subseteq \hat{E}(m)
\end{array}
$$

With the above predicate, it is trivial to define the analysis for the `throw` instruction:

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

As the final example we specify the judgement for the `const-class` instruction, illustrating both how to handle specialised Dalvik instructions as well as how

fields in runtime objects are accessed. Also note how shorthands are introduced for the long names of the Java API:

$$
\begin{array}{l}
(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\colon \texttt{const-class}\ \ v\ \ cl \\
\quad \text{iff} \quad oref_{\texttt{Str}} = \mathsf{ObjRef}\ (\texttt{java/lang/String}, m, pc) \\
\qquad\quad oref_{\texttt{Cl}} = \mathsf{ObjRef}\ (\texttt{java/lang/Class}, m, pc) \\
\qquad\quad \{cl.name\} \subseteq \hat{H}(oref_{\texttt{Str}})(\texttt{value}) \\
\qquad\quad \{oref_{\texttt{Str}}\} \subseteq \hat{H}(oref_{\texttt{Cl}})(\texttt{name}) \\
\qquad\quad \{oref_{\texttt{Cl}}\} \subseteq \hat{R}(m,pc+1)(v) \\
\qquad\quad \mathrm{dom}(\hat{H})|_{\texttt{java/lang/Class}} \subseteq \hat{R}(m,pc+1)(v) \\
\qquad\quad \hat{R}(m,pc) \sqsubseteq_{\{v\}} \hat{R}(m,pc+1)
\end{array}
$$

Following the semantics of the instruction (discussed in Section 3) the class name given as argument cl is stored in the `value` field of a newly created `String` object, the reference to which is stored in the `name` field of a `Class` object, which is stored in the destination register v. Finally, to take possible object sharing into account, we copy all abstract references to `java/lang/Class` found in the (domain of) the abstract heap to the result register v.

4.3. Correctness of the Analysis

We define and prove the semantic soundness of the analysis by means of a subject reduction result. To this end the relationship between the concrete and abstract domains is formalised. We use representation functions to map semantic domains to their abstract domains while keeping the best possible representation. For semantic domains that do not have corresponding abstract counterparts, correctness relations are introduced. We treat the analysis without exceptions. While this simplifies the presentation, the proof extends in a straightforward way to the full semantics.

4.3.1. Representation Functions and Correctness Relations

As values can be either primitive values or references (see Section 4.1) representation of a concrete value is delegated depending on type:

$$
\beta^H_{\mathsf{Val}}(v) = \begin{cases} \beta_{\mathsf{Prim}}(v) & \text{if } v \in \mathsf{Prim} \\ \beta^H_{\mathsf{Ref}}(v) & \text{if } v \in \mathsf{Ref} \\ \bot_{\widehat{\mathsf{Val}}} & \text{if } v = \bot_{\mathsf{Val}} \end{cases}
$$

Since the representation of a reference value depends on the heap as well as the value, as we shall soon see, the representation function for values in general is parameterised by it as well.

Primitive values are represented as integers in both the concrete and abstract domains ($\widehat{\mathsf{Prim}} = \mathcal{P}(\overline{\mathsf{Prim}}) = \mathcal{P}(\mathsf{Prim}) = \mathcal{P}(\mathbb{Z})$) so the best representation is to inject a value into its singleton set: $\beta_{\mathsf{Prim}}(p) = \{p\}$.

Since concrete references by themselves are not distinguished by whether they reference an object or an array, the concrete heap is necessary as a parameter to their representation function:

$$
\beta^H_{\mathsf{Ref}}(loc) = \begin{cases} \{\mathsf{ObjRef}\ (cl,m,pc)\} & \text{if } H(loc) \in \mathsf{Object} \wedge \\ & \quad H(loc).class = cl \wedge H(loc).origin = (m,pc) \\ \{\mathsf{ArrRef}\ t\} & \text{if } H(loc) \in \mathsf{Array} \wedge H(loc).type = t \\ \{\texttt{null}\} & \text{if } loc = \texttt{null} \end{cases}
$$

Register contents are modelled as a mapping from registers to values (see Section 3.2), so a register valuation $R \in \mathsf{LocalReg}$ can be represented by representing the values that the registers map to. Similarly for a static heap $S \in \mathsf{StaticHeap}$:

$$\beta^H_{\mathsf{LocalReg}}(R) = \beta^H_{\mathsf{Val}} \circ R \qquad \text{and} \qquad \beta^H_{\mathsf{StaticHeap}}(S) = \beta^H_{\mathsf{Val}} \circ S$$

The dynamic heap is more intricate. Since the analysis abstracts object references into classes, an abstract object reference may correspond to more than one concrete location. To handle this the representation function for heaps is also parameterised on an abstract reference, and the representations of the corresponding concrete objects are joined:

$$\beta_{\mathsf{Heap}}(H)(\mathsf{ObjRef}\ (cl, m, pc)) = \bigsqcup_{\substack{loc \in \mathrm{dom}(H) \\ \beta^H_{\mathsf{Ref}}(loc) = (\mathsf{ObjRef}\ (cl, m, pc))}} \beta^H_{\mathsf{Object}}(H(loc))$$

where the representation function for objects is defined as follows:

$$\beta^H_{\mathsf{Object}}(o) = \beta^H_{\mathsf{Val}} \circ (o.\mathit{field})$$

The representation function for array references is defined in a similar way.

Correctness Relations. A stack frame (of domain Frame) is correctly represented if the abstract representation of its register valuation is contained in the corresponding abstract register valuation at the right address:

$$\langle m, pc, R\rangle\ \mathcal{R}^H_{\mathsf{Frame}}\ \hat{R} \quad \text{iff} \quad \beta^H_{\mathsf{LocalReg}}(R) \sqsubseteq \hat{R}(m, pc)$$

A call stack is a sequence of stack frames, $SF = F_1 :: \cdots :: F_n$, that is correctly represented if every stack frame of the call stack is correctly represented:

$$SF\ \mathcal{R}^H_{\mathsf{CallStack}}\ \hat{R} \quad \text{iff} \quad \forall 1 \le i \le n\colon F_i\ \mathcal{R}^H_{\mathsf{Frame}}\ \hat{R}$$

We can now state that a complete configuration is correct if the abstract representations of the heaps are contained in their respective corresponding abstract domains:

$$\langle S, H, SF\rangle\ \mathcal{R}_{\mathsf{Conf}}\ (\hat{S}, \hat{H}, \hat{R}) \quad \text{iff} \quad \begin{array}{l} \beta^H_{\mathsf{StaticHeap}}(S) \sqsubseteq \hat{S} \wedge \\ \beta_{\mathsf{Heap}}(H) \sqsubseteq \hat{H} \wedge \\ SF\ \mathcal{R}^H_{\mathsf{CallStack}}\ \hat{R} \end{array}$$

4.3.2. Subject Reduction Theorem

The theorem states that an analysis result that is correct with respect to a semantic configuration remains correct under semantic reduction.

Theorem 1 (Subject Reduction). *Let* $A \in \mathsf{App}$ *and* $C \in \mathsf{Configuration}$ *such that* $(\hat{S}, \hat{H}, \hat{R}) \in \widehat{\mathsf{CFA}}$, $(\hat{S}, \hat{H}, \hat{R}) \models A$, *and* $A \vdash C \Longrightarrow C'$. *Then*

$$C\ \mathcal{R}_{\mathsf{Conf}}\ (\hat{S}, \hat{H}, \hat{R}) \Rightarrow C'\ \mathcal{R}_{\mathsf{Conf}}\ (\hat{S}, \hat{H}, \hat{R})$$

The proof is a tedious but straightforward case analysis on the semantic reductions. We have proved the theorem for a core of the language, see Appendix D. We expect that the proof can be extended easily to the full language. To handle exceptions and the `return` instruction a condition will have to be added such that the callstack can be trusted to be the result of an actual program execution. A well-formed configuration can be defined as a configuration with such a well-formed callstack, and it would also have to be shown that well-formedness is preserved under semantic reduction.

4.4. Concurrency

There are two Dalvik instructions related to concurrency: `monitor-enter` and `monitor-exit`. They are generated by the compiler when the Java keyword `synchronized` is used and they are used in most apps.

In the concurrent execution model, the execution order of instructions is defined by the Java Memory Model, JSR-133 [23], which formalises how shared variables should be read and written, and how instructions can be re-ordered to execute as-if-serially when they are concurrent [19]. According to unofficial statements by one of the Dalvik developers, Alexey Kryshen, on the Stackoverflow website[18], Dalvik tries to comply with the JSR-133 memory model, though there should be cases where it does not on versions prior to Android 3.0.

In the present work we have formalised neither the semantics nor the analysis of concurrency and the related instructions. However, we conjecture that our control flow analysis can be extended to a sound analysis even for multi-threaded apps since the analysis is *flow insensitive* with respect to heap access. Therefore the analysis already represents an over-approximation of all possible interleavings of all heap accesses, including those that would otherwise be added by allowing multi-threaded apps.

5. Reflection

Reflection allows a program to access class information at runtime, and use this information to create new objects, invoke methods or otherwise change the control flow of the program. When reflection is used, the types involved are usually not known statically. Instead, they are retrieved dynamically from strings. The strings can come from sources such as user input, files included with the app, the Internet, or, in some cases, constant strings in the program. We found that several of the apps in our data set specify constant strings in the program.

The most used method from the Java reflection API is `Method.invoke()`. It is an instance method on the `java/lang/reflect/Method` class used to invoke dynamically resolved methods. An example can be seen in Listing 1 where the method `bar(int)` on the class `pkg.examples.Foo` is invoked on an instance of the class with the argument '3'.

```
1 Class<?> clazz = Class.forName("pkg.examples.Foo");
2 Method method = clazz.getMethod("bar", int.class);
3 Integer result = (Integer) method.invoke(clazz.newInstance(), 3);
```

Listing 1: A method invoked through reflection in Java.

A `Method` object can be retrieved using the instance method `getMethod()` on the Java standard class `java/lang/Class`. The instance of `Class` does not have to represent a class that implements the method in question since it will be resolved with dynamic dispatch like normal method calls. In Listing 1, the `Class` object is retrieved using the static method `Class.forName()` that, given a fully qualified class name, returns a reference to a `Class` instance for the specified class.

[18] See `http://stackoverflow.com/questions/6973667/dalvik-vm-java-memory-model-concurrent-programming-on-android`. Last accessed 29 May 2012.

Another way to obtain a `Class` object is through the Dalvik instruction `const-class` (see Section 3 for the semantics of this instruction). It is generated when the static field `class`, which is found in all Java classes, is accessed. An alternative to `Class.forName("pkg.examples.Foo")` at line 1 in Listing 1 would therefore be `Foo.class`, given that the example code is located in the same package as the `Foo` class and that the class can be found at compile time.

The `Method` objects are mainly retrieved using the methods `getMethod()` and `getMethods()`. The latter returns an array of all public method declarations on a class while the former returns a single object that is found by specifying the name and parameter types of the desired method declaration. The methods `getMethod()` and `getMethods()` only find public method declarations. They both find the method declaration objects by traversing the class hierarchy, starting at the class represented by the `Class` object and searching upwards through superclasses and interfaces. We uncovered and reported an undocumented change in the semantics of this traversal which we discuss in Section 5.5. Developers can also use the `getDeclaredMethod()` and `getDeclaredMethods()` methods which only look in the given class but may also return private methods.

Once a `Method` object has been obtained, it can be used to retrieve information about the method declaration, for example access modifiers, name, and the (checked) exceptions it can throw. Accessing these requires no additional information besides that which is known statically from the method declaration. To invoke the method, an instance of the class or a subclass is required (except for static methods). The receiver object can be any Java object created using the regular Java `new` statement or through the `newInstance()` method on a `Class` object. In addition to creating a new instance, the `newInstance()` method calls the parameterless constructor for the class. To use another constructor, an instance of the `Constructor` class from the reflection API must be used.

5.1. Reflection in the Wild

We have run an automated informal search for constant strings supplied to the two `java/lang/Class` methods `forName()` and `getMethod()`. Of the 150 apps in three randomly selected market categories (*News and Magazines*, *Photography*, and *Productivity*),

- 18.7% of the apps use neither `forName()` nor `getMethod()`
- 17.3% use at least one of the methods and use only constant strings
- 25.3% use exactly one variable string in total for both methods
- 38.7% use more than one variable string in total for both methods.

Here we searched for instances of the two `Class` methods with a `const-string` instruction in the preceding vicinity, writing to the same register read by the method. This approach gives a number of false positives, most prominently those that simple constant propagation would find. But also in subtler cases, as seen in the 37.3% of the apps containing a class named `InstallReceiver`. This class uses `forName()` with a variable string, but only if it equals the constant `"com.google.android.apps.analytics.AnalyticsReceiver"`. So with the techniques we formalise in the following sections, it is possible to analyse more than a third of real-world apps. Handling of (the effects of) simple patterns of reflection usage may increase this number further (as indicated by the large number of apps that only sparingly use variable strings for reflection).

5.2. Usage Patterns

The use-cases for reflection vary from app to app, and Android developers use it for many different things. However, we have observed some patterns in usage, most of which we have found through manual inspection of the bytecode.

Hidden API methods are invoked. Certain features in Android are deliberately hidden such that they are not present in the JAR file for the Android API that app developers use when compiling their Android apps. This is typically done when an API is not yet considered stable or if the underlying functionality is not implemented on all devices intended to support it. A prominent example is the Bluetooth API, most of which was hidden in the early releases of Android. Developers tend to use these features anyway, and use reflection to do so[19] [8] instead of precompiling their own fully featured JAR file for the Android API.

Private API methods and fields are accessed by bypassing access modifiers. Several features of the Android platform are placed in private methods and fields, such as the ability to create a list of text messages from raw SMS data.

Backward compatibility As new versions of Android are often released with new features, developers tend to use reflection to check if certain methods/features exist and call these only when they do. This pattern is even encouraged by Google[20].

JSON and XML is generated and parsed with the use of reflection. Some apps use JSON and XML that contain information about their Java objects, and through reflection generation and parsing can be automated.

Libraries for Android apps are widely available on the Internet, and some of them use reflection. In many apps that use reflection, it is only used by the included libraries.

In [17] a reflection analysis for standard Java is defined and discussed as well as the use of reflection in a number of large open source projects. This study revealed some of the same patterns as those reported here, such as object serialisation and portability/backward-compatibility. However, it was found that reflection was mostly used to create new objects without invoking new methods on them. In contrast, we found that in Android, invoking methods is among the most common uses of reflection. Our findings are however consistent with those reported for Android in [8].

[19] See `http://developer.sonymobile.com/2011/10/28/code-examples-using-hidden-android-apis/`. Last accessed 5 May 2013.

[20] When our original work was done, the description of reflection for backward compatibility was part of the official Android developer's website at `http://developer.android.com/resources/articles/backward-compatibility.html` but the page has since been removed. The same content is now available at the Android Developers Blog, controlled by the Open Handset Alliance, at `http://android-developers.blogspot.com/2009/04/backward-compatibility-for-android.html`.

5.3. Assumptions

Static analysis of the reflection API is not possible in all cases. We therefore make the following assumptions:

- All classes used through reflection are known statically, such that its components can be analysed. In other words, we assume that dynamic class loading is not used.
- The program does not use a non-default class loader, as this could change the behaviour of `Class.forName()` and related methods. As mentioned in Section 2, class loaders are used in 13.1% of the apps to either load or define new classes. This raised other problems with regards to static analysis, and they were considered out of scope for this project.
- The strings used to obtain `Method` and `Class` objects used for reflection can be determined statically. This presumption only holds for some of the studied apps. A preliminary rough analysis of our data set shows that 57% and 44% of all calls to `Class.forName()` and `Class.getMethod()` respectively use locally defined constant strings. Of the studied apps, a total of 19% exclusively use locally defined constant strings for all calls to the methods `Class.forName()` and `Class.getMethod()`.

These last results are based on a sampling of `const-string` instructions syntactically appearing shortly before the invocation of the reflection methods. They are not based on the limited but inter-procedural data flow capabilities of our control flow analysis. Even with an inter-procedural analysis, improving the numbers requires the ability to track strings across collection APIs such as `java/util/ArrayList` and follow string manipulation such as that of the `java/lang/StringBuilder` class. An example of the former is discussed briefly in Section 8. For the latter, existing string analyses such as [5, 15, 26] may prove useful.

The operational semantics specified so far all represent single Dalvik instructions. We now change focus and specify operational semantics and flow logic judgements to represent Java API method calls. This has two reasons: (1) Using our analysis to analyse the Dalvik instructions that the API method consists of is not possible for APIs implemented natively in C or C++. (2) Analysing reflection at the level of API calls makes it simpler and more precise to recognise the special patterns employed when using reflection. Therefore, we have chosen to specify the operational semantics for the API methods as if they each were single, although advanced, Dalvik instructions. Most of the reflection API calls can throw exceptions, but for presentation purposes we only describe these in some cases.

5.4. Class Objects

When the Java method `Class.forName(string)` is used, it generates the Dalvik instruction `invoke-static` with the signature

```
Ljava/lang/Class;->forName(Ljava/lang/String;)Ljava/lang/Class;
```

For readability in the semantics and judgements, we identify such specialised calls using *meth* = `java/lang/Class->forName`, with the method signature

shortened. For this method the `invoke-static` instruction takes one register argument: a reference to a string that identifies the class or interface one wants to refer to as a `Class` object. As is the case with the `const-class` instruction (see Section 3.3), the corresponding class object may already exist. For this rule we additionally use the auxiliary function *lookupClass* to find the class in the semantic Class domain from the supplied fully qualified class name. If no class object exists a new one is allocated on the heap and the `name` field is updated to point to the string reference of the class name:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-static}\ v_1\ \ meth \\ meth = \texttt{java/lang/Class->forName} \qquad loc = R(v_1) \qquad o = H(loc) \\ cl = lookupClass(o.field(\texttt{value})) \in \mathsf{Class} \qquad findClassObject(H, cl) = \bot \\ (H', loc_{cl}) = newObject(H, \texttt{java/lang/Class}, m, pc) \qquad o_{cl} = H'(loc_{cl}) \\ o'_{cl} = o_{cl}[field \mapsto o_{cl}.field[\texttt{name} \mapsto loc]] \qquad R' = [\texttt{retval} \mapsto loc_{cl}] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H'[loc_{cl} \mapsto o'_{cl}], \langle m, pc+1, R'\rangle :: SF\rangle}$$

Since no new frame is pushed on the stack, the program counter is incremented directly. If the class object already exists, a reference to it is returned:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-static}\ v_1\ \ meth \\ meth = \texttt{java/lang/Class->forName} \qquad loc = R(v_1) \qquad o = H(loc) \\ cl = lookupClass(o.field(\texttt{value})) \in \mathsf{Class} \qquad findClassObject(H, cl) = loc_{cl} \neq \bot \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc+1, R[\texttt{retval} \mapsto loc_{cl}]\rangle :: SF\rangle}$$

For the analysis, register v_1 may contain several values but we can safely ignore anything other than strings, as the API method will only accept a string as an argument. Every string reference in v_1 is transferred to a new location on the heap, into the field `name` on the object identified by the type `java/lang/Class`, the current method and program counter. To take object sharing into account we copy all references to `java/lang/Class` (including the newly created one) to the `retval` register:

$$\begin{array}{l} (\hat{S}, \hat{H}, \hat{R}, \hat{E}) \models (m, pc)\colon \texttt{invoke-static}\ v_1\ \ meth \\ \quad \text{iff} \quad meth = \texttt{java/lang/Class->forName} \\ \qquad oref = \mathsf{ObjRef}\ (\texttt{java/lang/Class}, m, pc) \\ \qquad \forall oref' \in \hat{R}(v_1)|_{\texttt{java/lang/String}}\colon \{oref'\} \subseteq \hat{H}(oref)(\texttt{name}) \\ \qquad \mathrm{dom}(\hat{H})|_{\texttt{java/lang/Class}} \subseteq \hat{R}(m, pc+1)(\texttt{retval}) \\ \qquad \hat{R}(m, pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc+1) \end{array}$$

5.5. Method Objects

A `Method` object represents a method declaration, i.e., an element in the MethodDeclaration domain. This means that `Class.getMethod()` finds a method declaration resolved from the class or interface represented by the `Class` object. Two auxiliary functions are used in the semantics to handle method resolution and object creation respectively: *resolvePublicMethodDeclaration* and *newMethodObject*.

The *resolvePublicMethodDeclaration* function takes as arguments a class name, a method name, and a type signature (in total, a method signature), and searches through the class and interface hierarchy for a matching method declaration. The function only finds methods that are defined as public and which are not constructors.

The function *newMethodObject* is given the method declaration and the existing heap and returns the updated heap and the location of the `Method` object where the relevant fields have been initialised. In fact it creates three new objects: a `Method`, a `Class`, and a `String`, because the field `declaringClass` on the `Method` object references a `Class` where the field `name` references a `String` with the actual class or interface name:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-virtual}\ \ v_1\ \ v_2\ \ v_3\ \ meth \\ meth = \texttt{java/lang/Class->getMethod} \\ clname_o = H(R(v_1)).field(\texttt{name}) \qquad clname = H(clname_o).field(\texttt{value}) \\ mname = H(R(v_2)).field(\texttt{value}) \qquad types = H(R(v_3)).field(\texttt{value}) \\ m = resolvePublicMethodDeclaration(clname, mname, types) \\ m \neq \bot \qquad (H', loc_m) = newMethodObject(H, m) \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H', \langle m, pc+1, R[\texttt{retval} \mapsto loc_m]\rangle :: SF\rangle}$$

We have defined the search order of the *resolvePublicMethodDeclaration* function as follows: The current class, the superclasses of the current class and finally the interface hierarchy of the current class. The interface hierarchies of the superclasses are not searched despite the fact that methods declared in these would be found if standard method resolution was used instead of reflection. This unexpected behaviour is consistent with the Java documentation [22] and the behaviour in Android 2.3. However, the search order has changed in Android 4.0 to be consistent with the "natural" behaviour where interfaces of superclasses are searched as well. This change in behaviour is undocumented and we have reported this as a bug[21]. Regardless of search order, the function is able to find more than one applicable method declaration due to covariant return types. In such cases, the one with the most specific return type is returned, and if a single return type is not more specific than the others, an arbitrary method declaration is returned.

In the analysis, we define *mref* for readability to be the reference to the new `Method` object. For all references to `Class` objects in v_1, there are one or more class names referenced by a `String` on the heap in the field `name`. The set of class names is bound to *clnames*. For each of the `String` references from v_2 the string values (method names) are bound to *mnames*. The set of method names is also put in the field `name` on the heap at *mref*. Furthermore, we use *resolvePublicMethodDeclarationsFromNames* to do a search through the class and interface hierarchy for valid method declarations, similar to the semantic *resolvePublicMethodDeclaration*, but for sets of class and method names. However, it does not take argument types into account since we do not model arrays precisely enough to do a reasonable comparison of the argument types. For each of the resulting method declarations (m'): the class name of the method declaration is created as a `String`, the string reference is put into a (newly created) `Class` object. Taking potential object sharing into account, all references to the `Class` object are copied to the field `declaringClass` for *mref* on the heap. Finally, a reference to the method object is present in the `retval` register:

[21] See the Android Issue Tracker (issue 31485) at `http://code.google.com/p/android/issues/detail?id=31485`.

$$
\begin{array}{l}
(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\colon \texttt{invoke-virtual}\ \ v_1\ \ v_2\ \ v_3\ \ meth \\
\quad \text{iff} \quad meth = \texttt{java/lang/Class->getMethod} \\
\qquad mref = (\mathsf{ObjRef}\ (\texttt{java/lang/reflect/Method}, m, pc)) \\
\qquad \forall oref_{\mathtt{c}} \in \hat{R}(m,pc)(v_1)|_{\texttt{java/lang/Class}}\colon \\
\qquad\quad \forall oref_{\mathtt{s}} \in \hat{H}(oref_{\mathtt{c}})(\texttt{name})|_{\texttt{java/lang/String}}\colon \\
\qquad\qquad clnames = \hat{H}(oref_{\mathtt{s}})(\texttt{value}) \\
\qquad\qquad \forall oref'_{\mathtt{s}} \in \hat{R}(m,pc)(v_2)|_{\texttt{java/lang/String}}\colon \\
\qquad\qquad\quad mnames = \hat{H}(oref'_{\mathtt{s}})(\texttt{value}) \\
\qquad\qquad\quad \{oref'_{\mathtt{s}}\} \subseteq \hat{H}(mref)(\texttt{name}) \\
\qquad\qquad\quad \forall m' \in resolvePublicMethodDeclarationsFromNames(mnames, clnames)\colon \\
\qquad\qquad\qquad \beta(m'.class.name) \sqsubseteq \hat{H}(\mathsf{ObjRef}\ \texttt{java/lang/String}, m, pc)(\texttt{value}) \\
\qquad\qquad\qquad \{\mathsf{ObjRef}\ (\texttt{java/lang/String}, m, pc)\} \subseteq \\
\qquad\qquad\qquad\qquad \hat{H}(\mathsf{ObjRef}\ (\texttt{java/lang/Class}, m, pc))(\texttt{name}) \\
\qquad\qquad\qquad \mathrm{dom}(\hat{H})|_{\texttt{java/lang/Class}} \subseteq \hat{H}(mref)(\texttt{declaringClass}) \\
\qquad \{mref\} \subseteq \hat{R}(m,pc+1)(\texttt{retval}) \\
\qquad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc+1)
\end{array}
$$

5.6. Instantiation

The API method `Class.newInstance()` is used to instantiate new objects through reflection. It requires a `Class` object representing the class to be instantiated. The class must be a regular class, not an interface, abstract class, primitive type or array class. In such cases, an exception is thrown (this is left out of the semantics and analysis to simplify presentation but could easily be added). The `Class` object contains a fully qualified class name and we use the auxiliary function *lookupClass* to find the corresponding class in the semantic Class domain. Next, the new instance is created on the heap using the same function *newObject* as in the regular `new-instance` instruction. Unlike the `new-instance` instruction, `Class.newInstance()` also calls the default constructor for the class being instantiated. We use an auxiliary function *lookupDefaultConstructor* to find this constructor, and if none exists the function will return $\bot$ and an exception should be thrown. The constructor is given registers where the argument register has been initialised to a reference to the newly allocated object. Control is transferred to the constructor by adding a new stack frame, just like regular method invocation, but a reference to the newly allocated object is also put into the `retval` register on the stack frame for the current method. A constructor cannot return a value, and therefore this reference cannot be replaced before control is returned to the current method:

$$
\frac{\begin{array}{c}
m.instructionAt(pc) = \texttt{invoke-virtual}\ \ v_1\ \ meth \\
meth = \texttt{java/lang/Class->newInstance} \\
loc_{cl} = R(v_1) \neq \texttt{null} \qquad o_{cl} = H(loc_{cl}) \\
o_n = H(o_{cl}.field(\texttt{name})) \qquad cl = lookupClass(o_n.field(\texttt{value})) \\
(H', loc) = newObject(H, cl, m, pc) \qquad m' = lookupDefaultConstructor(cl) \neq \bot \\
R' = [0 \mapsto \bot, \ldots, m'.numLocals - 1 \mapsto \bot, m'.numLocals \mapsto H'(loc)]
\end{array}}{\begin{array}{c}
A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \\
\langle S, H', \langle m', 0, R'\rangle :: \langle m, pc, R[\texttt{retval} \mapsto loc]\rangle :: SF\rangle
\end{array}}
$$

The flow logic judgement specifies that for all the `Class` references given to the method, `String` references exist on the heap to specify the class name, and for each of these class names (*clname*) the semantic class must be found using the function *lookupClass*. A reference for each of these classes is put into the `retval` register for the current method, a default constructor is found and the new object reference is placed as an argument to the constructor:

$$
\begin{array}{l}
(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\colon \texttt{invoke-virtual}\ \ v_1\ \ meth \\
\quad \text{iff}\quad meth = \texttt{java/lang/Class->newInstance} \\
\qquad \forall oref \in \hat{R}(m,pc)(v_1)|_{\texttt{java/lang/Class}}\colon \\
\qquad\quad \forall oref' \in \hat{H}(oref)(\texttt{name})|_{\texttt{java/lang/String}}\colon \\
\qquad\qquad \forall clname \in \hat{H}(oref')(\texttt{value})\colon \\
\qquad\qquad\quad cl = lookupClass(clname) \\
\qquad\qquad\quad \{\mathsf{ObjRef}\ (cl,m,pc)\} \subseteq \hat{R}(m,pc+1)(\texttt{retval}) \\
\qquad\qquad\quad m' = lookupDefaultConstructor(cl) \\
\qquad\qquad\quad \{\mathsf{ObjRef}\ (cl,m,pc)\} \subseteq \hat{R}(m',0)(m'.numLocals) \\
\qquad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc+1)
\end{array}
$$

5.7. Method Invocation

Once a `Method` object is created it can be used to invoke the method it represents. The API method `Method.invoke()` takes two arguments besides the `Method` object (v_1) itself: an object reference (v_2) for the receiver object on which the method should be invoked, and an array of arguments (v_3). The receiver object should be `null` if the method is static, and the method implementation will then be resolved from the declaring class in the `Method` object. We do not formalise the invocation on static methods as this is a straightforward simplification of the case with a receiver object. We use the auxiliary function *methodSignature* to extract information from a `Method` object to create a corresponding signature in the semantic MethodSignature domain. The actual method to invoke is resolved using *resolveMethod*, just like in the regular `invoke-virtual` instruction:

$$
\frac{\begin{array}{c}
m.instructionAt(pc) = \texttt{invoke-virtual}\ \ v_1\ \ v_2\ \ v_3\ \ meth \\
meth = \texttt{java/lang/reflect/Method->invoke} \qquad R(v_1) = loc_1 \neq \texttt{null} \\
o_1 = H(loc_1) \qquad o_1.class \preceq \texttt{java/lang/reflect/Method} \\
meth' = methodSignature(H,o_1) \qquad R(v_2) = loc_2 \neq \texttt{null} \qquad o_2 = H(loc_2) \\
R(v_3) = loc_3 \qquad a = H(loc_3) \in \mathsf{Array} \qquad m' = resolveMethod(meth', o_2.class) \\
a' = unboxArgs(a, m'.argTypes, H) \qquad bf = boxingFrame(m'.returnType) \\
R' = [0 \mapsto \bot, \ldots, m'.numLocals - 1 \mapsto \bot, m'.numLocals \mapsto a'.value(0), \ldots, \\
\qquad m'.numLocals + a'.length - 1 \mapsto a'.value(a'.length - 1)]
\end{array}}{A \vdash \langle S,H,\langle m,pc,R\rangle :: SF\rangle \Longrightarrow \langle S,H,\langle m',0,R'\rangle :: bf :: \langle m,pc,R\rangle :: SF\rangle}
$$

Before the arguments are transferred to the resolved method registers they may have to be unboxed: the API method (`Method.invoke()`) receives the arguments in an array with elements of type `Object` (Java `varargs`). This means that if the invoked method has any formal arguments of primitive types, the API method unboxes the primitive values that were boxed before the call occurred. The primitive values are extracted from the box object based on the argument types of the resolved method. We use an auxiliary function, *unboxArgs*, to unbox all the relevant arguments and return an array with the correctly typed

values. These values are then transferred into the relevant registers that are put into a new stack frame along with the method to invoke. The unboxed array, a', is longer than a if any of the unboxed values are of wide data types, i.e., `long` or `double`.

The API method always returns a value of type `Object`, and if the invoked method returns a primitive value it must therefore be boxed by the API method. The return value is not available until the invoked method returns, and therefore we cannot yet box the value. Instead, we add an additional stack frame with a special method to be run after the invoked method. We use an auxiliary function *boxingFrame* to generate this frame. The function takes the return type of the invoked method as an argument, such that the boxing method is able to determine if the return value should be boxed, and what class it should be boxed in. If boxing is to occur, it boxes the return value from the `retval` register and replaces it with a reference to the boxed value. The boxing method then returns as a regular method by incrementing the program counter of the frame below and updating its `retval` register.

In the analysis, for all the `Method` object references in v_1, we use the auxiliary function *methodSignatures* to extract and create all possible method signatures that correspond with the information on the heap for the given `Method` object. All these method signatures must be resolved on all the object references for receiver objects in v_2. We do not store the order of the arguments in the array referenced in v_3, and therefore we cannot determine which of the arguments that must be unboxed. Instead, we transfer all values as they were, as well as unboxing all arguments that are object references, if the class (cl_o) is a class that can be unboxed. The latter is determined by the auxiliary function *isBoxClass*. Depending on the return type of the invoked method, the return value of `Method.invoke()` is either `null` (if void), passed unchanged (if it is already of a reference type) or boxed (if it is of a primitive type). The function *primToBoxClass* translates a return type to the corresponding boxing class, e.g. `int` to `Integer`, and the return value of the method is then boxed by putting the value in the field `value` on the heap for the found class and the current method and program counter. In addition, the same object reference is put in the `retval` register for the next program counter in the current method. Finally, we handle any exceptions that are referenced in the exception cache since the invoked method might have thrown an exception:

$$
\begin{array}{l}
(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\colon \texttt{invoke-virtual}\ v_1\ v_2\ v_3\ meth \\
\quad \text{iff}\quad meth = \texttt{java/lang/reflect/Method->invoke} \\
\qquad \forall oref_{\texttt{M}} \in \hat{R}(m,pc)(v_1)|_{\texttt{java/lang/reflect/Method}}\colon \\
\qquad\quad \forall meth' \in methodSignatures(\hat{H}, oref_{\texttt{M}})\colon \\
\qquad\qquad \forall(\mathsf{ObjRef}\ (cl_r,m_r,pc_r)) \in \hat{R}(m,pc)(v_2)\colon \\
\qquad\qquad\quad m' = resolveMethod(meth', cl_r) \\
\qquad\qquad\quad \{\mathsf{ObjRef}\ (cl_r,m_r,pc_r)\} \subseteq \hat{R}(m',0)(m'.numLocals) \\
\qquad\qquad\quad \forall 1 \le i \le arity(meth')\colon \\
\qquad\qquad\qquad \forall aref \in \hat{R}(m,pc)(v_3) \cap \overline{\mathsf{ArrRef}}\colon \\
\qquad\qquad\qquad\quad \hat{H}(aref) \sqsubseteq \hat{R}(m',0)(m'.numLocals + i) \\
\qquad\qquad\qquad\quad \forall(\mathsf{ObjRef}\ (cl_o,m_o,pc_o)) \in \hat{H}(aref)\colon isBoxClass(cl_o) \Rightarrow \\
\qquad\qquad\qquad\qquad \hat{H}(\mathsf{ObjRef}\ (cl_o,m_o,pc_o))(\texttt{value}) \sqsubseteq \hat{R}(m',0)(m'.numLocals + i) \\
\qquad\qquad\quad m'.returnType = \texttt{void} \Rightarrow \beta(\texttt{null}) \sqsubseteq \hat{R}(m,pc+1)(\texttt{retval}) \\
\qquad\qquad\quad m'.returnType \in \mathsf{RefType} \Rightarrow \hat{R}(m',\mathsf{END}) \sqsubseteq \hat{R}(m,pc+1)(\texttt{retval}) \\
\qquad\qquad\quad m'.returnType \in \mathsf{PrimType} \Rightarrow \\
\qquad\qquad\qquad cl_b = primToBoxClass(m'.returnType) \\
\qquad\qquad\qquad \hat{R}(m',\mathsf{END}) \sqsubseteq \hat{H}(\mathsf{ObjRef}\ (cl_b,m,pc))(\texttt{value}) \\
\qquad\qquad\qquad \{\mathsf{ObjRef}\ (cl_b,m,pc)\} \subseteq \hat{R}(m,pc+1)(\texttt{retval}) \\
\qquad\qquad\quad \forall oref_e \in \hat{E}(m')\colon \mathsf{HANDLE}_{(\hat{R},\hat{E})}(oref_e,(m,pc)) \\
\qquad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc+1)
\end{array}
$$

6. Prototype Implementation

The control flow analysis, specified as flow logic judgements, is not in itself immediately useful for analysing actual Dalvik bytecode apps. In this section we describe a prototype implementation of a tool that converts the flow logic judgements into a Prolog program that can then be executed in order to compute an analysis result.

Our prototype combines several existing tools with our Python-based parser and constraint generator as shown in Figure 1. First, *apktool* extracts the bytecode content of an app and, leveraging another tool, *baksmali*, translates the bytecode to *smali*, a human readable format akin to assembly languages with instruction mnemonics, inlined constants and various annotations. We feed this output to our parser which builds lists of classes, methods, instructions, etc., and a tree representing the type hierarchy in the app. Our constraint generator traverses the lists and emits Prolog rules for method resolution, exception handlers, entry points, etc., as well as rules for each instance of each Dalvik instruction in the program. The Prolog program can then be queried for any information that the analysis specifies. This can for example happen interactively or as part of a more specific, programmed analysis. As an example of the latter, we generate call graphs with a special query and further process the output to visualise the call graph of an app.

The source code for the prototype is available at: `https://bitbucket.org/erw/dalvik-bytecode-analysis-tool`.

6.1. Examples

Here we demonstrate the conversion of flow logic judgements to Prolog source code. For example, the judgement for the `const` instruction is:

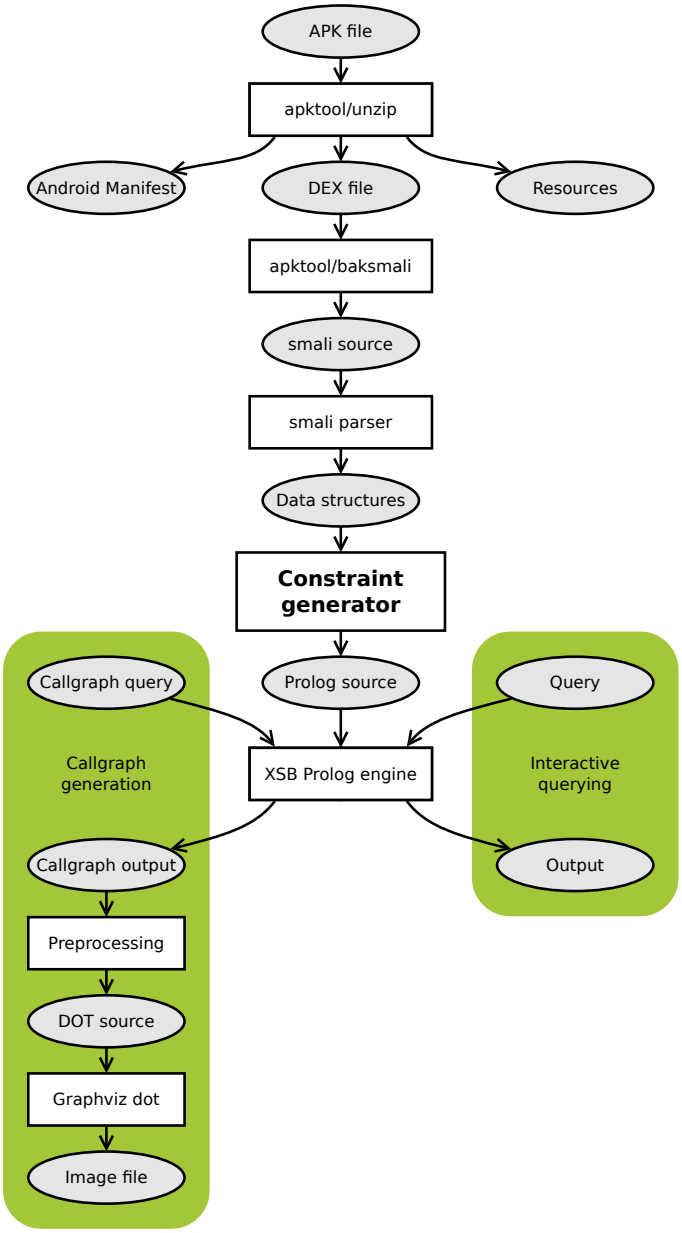

Figure 1: Diagram of the prototype of the analysis tool. Rectangles represent data processors and ellipses represent data.

$$(\hat{S}, \hat{H}, \hat{R}, \hat{E}) \models (m, pc)\colon \texttt{const}\ \ v\ \ c$$
$$\text{iff}\quad \beta(c) \sqsubseteq \hat{R}(m, pc+1)(v)$$
$$\hat{R}(m, pc) \sqsubseteq_{\{v\}} \hat{R}(m, pc+1)$$

For a `const` instruction located at PC 48 in method `m1` in some app, these two Prolog clauses will be generated:

```
% 48: const v5, 0x1
hatR(m1, 49, 5, 0x1).
hatR(m1, 49, V, Y) :-
        hatR(m1, 48, V, Y),
        V \= 5.
```

As can be seen, this conversion and instantiation is fairly straightforward.

An example of a slightly more advanced instruction is `iput` which sets an instance field to a given value on a given object, provided the object's class matches the one that is part of the (fully qualified) field name. Its flow logic judgement is:

$$(\hat{S}, \hat{H}, \hat{R}, \hat{E}) \models (m, pc)\colon \texttt{iput}\ \ v_1\ \ v_2\ \ \mathit{fld}$$
$$\text{iff}\quad \forall(\mathsf{ObjRef}\ (cl, m', pc')) \in \hat{R}(m, pc)(v_2)\colon$$
$$cl \preceq \mathit{fld}.\mathit{class} \Rightarrow$$
$$\hat{R}(m, pc)(v_1) \sqsubseteq \hat{H}(\mathsf{ObjRef}\ (cl, m', pc'))(\mathit{fld})$$
$$\hat{R}(m, pc) \sqsubseteq \hat{R}(m, pc+1)$$

As an instantiation of the Prolog code for `iput`, we show one from the method `Lru/watabou/moon3d/MoonView;-><init>(Landroid/content/Context;)V` which we here abbrivate to `m2`:

```
% 3: iput v1, p0, Lru/watabou/moon3d/MoonView;->targetSunAngle:F
hatH((CL, CLAddrMID, CLAddrPC), 'targetSunAngle', Y) :-
        hatR(m2, 3, 2, (CL, CLAddrMID, CLAddrPC)),
        subclass(CL, 'Lru/watabou/moon3d/MoonView;'),
        hatR(m2, 3, 1, Y).
hatR(m2, 4, V, Y) :-
        hatR(m2, 3, V, Y).
```

The references from register `p0`[22] at the current program counter are extracted. The subclass condition check is implicit: if the `subclass/2` goal fails, the `hatH/3` relation does not hold for those particular arguments and another reference from register `p0` can be tried. If it succeeds, the variable `Y` is bound to each value from the source register in turn. Also, all registers are transferred to the next program point.

6.2. Modelling Java and Android

This section discusses features and program components that are not implemented in the apps and must therefore be modelled separately.

[22]Registers numbered by parameter, such as `p0`, are aliases for other registers. In this method `p0` corresponds to `v2` (because *numLocals* is 2). We perform this translation prior to generating the Prolog code.

6.2.1. API Methods

Methods that are called but not implemented in an app may be from Java standard classes, Android APIs, and from other apps signed by the same developer key. They can also come from classes loaded at runtime but apps that do this are not amenable to static analysis before installation in the first place as discussed in Section 2.

Without handling external methods in some way, it is not possible to resolve the implemented methods. The potential effects of a single method are far-reaching. Using reflection, any method, including private ones, can be called, and anything reachable on the heap from the references given to the unknown method can be changed, again including private and final fields. It is even possible that debugging or diagnostics APIs allow programmatic access to the full heap.

In a specialised analysis it would be useful to be able to trust Java and Android API methods not to do anything malicious, for example by just setting their return values to top, but due to the possibilities of affecting the heap, every API method would require some inspection to determine its effects on the heap. APIs can also be handled as we have done with parts of the reflection API by modelling the methods with individual flow judgements. Alternatively they could be compiled to Dalvik bytecode and analysed along with the app. This last approach might impact the running time of the analysis by increasing the effective size of the app considerably and would only work for the parts of the Java standard classes that are implemented in Java.

6.2.2. Java Features

One Java feature we needed to represent is the `java/lang/Class` instances that represent primitive types. They are stored as static fields named `TYPE` on the corresponding box classes, so an `sget` instruction is produced by the Dalvik compiler instead of `const-class`. For example, the `java/lang/Class` instance representing the `int` type is stored as `Integer.TYPE`. We model this with two Prolog facts like the following for each of the eight primitive Java types plus `void`:

```
1 hatS('Ljava/lang/Integer;', 'TYPE:Ljava/lang/Class;', ('Ljava/lang/Class;', java, 1)).
2 hatH(('Ljava/lang/Class;', java, 1), 'name', 'int').
```

Here, (`java`, 1) is introduced as the creation point of the `java/lang/Class` instance. The program counter values from 1 to 9 are used for the 9 objects.

6.2.3. Reflection

In order to make analysis of apps with reflection tractable, we rely on the assumptions about the use of reflection made in Section 5.3. Briefly, they are: all classes used through reflection are known statically, the program only uses the standard class loader, and strings used to obtain object representations of methods or classes can be determined statically.

Essentially these assumptions ensure that all uses of reflection can be resolved statically. Apps that break the first and the last assumption can not easily be analysed, if at all. Apps that use a non-standard class loader may still be analysed, depending on the specifics of the class loader used. Intuitively, if the used class loader implements a restricted subset of the standard class loader

it can be handled by the current implementation. For other class loaders, that do not work like the standard class loader, a formalisation and/or representation of that class loader is needed to enable analysis with our prototype.

6.2.4. Entry Points

Android apps are simply collections of classes with methods that can be called by the Android system. As an immediate consequence, apps do not have a single uniquely determined entry point. Furthermore, parameters can be passed to the entry points, both from the libraries and the Android system but also from within the app itself. In order for an analysis to soundly capture all of this information, not only the app but the entire system would have to be formalised and analysed. Although an interesting challenge, such comprehensive modelling is out of scope for our analysis and prototype. Instead we opt for an approximate solution in which we simulate calls to potential event handler methods.

We have identified 1,695 event handler methods in the Android API such as `onCreate()` for activities, `onLocationChanged()` for location services, or `onPictureTaken()`. Some are declared on interfaces, others on classes. We generate Prolog facts to simulate calls to the methods on all classes in the app that implement one of the relevant API interfaces and on all subclasses of the relevant API classes. We also call the constructors on subclasses of the four main app components that are instantiated by Android: activities, services, broadcast receivers, and content providers. The remaining many minor app components are listeners that are instantiated by the app itself before they are registered such that Android can call them.

All of these entry point methods are instance methods and as a simple over-approximation, we invoke the methods on all object references of the class, the method is implemented in, using the Prolog anonymous variable to ignore the creation point. The arguments passed to the entry point methods are $\top_{\widehat{\mathsf{Prim}}}$ for primitive arguments. For arguments of reference type, objects with the artificial creation point (`android`, 0) are passed to show that the exact creation point of the argument is unknown and could be in the Android system itself.

Class constructors (see Section 3.1) also form entry points but they do not need to be called explicitly as they do not depend on arguments. All instructions generate constraints whether the method they reside in is called or not.

7. Evaluating the Prototype

Here we perform an experiment to examine the practical aspects of our formal analysis as implemented in the prototype. Therefore we only take the formalised features into account, hence APIs and libraries are not accounted for. Reflection is handled in the limited ways described in Section 5. While the exception analysis has been formalised it is excluded here as well. Yet, while unsound, our prototype has a stronger theoretical foundation than any tool we have studied so far.

We have tested the prototype with many forms of interactive querying and with call graph extraction on several apps from our data set, and in practice memory usage is a bigger issue than running time. As an evaluation of the performance of the prototype we show in Figure 2 memory usage during call graph extraction, i.e., the computation of the domain $\mathsf{MethodCall} = \mathsf{Method} \to$

$\mathcal{P}(\mathsf{Method})$. Here, the formalised analysis ($\widehat{\mathsf{CFA}}$ as defined in Section 4.1, sans the exception component) is used to find the object references that govern dynamic dispatch as well as values that affect reflective calls.

The app size used in Figure 2 is measured in the number of methods, and the evaluation is based on apps with up to 2000 methods in the three market categories *News and Magazines*, *Photography*, and *Productivity*.

A memory ceiling of 5 GiB was set and experiments that hit the ceiling and were aborted are indicated. The data shows the expansion by doubling of the evaluation stacks in the XSB Prolog engine with groupings of values around 1, 2, and 4 GiB. The values lying between these groups result from the two main stacks being expanded independently. Analysis of approximately half of the apps with more than 1000 methods exceeds the memory limit.

The CPU time of the experiments that completed within the memory limit ranged from 0.05 to 1227 s with 89% of the runs being below 200 s on an Intel Core i7 processor. The time used for extraction and disassembly of the bytecode from the APK file is negligible and not included. The conversion from smali source to Prolog is included and typically takes less than 10 s for the largest apps in this experiment. Most of the running time is spent compiling the generated Prolog source and evaluating the call graph query, with time distributed almost equally between these two tasks. Consequently, subsequent queries will be answered much faster than the first one for an app.

Turning the above experiment into an estimate of the percentage of real-world apps the prototype can handle, Figure 3 shows the distribution of app sizes in our data set. Apps with 1000 methods or less comprise 42% of the data set, indicating that the prototype in its current state can analyse this part of the apps with 5 GiB of memory at its disposal. Figure 2 reveals an exponential tendency, suggesting both that almost the same number of apps can be handled with only 2.5 GiB memory, for example, but also that exceptionally large apps with 20000–50000 methods will use disproportionate amounts of memory. Indeed our tests reveal that some apps require too much memory to analyse, even on a server with 68 GB memory available to us.

Since we have implemented and improved the analysis simultaneously, readability and debuggability of the Prolog code has been important to us. The prototype is strictly proof-of-concept but we expect that it would be possible to improve the efficiency of the analysis considerably.

The large majority of clauses are concerned with copying register values for every step of a method. An optimisation would concern copying values directly to the program points where they are read, thus bringing down the number of clauses significantly. Many Dalvik instructions produce runtime exceptions and their clauses often make up one third of the generated Prolog program. This is another opportunity for more efficient handling. In the above experiments runtime exceptions were not considered.

Currently the solver runs only a single thread so with respect to running time there are big gains to be had by parallelising the computation appropriately, as has been done for the analysis of timed automata in [6, 16]. In [21] it is shown how the framework can be extended to general program analysis, such as the analysis specified in this paper. The parallel implementation of the framework is shown to scale almost linearly, up to a factor of 40 on a 48-core machine [6].

Another implementation approach, similar in spirit to our current approach,

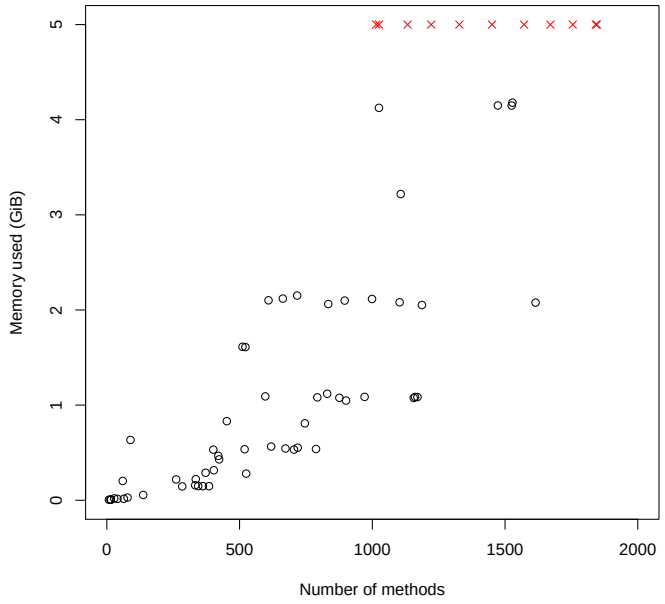

Figure 2: Memory usage evaluation. Crosses represent experiments running out of memory.

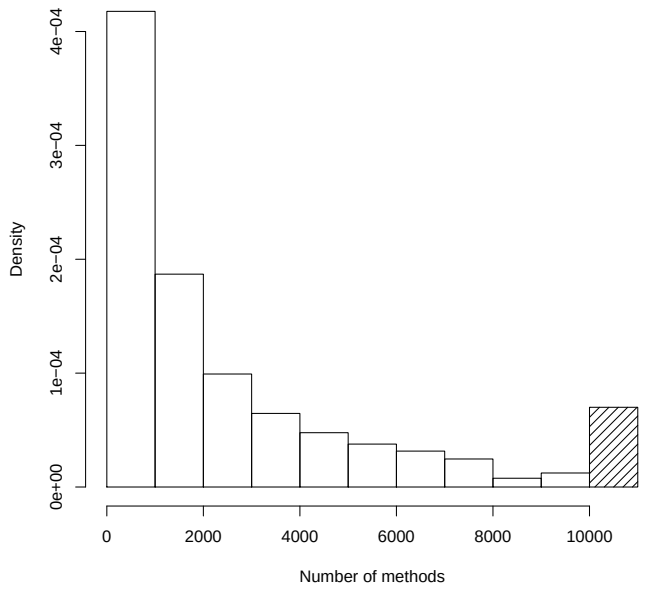

Figure 3: Histogram of number of methods per app. The last bin contains all apps with more than 10000 methods.

would be to use a Datalog solver. Prolog allowed us to rapidly design and implement a proof-of-concept prototype to validate the analysis as well as the implementation strategy of generating constraints. Using Datalog is a suitable avenue for future work.

8. Looking for Malware

As an example of an approach to malware detection, we have examined apps that send text messages to see which phone numbers are used as destinations. In many cases, it was not possible to learn the numbers because they come from API methods that we do not yet support. The typical pattern for legitimate apps that send text messages is to retrieve numbers from the database with the user's contact list. This requires a large number of API calls, some to connect to the database, some to read the content, and some to store and retrieve the results from Java collections. However, to test tracking values across collection APIs, we created special cases of the `invoke-virtual` instruction for calls to methods on the `java/util/ArrayList` standard class (including `add()`, `get()`, and `toArray()`), as well as special cases of `invoke-direct` for two of its constructors. Similar to regular Java arrays, we treated ArrayLists as unordered collections. Combined with tracking of the return value of the `android/database/Cursor;->getString()` method which is used, among other things, to retrieve phone numbers of contacts, we were able to verify for an app that only contact phone numbers were used as the destination argument of the `android/telephony/SmsManager;->sendTextMessage()` method.

A typical pattern for malware is to send messages to hardcoded numbers. We found an app in our data set for which the analysis determined that the string "`1277`" is the only possible value given as the destination for text messages. The number is a Danish premium number but the description of the app on Google Play did state that the app sends a premium message for each look-up, so the app cannot be classified as malicious even though it seems to make money on inattentive users.

Some apps that use hardcoded numbers (whether malicious or not) are specialised and store different numbers for different countries in XML files. These require a considerable number of API calls to parse and extract, so in general finding this type of malware requires implementing more API methods. We also note that the above result in which only phone numbers of contacts are used is not enough, on its own, to classify an app as benign. A malicious app could for example covertly create new contact entries with expensive numbers beforehand. However, some malware authors are looking for financial gain from a minimal coding effort. In the next section we present such a case.

8.1. The FakePlayer Trojan

We also analysed a known malicious app posing as a movie player [3], named *Android/FakePlayer.A* and *AndroidOS.FakePlayer* by anti-virus vendors. It is a very small app with 15 methods in nine classes. Five of the classes are standard Android "R" resource classes with a constructor each. Running the analysis takes 0.3 s and less than 10 MB of memory. It identifies 45 method calls comprising 27 unresolved API calls, six calls to the `java/lang/Object` constructor API method, six calls within the app, hereof four unique, and,

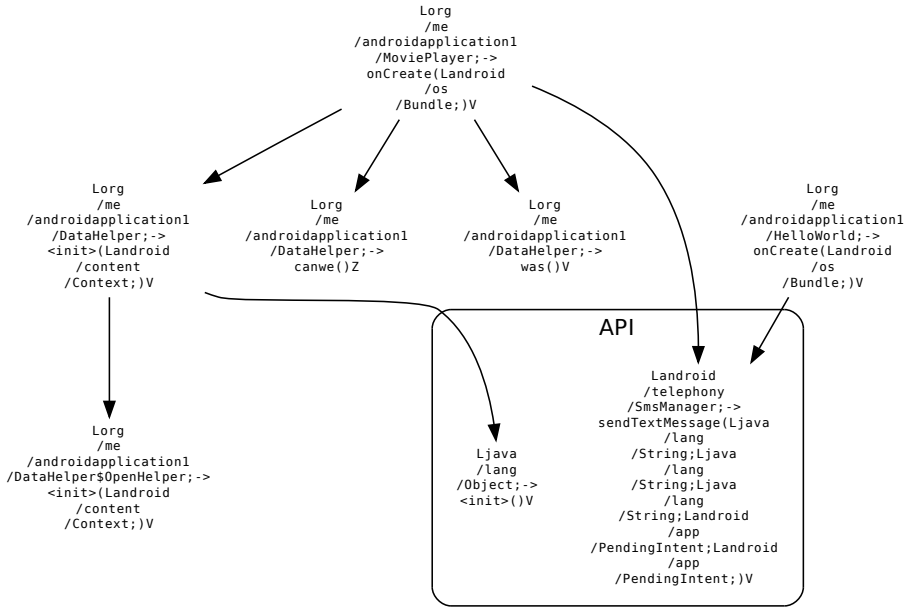

Figure 4: A small part of the FakePlayer callgraph.

finally, six calls to the `sendTextMessage()` API method, hereof two unique. Figure 4 shows a portion of the call graph. The unresolved part includes calls to the Android `android/widget/TextView` API to display a message prompting the user to wait for access to the video library. There is also a number of calls to the `android/database` API that by manual inspection appear pointless. No API to actually play videos to the user are accessed.

Independently of the unresolved calls our analysis shows the hardcoded phone numbers by querying the destination argument of `sendTextMessage()`:

```
invoke('Landroid/telephony/SmsManager;->sendTextMessage(Ljava/lang/String;
    Ljava/lang/String;Ljava/lang/String;Landroid/app/PendingIntent;
    Landroid/app/PendingIntent;)V', 1, ('Ljava/lang/String;', M, PC)),
    hatH(('Ljava/lang/String;', M, PC), 'value', Y).
```

This query yields the Russian phone numbers 3353 and 3354 for which each text message may cost around €4–8[23].

9. Conclusion

In this paper we have discussed the results from studying 1,700 popular Android apps available from the Google Play app market. The apps were studied to determine which instructions and language features are most commonly used in "real-life" Android apps. In addition to guiding the design of the semantics and the control flow analysis, we believe that our study is of independent interest. The insights gained into the design and implementation of Android apps

[23]See `http://sms-price.ru/number/3353/` and `http://sms-price.ru/number/3354/`. Last accessed 5 June 2012.

are very useful, necessary even, for developing new analyses, designing heuristics to cover advanced language system features, and for prioritising which among the many language features, libraries, and components in an Android system to focus on in future work.

The main insights revealed by our study was: (1) that almost all instructions in the Dalvik instruction set are used in most apps; (2) use of advanced language/system features, e.g., reflection, dynamic class loading etc. is widespread.

Based on the app study, we have developed a formal operational semantics for the core Dalvik bytecode language, incorporating all major core language features, including dynamic dispatch, exceptions, and reflection. Concurrency is the most notable omission and is the topic of future work. Of the non-core language features that have not been included in the formalisation, libraries are the most important. Without a good model of libraries and their APIs, it is almost impossible to perform analysis of real-life apps. The formal semantics is, of course, a prerequisite for formally developing a sound program analysis and the basis for a proof of correctness. However, a formal semantics is also useful as succinct yet detailed and unambiguous documentation of a language and, in particular, of advanced and subtle language features. In our semantics, this is especially evident in the rules dealing with reflection. We believe that our semantics can serve as a useful tread stone for the Dalvik community, both for developing and discussing (new) language features, but also for better tool support able to handle also corner cases.

In this paper, the formal semantics was used for designing and formally specifying, in a very direct way, a sound control flow analysis. As was the case for the semantics, we believe that the formal specification of a fundamental analysis, such as the control flow analysis, is useful for a wider community for understanding the issues involved in analysing Dalvik bytecode. In addition, many advanced analyses, e.g., secure information flow analysis, use control flow analysis as a building block. Providing this building block, backed by a formal semantics, significantly simplifies the development of new, advanced analyses. During the development of the flow logic specification, the main difficulties turned out to be, again, the reflection features as well as subtle interactions between apps and the Android system.

Finally, a prototype implementation of the analysis was developed by systematically, and almost mechanically, turning the flow logic specification for a program under analysis into a corresponding Prolog program. By executing and querying the Prolog program the analysis result can be established. The prototype was mainly developed as a testbed for variations of and extensions to the control flow analysis. Furthermore, even though high performance was not a goal of the prototype, working with the prototype helped identify particularly costly parts of the analysis, e.g., that taking all the built-in exceptions into account in the exception analysis is prohibitively expensive.

Future work. The work described in this paper leaves many avenues open for future work. However, we have identified the following four areas as most promising: (1) extending the semantics and analysis to handle concurrency; (2) extending the analysis of reflection, e.g., by adding string analysis; (3) formal modelling/specification of the Android system and libraries, e.g., by making formal (abstract) models/specifications for the APIs and use these for analysis; and (4) development of a more comprehensive and efficient implementation, e.g.,

using Datalog.

Acknowledgements. We would like to thank the anonymous reviewers for catching subtle (and not so subtle) errors and for providing detailed and constructive feedback.

References

[1] Bytecode for the Dalvik VM. Available from the Android Project website: `http://source.android.com/tech/dalvik/dalvik-bytecode.html`, last accessed 14 December 2011

[2] Bertelsen, P.: Semantics of Java byte code. Student project report, Technical University of Denmark (1997)

[3] Blasco, J.: Analysis of Trojan-SMS.AndroidOS.FakePlayer.a. Alienvault Labs website: `http://labs.alienvault.com/labs/index.php/2010/analysis-of-trojan-sms-androidos-fakeplayer-a/` (Aug 2010), last accessed 20 November 2011

[4] Chin, E., Felt, A., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (2011)

[5] Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Proceedings 10th International Static Analysis Symposium (SAS). LNCS, vol. 2694, pp. 1–18. Springer-Verlag (June 2003), available from `http://www.brics.dk/JSA/`

[6] Dalsgaard, A.E., Laarman, A., Larsen, K.G., Olesen, M.C., Pol, J.v.d.: Multi-core reachability for timed automata. In: Proceedings of Formal Modeling and Analysis of Timed Systems (FORMATS 2012). Lecture Notes in Computer Science, vol. 7595, pp. 91–106. Springer (2012)

[7] Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of Android application security. In: Proceedings of the 20th USENIX Security Symposium (SEC'11). pp. 315–330. USENIX Association, San Francisco, CA, USA (Aug 2011)

[8] 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 (CCS 2011). pp. 627–638 (2011)

[9] Freund, S.N., Mitchell, J.C.: A type system for object initialization in the Java bytecode language. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98). pp. 310–328. ACM Press, Vancouver, British Columbia, Canada (1998)

[10] Freund, S.N., Mitchell, J.C.: A formal framework for the Java bytecode language and verifier. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99). pp. 147–166. ACM Press, Denver, CO, USA (Nov 1999)

[11] Hansen, R.R.: Flow Logic for Language-Based Safety and Security. Ph.D. thesis, Technical University of Denmark (2005)

[12] Jeon, J., Micinski, K.K., Foster, J.S.: SymDroid: Symbolic execution for Dalvik bytecode (Jul 2012), http://www.cs.umd.edu/~jfoster/papers/symdroid.pdf

[13] Karlsen, H.S., Wognsen, E.R.: Static Analysis of Dalvik Bytecode and Reflection in Android. Master's thesis, Aalborg University (Jun 2012), available from http://projekter.aau.dk/projekter/en/studentthesis/static-analysis-of-dalvik-bytecode-and-reflection-in-android%284dd9e717-c5d2-4603-a2d7-0f043fe9ea1f%29.html

[14] Karlsen, H.S., Wognsen, E.R., Olesen, M.C., Hansen, R.R.: Study, formalisation, and analysis of Dalvik bytecode. In: Informal proceedings of The Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2012) (2012), informal proceedings

[15] Kieżun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: A solver for string constraints. In: Proceedings of the 2009 International Symposium on Software Testing and Analysis (ISSTA 2009). Chicago, IL, USA (Jul 2009)

[16] Laarman, A., Olesen, M.C., Dalsgaard, A., Larsen, K.G., van de Pol, J.: Multi-core emptiness checking of timed Buchi automata using inclusion abstraction. In: Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013). pp. 968–983. Lecture Notes in Computer Science, Springer, Saint Petersburg, Russia (2013)

[17] Livshits, B., Whaley, J., Lam, M.S.: Reflection analysis for Java. Tech. rep., Stanford University (Oct 2005)

[18] Luo, T., Hao, H., Du, W., Wang, Y., Yin, H.: Attacks on WebView in the Android system. In: Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC 2011). pp. 343–352 (2011)

[19] Manson, J., Goetz, B.: JSR 133 (Java memory model) FAQ. Webpage available at http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html (Feb 2004)

[20] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Verlag (1999)

[21] Olesen, M.C.: Program Analysis as Model Checking. Ph.D. thesis, Aalborg University (2013), to appear

[22] Oracle Corporation: Java Platform, Standard Edition 6: API Specification, available from http://docs.oracle.com/javase/6/docs/api/overview-summary.html. Last accessed 23 May 2012

[23] Oracle Corporation: JSR 133: Java Memory Model and Thread Specification Revision, available from http://jcp.org/en/jsr/detail?id=133, last accessed 29 May 2012

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

[25] Plotkin, G.D.: A structural approach to operational semantics. DAIMI FN-19, Computer Science Department (DAIMI), Aarhus University (Sep 1981)

[26] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P 2010). pp. 513–528 (May 2010)

[27] Siveroni, I.: Operational semantics of the Java Card virtual machine. Journal of Logic and Algebraic Programming 58(1–2), 3–25 (Jan/Mar 2004)

[28] Spoto, F.: Julia: A generic static analyser for the Java bytecode. In: Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP'2005) (2005)

[29] Spoto, F., Jensen, T.P.: Class analyses as abstract interpretations of trace semantics. ACM Trans. Program. Lang. Syst. 25(5), 578–630 (2003)

[30] Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot — a Java optimization framework. In: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1999). pp. 125–135 (1999)

[31] Vitek, J., Horspool, R.N., Uhl, J.S.: Compile-time analysis of object-oriented programs. In: Proceedings International Conference on Compiler Construction (CC'92). Lecture Notes in Computer Science, vol. 641. Springer Verlag (1992)

Appendix A. Generalised Instruction Set

New instruction	Opcode	Corresponding original instructions
nop	00	nop
move	01..09	move, move/from16, move/16, move-wide, ...
move-result	0a..0c	move-result, move-result-wide, move-result-object
move-exception	0d	move-exception
return-void	0e	return-void
return	0f..11	return, return-wide, return-object
const	12..19	const/4, const/16, const, const/high16, ...
const-string	1a..1b	const-string, const-string/jumbo
const-class	1c	const-class
monitor-enter	1d	monitor-enter
monitor-exit	1e	monitor-exit
check-cast	1f	check-cast
instance-of	20	instance-of
array-length	21	array-length
new-instance	22	new-instance
new-array	23	new-array
filled-new-array	24..25	filled-new-array, filled-new-array/range
fill-array-data	26	fill-array-data
throw	27	throw

New instruction	Opcode	Corresponding original instructions
goto	28..2a	goto, goto/16, goto/32
packed-switch	2b	packed-switch
sparse-switch	2c	sparse-switch
cmp	2d..31	cmpl-float, cmpg-float, cmpl-double, ...
if	32..37	if-eq, if-ne, if-lt, if-ge, if-gt, if-le
ifz	38..3d	if-eqz, if-nez, if-ltz, if-gez, if-gtz, if-lez
aget	44..4a	aget, aget-wide, aget-object, aget-boolean, ...
aput	4b..51	aput, aput-wide, aput-object, aput-boolean, ...
iget	52..58	iget, iget-wide, iget-object, iget-boolean, ...
iput	59..5f	iput, iput-wide, iput-object, iput-boolean, ...
sget	60..66	sget, sget-wide, sget-object, sget-boolean, ...
sput	67..6d	sput, sput-wide, sput-object, sput-boolean, ...
invoke-virtual	6e, 74	invoke-virtual, invoke-virtual/range
invoke-super	6f, 75	invoke-super, invoke-super/range
invoke-direct	70, 76	invoke-direct, invoke-direct/range
invoke-static	71, 77	invoke-static, invoke-static/range
invoke-interface	72, 78	invoke-interface, invoke-interface/range
unop	7b..8f	neg-int, not-int, neg-long, not-long, ...
binop	90..cf	add-int, sub-int, ..., add-int/2addr, ...
binop-lit	d0..e2	add-int/lit16, rsub-int, mul-int/lit16, ...

Appendix B. Occurrences of Instructions

The generalized instructions in our data set ordered by the percentage of the 1,700 apps in our data set that use the instruction.

Instruction	Occurs in	# of occurrences	Part of total occ.
`invoke-direct`	100.00 %	4,533,934	4.80 %
`return-void`	100.00 %	2,683,104	2.84 %
`invoke-virtual`	99.59 %	12,718,970	13.47 %
`const`	99.53 %	8,157,468	8.64 %
`move-result`	99.47 %	12,391,920	13.13 %
`invoke-super`	99.47 %	215,434	0.23 %
`const-string`	99.29 %	5,200,603	5.51 %
`new-instance`	99.29 %	2,900,269	3.07 %
`invoke-static`	99.24 %	3,833,347	4.06 %
`iput`	99.12 %	3,389,122	3.59 %
`iget`	99.06 %	8,062,226	8.54 %
`ifz`	99.06 %	3,984,192	4.22 %
`goto`	98.76 %	3,263,902	3.46 %
`return`	98.06 %	2,166,727	2.29 %
`move-exception`	97.71 %	761,554	0.81 %
`check-cast`	97.53 %	1,055,790	1.12 %
`if`	97.24 %	1,304,228	1.38 %
`binop-lit`	96.59 %	1,232,732	1.31 %
`invoke-interface`	96.35 %	1,761,883	1.87 %
`move`	96.24 %	5,503,780	5.83 %
`new-array`	95.47 %	557,610	0.59 %
`sget`	95.18 %	1,792,583	1.90 %
`aput`	94.88 %	1,864,219	1.97 %
`binop`	94.53 %	1,218,279	1.29 %
`aget`	94.47 %	734,425	0.78 %
`unop`	94.00 %	530,779	0.56 %
`sput`	93.88 %	607,269	0.64 %
`array-length`	93.65 %	263,662	0.28 %
`const-class`	93.53 %	182,077	0.19 %
`throw`	93.47 %	521,299	0.55 %
`packed-switch`	93.35 %	86,468	0.09 %
`nop`	92.76 %	56,951	0.06 %
`cmp`	92.00 %	189,789	0.20 %
`monitor-exit`	88.76 %	287,310	0.30 %
`monitor-enter`	88.76 %	134,466	0.14 %
`fill-array-data`	86.71 %	97,906	0.10 %
`instance-of`	85.76 %	144,576	0.15 %
`sparse-switch`	69.71 %	21,149	0.02 %
`filled-new-array`	22.29 %	1,930	0.00 %
Total		94,413,932	100.00 %

Appendix C. Structural Domains

The semantic domains that were left out of the text are listed here. The domains included in App could be specified further for an extension of the semantics into the Android API.

$$\begin{aligned}\mathsf{App} = {} & (\mathit{name}\colon \mathsf{AppName}) \times (\mathit{classes}\colon \mathcal{P}(\mathsf{Class}))\times \\ & (\mathit{interfaces}\colon \mathcal{P}(\mathsf{Interface})) \times (\mathit{manifest}\colon \mathsf{Manifest})\times \\ & (\mathit{certificate}\colon \mathsf{Certificate}) \times (\mathit{resources}\colon \mathcal{P}(\mathsf{Resource}))\times \\ & (\mathit{assets}\colon \mathcal{P}(\mathsf{Asset})) \times (\mathit{libs}\colon \mathcal{P}(\mathsf{Lib}))\end{aligned}$$

$$\mathsf{Package} = (\mathit{name}\colon \mathsf{PackageName}) \times (\mathit{app}\colon \mathsf{App}) \times (\mathit{classes}\colon \mathcal{P}(\mathsf{Class}))$$

$$\mathsf{DataTable} = \mathsf{ArrayData} \cup \mathsf{SparseSwitch} \cup \mathsf{PackedSwitch}$$

$$\mathsf{ArrayData} = (\mathit{size}\colon \mathbb{N}_0) \times (\mathit{data}\colon \mathbb{N}_0 \to \mathsf{Prim})$$

$$\mathsf{SparseSwitch} = (\mathit{sparseTargets}\colon \mathbb{N}_0 \to \mathsf{PC})$$

$$\mathsf{PackedSwitch} = (\mathit{firstKey}\colon \mathbb{N}_0) \times (\mathit{size}\colon \mathbb{N}_0) \times (\mathit{packedTargets}\colon \mathbb{N}_0 \to \mathsf{PC})$$

The type domains are specified using BNF notation:

$$\begin{array}{rcl}
\mathsf{Type} & ::= & \mathsf{RefType} \mid \mathsf{PrimType} \\
\mathsf{PrimType} & ::= & \mathsf{PrimSingle} \mid \mathsf{PrimDouble} \\
\mathsf{PrimSingle} & ::= & \texttt{boolean} \mid \texttt{char} \mid \texttt{byte} \mid \texttt{short} \mid \texttt{int} \mid \texttt{float} \\
\mathsf{PrimDouble} & ::= & \texttt{long} \mid \texttt{double} \\
\mathsf{RefType} & ::= & \mathsf{Class} \mid \mathsf{ArrayType} \\
\mathsf{ArrayType} & ::= & \mathsf{ArrayTypeSingle} \mid \mathsf{ArrayTypeDouble} \\
\mathsf{ArrayTypeSingle} & ::= & \texttt{array}\ (\mathsf{RefType} \mid \mathsf{PrimSingle}) \\
\mathsf{ArrayTypeDouble} & ::= & \texttt{array}\ \mathsf{PrimDouble}
\end{array}$$

Class hierarchy functions and the subtyping relation:

$$\begin{aligned}
\mathit{super}^*(\bot) &= \emptyset \\
\mathit{super}^*(\mathit{cl}) &= \{\mathit{cl}.\mathit{super}\} \cup (\mathit{cl}.\mathit{super}).\mathit{super}^* \\
\mathit{implements}^*(\bot) &= \emptyset \\
\mathit{implements}^*(\mathit{cl}) &= \mathit{cl}.\mathit{implements} \cup (\mathit{cl}.\mathit{super}).\mathit{implements}^* \cup (\mathit{cl}.\mathit{implements}).\mathit{super}^* \\
\mathit{super}^*(\mathit{ifaces}) &= \bigcup_{\mathit{iface} \in \mathit{ifaces}} \mathit{iface}.\mathit{super} \cup (\mathit{iface}.\mathit{super}).\mathit{super}^*
\end{aligned}$$

$$\frac{\mathit{cl} \in \mathsf{Class}}{\mathit{cl} \preceq \mathit{cl}} \qquad \frac{\mathit{cl}' \in \mathit{super}^*(\mathit{cl})}{\mathit{cl} \preceq \mathit{cl}'} \qquad \frac{\mathit{iface} \in \mathit{implements}^*(\mathit{cl})}{\mathit{cl} \preceq \mathit{iface}} \qquad \frac{t \preceq t'}{(\texttt{array}\ t) \preceq (\texttt{array}\ t')}$$

Appendix D. Subject Reduction Proof

We here present four cases of the subject reduction proof introduced in Section 4.3. The cases demonstrate updates to a register value and the heap as well as the central `invoke-virtual` instruction and object sharing in the `const-class` instruction.

Appendix D.1. The `move` *Case*

By definition:

$$\frac{\mathit{m}.\mathit{instructionAt}(\mathit{pc}) = \texttt{move}\ v_1\ v_2}{A \vdash \langle S, H, \langle m, \mathit{pc}, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, \mathit{pc}+1, R[v_1 \mapsto R(v_2)]\rangle :: SF\rangle}$$

For the proof we assume an analysis result satisfying the judgement for the instruction parameterized by its program point as well as an instantiation of the theorem assumption with an expansion of the domains to facilitate referencing:

$$(\hat{S}, \hat{H}, \hat{R}) \models (m, \mathit{pc})\colon \texttt{move}\ v_1\ v_2 \tag{D.1}$$

$$\langle S, H, \langle m, \mathit{pc}, R\rangle :: SF\rangle\ \mathcal{R}_{\mathsf{Conf}}\ (\hat{S}, \hat{H}, \hat{R}) \tag{D.2}$$

From (D.1) and the flow logic judgement (see Section 4.2) we get

$$\hat{R}(m, \mathit{pc})(v_2) \sqsubseteq \hat{R}(m, \mathit{pc}+1)(v_1) \tag{D.3}$$

$$\hat{R}(m, \mathit{pc}) \sqsubseteq_{\{v_1\}} \hat{R}(m, \mathit{pc}+1) \tag{D.4}$$

and from (D.2) it follows that

$$\beta^H_{\mathsf{StaticHeap}}(S) \sqsubseteq \hat{S} \tag{D.5}$$

$$\beta_{\mathsf{Heap}}(H) \sqsubseteq \hat{H} \tag{D.6}$$

$$\beta^H_{\mathsf{LocalReg}}(R) \sqsubseteq \hat{R}(m, pc) \tag{D.7}$$

From (D.7) and the definition of $\sqsubseteq$ (see Section 4.1) we have

$$\beta^H_{\mathsf{Val}}(R(v_2)) \sqsubseteq \hat{R}(m, pc)(v_2) \tag{D.8}$$

While (D.8) holds for any v_2 we are interested in the specific v_2 given as an argument to the instruction. Combining (D.8) and (D.3),

$$\beta^H_{\mathsf{Val}}(R(v_2)) \sqsubseteq \hat{R}(m, pc)(v_2) \sqsubseteq \hat{R}(m, pc+1)(v_1) \tag{D.9}$$

From (D.4) we get

$$\forall r \in \mathrm{dom}(\hat{R}(m, pc)) \setminus \{v_1\} : \hat{R}(m, pc)(r) \sqsubseteq \hat{R}(m, pc+1)(r) \tag{D.10}$$

Now (D.9) covers the case where we consider register v_1 and (D.10) covers all other cases. Therefore:

$$\beta^H_{\mathsf{LocalReg}}(R[v_1 \mapsto R(v_2)]) \sqsubseteq \hat{R}(m, pc+1) \tag{D.11}$$

The case then follows from (D.5), (D.6), and (D.11).

Appendix D.2. The `iput` *Case*

By definition:

$$\frac{\begin{array}{c} m.\mathit{instructionAt}(pc) = \texttt{iput}\ \ v_1\ \ v_2\ \ \mathit{fld} \\ R(v_2) = \mathit{loc} \neq \texttt{null} \qquad o = H(\mathit{loc}) \\ o.\mathit{class} \preceq \mathit{fld}.\mathit{class} \qquad o' = o[\mathit{field} \mapsto o.\mathit{field}[\mathit{fld} \mapsto R(v_1)]] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H[\mathit{loc} \mapsto o'], \langle m, pc+1, R\rangle :: SF\rangle}$$

We assume

$$(\hat{S}, \hat{H}, \hat{R}) \models (m, pc) \colon \texttt{iput}\ \ v_1\ \ v_2\ \ \mathit{fld} \tag{D.12}$$

$$\langle S, H, \langle m, pc, R\rangle :: SF\rangle\ \mathcal{R}_{\mathsf{Conf}}\ (\hat{S}, \hat{H}, \hat{R}) \tag{D.13}$$

From (D.12) we get

$$\hat{R}(m, pc) \sqsubseteq \hat{R}(m, pc+1) \tag{D.14}$$

and

$$\begin{array}{l} \forall(\mathsf{ObjRef}\ \ cl, m', pc') \in \hat{R}(m, pc)(v_2) \colon \\ \quad cl \preceq \mathit{fld}.\mathit{class} \Rightarrow \\ \qquad \hat{R}(m, pc)(v_1) \sqsubseteq \hat{H}(\mathsf{ObjRef}\ \ cl, m', pc')(\mathit{fld}) \end{array} \tag{D.15}$$

From (D.13) follows

$$\beta^H_{\mathsf{StaticHeap}}(S) \sqsubseteq \hat{S} \tag{D.16}$$

$$\beta_{\mathsf{Heap}}(H) \sqsubseteq \hat{H} \tag{D.17}$$

$$\beta^H_{\mathsf{LocalReg}}(R) \sqsubseteq \hat{R}(m, pc) \tag{D.18}$$

Combining (D.18) and (D.14),

$$\beta^H_{\mathsf{LocalReg}}(R) \sqsubseteq \hat{R}(m, pc) \sqsubseteq \hat{R}(m, pc + 1) \tag{D.19}$$

Two out of the three parts of the correctness of the new configuration follow from (D.16) and (D.19).

For the final part, the dynamic heap, we recall from Section 4.3 that the abstract representation of the concrete heap is the union of the representations of its objects who are in turn represented as the representations of their fields. Since we know from (D.17) that the existing representation is part of the analysis result, we can reduce the problem of showing

$$\beta_{\mathsf{Heap}}(H[\mathit{loc} \mapsto o[\mathit{field} \mapsto o.\mathit{field}[\mathit{fld} \mapsto R(v_1)]]]) \sqsubseteq \hat{H} \tag{D.20}$$

to showing

$$\beta^H_{\mathsf{Val}}(R(v_1)) \sqsubseteq \hat{H}(\mathsf{ObjRef}\ \mathit{cl}, m', pc')(\mathit{fld}) \tag{D.21}$$

From (D.18) we have

$$\beta^H_{\mathsf{Val}}(R(v_1)) \sqsubseteq \hat{R}(m, pc)(v_1) \tag{D.22}$$

Because the semantic step has been taken, the premises of the semantic rule must have been satisfied, so from (D.15) we get

$$\hat{R}(m, pc)(v_1) \sqsubseteq \hat{H}(\mathsf{ObjRef}\ \mathit{cl}, m', pc')(\mathit{fld}) \tag{D.23}$$

Combining (D.22) and (D.23) finishes the proof of this case.

Appendix D.3. The `const-class` *Sharing Object Case*

There are two subcases for `const-class`: one that creates a new object and one that re-uses and shares an already existing object. Here we look only at the latter case.

By definition:

$$\frac{\begin{array}{c} m.\mathit{instructionAt}(pc) = \texttt{const-class}\ v\ \ \mathit{cl} \\ \mathit{findClassObject}(H, \mathit{cl}) = \mathit{loc}_c \neq \bot \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc + 1, R[v \mapsto \mathit{loc}_c]\rangle :: SF\rangle}$$

We assume

$$(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\colon \texttt{const-class}\ v\ \ \mathit{cl} \tag{D.24}$$

$$\langle S, H, \langle m, pc, R\rangle :: SF\rangle\ \mathcal{R}_{\mathsf{Conf}}\ (\hat{S}, \hat{H}, \hat{R}) \tag{D.25}$$

From (D.25) it follows that

$$\beta^H_{\mathsf{StaticHeap}}(S) \sqsubseteq \hat{S} \tag{D.26}$$

$$\beta_{\mathsf{Heap}}(H) \sqsubseteq \hat{H} \tag{D.27}$$

$$\beta^H_{\mathsf{LocalReg}}(R) \sqsubseteq \hat{R}(m, pc) \tag{D.28}$$

Further, it follows from (D.24) that

$$\forall v'\colon v \neq v' \ \Rightarrow\ \hat{R}(m, pc + 1)(v') = \hat{R}(m, pc)(v') \tag{D.29}$$

and

$$\mathrm{dom}(\hat{H})|_{\texttt{java/lang/Class}} \sqsubseteq \hat{R}(m, pc+1)(v)$$

Now, since $loc_c = \mathit{findClassObject}(H, cl)$ it follows that $H(loc_c) \in \mathsf{Object}$ with $H(loc_c).\mathit{class} = \texttt{java/lang/Class}$, $H(loc_c).\mathit{class} = cl$, and $H(loc_c).\mathit{origin} = (m', pc')$ for some (m', pc'), it follows from (D.27) that:

$$\beta^H_{\mathsf{Object}}(H(loc_c)) \sqsubseteq \beta_{\mathsf{Heap}}(H)(\mathsf{ObjRef}\ (\texttt{java/lang/Class}, m', pc'))$$

where $\beta^H_{\mathsf{Val}}(loc_c) = (\mathsf{ObjRef}\ (\texttt{java/lang/Class}, m', pc'))$. We can now calcluate as follows:

$$\begin{array}{lcl} (\beta^H_{\mathsf{Val}} \circ R[v \mapsto loc_c])(v) & = & \beta^H_{\mathsf{Val}}(loc_c) \\ & = & \{(\mathsf{ObjRef}\ (\texttt{java/lang/Class}, m', pc'))\} \\ & \subseteq & \mathrm{dom}(\hat{H})|_{\texttt{java/lang/Class}} \\ & \subseteq & \hat{R}(m, pc+1)(v) \end{array} \tag{D.30}$$

Combining (D.29) and (D.30) we have

$$\beta^H_{\mathsf{LocalReg}}(R[v \mapsto loc_c]) \sqsubseteq \hat{R}(m, pc+1) \tag{D.31}$$

The case now follows from (D.26), (D.27), and (D.31).

Appendix D.4. The `invoke-virtual` *Case*

By definition:

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

We assume

$$(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\colon \texttt{invoke-virtual}\ \ v_1 \ldots v_n\ \ \mathit{meth} \tag{D.32}$$

$$\langle S, H, \langle m, pc, R\rangle :: SF\rangle\ \mathcal{R}_{\mathsf{Conf}}\ (\hat{S}, \hat{H}, \hat{R}) \tag{D.33}$$

From (D.32) we get

$$\begin{array}{l} \forall(\mathsf{ObjRef}\ \ cl, m_t, pc_t) \in \hat{R}(m, pc)(v_1)\colon \\ \quad m' = \mathit{resolveMethod}(\mathit{meth}, cl) \\ \quad \forall 1 \leq i \leq n\colon \\ \qquad \hat{R}(m, pc)(v_i) \sqsubseteq \hat{R}(m', 0)(m'.\mathit{numLocals} - 1 + i) \end{array} \tag{D.34}$$

From (D.33) follows

$$\beta^H_{\mathsf{StaticHeap}}(S) \sqsubseteq \hat{S} \tag{D.35}$$

$$\beta_{\mathsf{Heap}}(H) \sqsubseteq \hat{H} \tag{D.36}$$

$$\beta^H_{\mathsf{LocalReg}}(R) \sqsubseteq \hat{R}(m, pc) \tag{D.37}$$

It remains to be shown that the new stack frame, $\langle m', 0, R' \rangle$, is correctly represented in $\hat{R}$. It follows from (D.33) that the rest of the stack, $\langle m, pc, R \rangle :: S$, is correctly represented. Therefore we now need to show

$$\beta^H_{\mathsf{LocalReg}}(R') \sqsubseteq \hat{R}(m', 0) \tag{D.38}$$

and that the m' referenced in the analysis is the same as the one in the semantics. The latter follows because $\hat{R}(m, pc)(v_1)$ is a sound over-approximation of $R(v_1)$ and thus, by the induction hypothesis, must contain an abstract representation of the correct object reference. The same method lookup is used in the semantics and in (D.34) so the correct method must be included in the analysis.

By the definition of $\sqsubseteq$, (D.38) is equivalent to

$$\forall r \in \mathrm{dom}(R') : \beta^H_{\mathsf{Val}}(R'(r)) \sqsubseteq \hat{R}(m', 0)(r) \tag{D.39}$$

Since R' is used both to transfer parameter values to the called method as well as for storing (the value of) local variables in the called method, we split this into two cases for $r \in \mathrm{dom}(R')$:

1. $0 \leq r \leq m'.numLocals - 1$
2. $m'.numLocals \leq r \leq m'.numLocals + n - 1$

From the semantics it follows that $R'(r) = \bot_{\mathsf{Val}}$ for all r such that $0 \leq r \leq m'.numLocals - 1$ and thus

$$\begin{array}{rcl} \beta^H_{\mathsf{LocalReg}}(R')(r) & = & \beta^H_{\mathsf{Val}}(R'(r)) \\ & = & \beta^H_{\mathsf{Val}}(\bot_{\mathsf{Val}}) \\ & = & \bot_{\widehat{\mathsf{Val}}} \\ & \sqsubseteq & \hat{R}(m', 0)(r) \end{array} \tag{D.40}$$

which completes the first sub-case. For the second sub-case we have from the definition of R' in the semantics and equations (D.37) and (D.34) that for $1 \leq i \leq n$:

$$\begin{array}{rcll} \beta^H_{\mathsf{Val}}(R'(m'.numLocals - 1 + i)) & = & \beta^H_{\mathsf{Val}}(R(v_i)) & \\ & \sqsubseteq & \hat{R}(m, pc)(v_i) & \\ & \sqsubseteq & \hat{R}(m, 0)(m'.numLocals - 1 + i) & \end{array} \tag{D.41}$$

Combining equations (D.40) and (D.41) we conclude that $\beta^H_{\mathsf{LocalReg}}(R') \sqsubseteq \hat{R}(m', 0)$ which finishes the proof.