Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot

Alexandre Bartel Jacques Klein
Yves Le Traon
University of Luxembourg - SnT, Luxembourg
firstname.lastname@uni.lu

Martin Monperrus
University of Lille - INRIA, France
martin.monperrus@univ.lille1.fr

Abstract

This paper introduces Dexpler, a software package which converts Dalvik bytecode to Jimple. Dexpler is built on top of Dedexer and Soot. As Jimple is Soot's main internal representation of code, the Dalvik bytecode can be manipulated with any Jimple based tool, for instance for performing point-to or flow analysis.

Categories and Subject Descriptors D.3.4 [*Software*]: Programming Languages—Code generation

General Terms Code Generation

Keywords Dalvik Bytecode, Android, Soot, Jimple, Static Analysis

1. Introduction

Android applications are mainly written in Java. However, they are not distributed as Java bytecode but rather as Dalvik bytecode. Indeed, the original Java code is first compiled into Java bytecode which is then transformed into Dalvik bytecode by the *dx* tool[1]. Dalvik bytecode is register based and optimized to run on devices where memory and processing power are scarce.

Analyzing Android applications with Java static analysis tools means either that the Java source code or the Java bytecode of the Android application must be available. Most of the time, Android applications developers do not distribute the source code of their applications. One must then analyze the bytecode, for instance for malware detection.

[1] *dx* is part of the Android SDK available at `http://developer.android.com/sdk/index.html`

SOAP'12 June 14, Beijing, China.

Thus, to analyse Android applications, one is forced to use a Dalvik disassembler such as Smali [2] or Androguard [5]. The problem with disassemblers is that they generaly use their own representation of the bytecode which prevents them to use existing tools.

Another possibility is to first convert Dalvik bytecode to Java bytecode using Ded [7], Dex2jar [16] or undx [17] and then use Java tailored static analysis tools such as Soot [20], BCEL [4] or WALA [9]. Tools which generate Java bytecode can leverage existing Java bytecode analyzers. However, the conversion from Dalvik to Java bytecode could be avoided by directly converting Dalvik bytecode to the internal representation of a tool.

We introduce Dexpler, a Soot modification which allows Soot to directly read Dalvik bytecode and perform analysis and/or transformation on it's internal Jimple representation. Using this method eliminates the intermediate Dalvik to Java bytecode conversion step and enables to use a faster and simpler tool chain for static analysis. Dexpler only uses a disassembler and then does the rest of the work itself or by using Soot.

The contributions of this paper are the following:

- we describe a Dalvik to Jimple converter tool
- we provide a comprehensive table which maps Dalvik bytecode instructions to Jimple statements

The reminder of this paper is organized as follows. In Section 2 we explain what Soot is, and how it has been modified to handle Dalvik bytecode. Section 3 is an overview of the Dalvik bytecode. In Section 4 we propose a Soot modification called Dexpler which enables Soot to read Dalvik bytecode. In Section 5 we evaluate Dexpler on test cases and on one Android application, present and discuss the results. Section 6 explains the current limitation of our tool. We present the related work in Section 7. Finally we conclude the paper and discuss open research challenges in Section 8.

2. Soot

In this Section we give a brief overview of Soot and then describe how we incorporate Dexpler in Soot.

2.1 Soot Overview

Soot [11, 20] was created as a Java compiler testbed at McGill University. It has evolved to become a Java static analysis and transformation tool.

Soot can be used as a code analyzer to, among others, check that certain properties hold [22] or guarantee correctness of programs [8].

Multiple tools based on Soot have been developed to perform transformations such as translation of Java to C [21], instrumentation of Java programs [23], obfuscator for Java [18], software watermarking [3], ...[2].

Soot accepts Java source code, Java bytecode and Jimple source code as input files. Whatever the input format, it is converted into Soot's internal representation: Jimple. Java sIMPLE, is a stack-less, three address representation which features only 15 instructions. Any method code can be viewed as a graph of Jimple statements associated with a list of Jimple local variables.

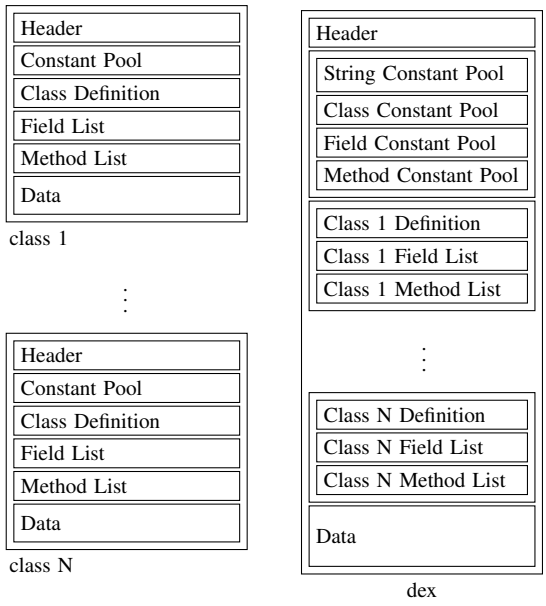
2.2 From Java Bytecode to Jimple

We now describe how Soot handles Java bytecode classes.

In a typical case, Soot is launched by specifying the target directory as a parameter. This directory contains the code of the program to analyze, called `Application Code` (only Java bytecode in this example). First, the `main()` method of the `Main` class is executed and calls `Scene.load NecessaryClasses()`. This method loads basic Java classes and then loads specific `Application classes` by calling `loadClass()`. Then, `SootResolver.resolveClass()` is called. The resolver calls `SourceLocator.getClass Source()` to fetch a reference to a `ClassSource`, an interface between the file containing the Java bytecode and Soot. In our case the class source is a `CoffiClassSource` because it is the *coffi* module which handles the conversion from Java bytecode to Jimple. When the resolver has a reference to a class source, it calls `resolve()` on it. This methods in turn calls `soot.coffi.Util.resolveFromClass File()` which creates a `SootClass` from the corresponding Java bytecode class. All source fields of Soot class' methods are set to refer to a `CoffiMethodSource` object. This object is used later to get the Jimple representation of the method.

For instance, if during an analysis with Soot the analysis code calls `SootMethod.getActiveBody()` and the Jimple code of the method was not already generated, `getActiveBody()` will call `CofficMethodSource.get Body()` to compute Jimple code from the Java bytecode. The Jimple code representation of the method can then be analyzed and/or transformed.

[2] see `https://svn.sable.mcgill.ca/wiki/index.cgi/SootUsers` for a comprehensive list



(a) Class Files (b) Dex File

Figure 1. Dalvik Dex and Java Class

2.3 Soot and Dalvik

Soot is missing a Dalvik to Jimple transformation module. We implemented such a module called Dexpler and incorporated it to Soot using the same structure as Soot's Java bytecode parser module, *coffi* by adding the `DalvikClassSource` and DalvikMethodSource classes.

3. Dalvik Bytecode

We present in this Section the structure of a `.dex` file containing Dalvik classes and Dalvik bytecode.

3.1 Overall Structure

A single Dalvik executable is produced from N Java bytecode classes through the *dx* compiler. The resulting Dalvik bytecode is stored in a `.dex` file as represented in Figure 1b.

As represented in Figure 1a, there is only a single place where literal constant values are stored (constant pool) per Java class. It is heterogeneous since different kind of Objects are mixed together (ex: Class, MethodRef, Integer, String, ...). A `.dex` file contains four homogeneous constants pools: for Strings, Class, Fields and Methods. It is shared by all the classes. A `.dex` file contains multiple *Class Definitions* each containing one or more *Method definition* each of those being linked to Dalvik bytecode instructions present in the *Data* section.

3.2 Dalvik Instruction

The Dalvik virtual machine is register based. This means most instructions must specify the registers which they ma-

```
int i = 0;                          00: const/4 v0, #int 0
Object o = null;                    01: const/4 v1, #int 0
(Java)                              (Dalvik)
```

Figure 2. Dalvik Representation of *null* and *zero*

nipulate. Registers could be specified on 4, 8 or 16 bits depending on the instruction.

There are 237 opcodes present in the Dalvik opcode constant list[3]. However, 12 odex (optimized dex) instructions can not be found in Android applications Dalvik bytecode as they are unsafe instructions generated within the Android system to optimize Dalvik bytecode. Moreover, 8 instructions were never found in application code [15]. According to those numbers, only 217 instructions can be found in Android PacKages (.apk) in practice.

The set of instructions can be divided between instructions which provide the type of the registers they manipulate (ex: `sub-long v1, v2, v3`) and those which do not (ex: `const v0, 0xBEEF`). Moreover, there is no distinction between `NULL` and `0` which are both represented as `0` (see Figure 2). As we will see in Section 4, the lack of type and the `NULL` representation become problematic when translating the Dalvik bytecode to Jimple.

4. Dexpler

This section describes Dexpler, the Dalvik to Jimple converter tool. It leverages the *dedexer* [14] Dalvik bytecode disassembler and the Soot *fast typing* Jimple component implementing a type inferrence algorithm [1] for local variables. We first give a brief overview on *dedexer* and on how Dexpler is working in Sections 4.1 and 4.2, respectively. Then, we detail issues we have to deal with.

4.1 Dedexer

Our tool leverages *dedexer* a Dalvik bytecode parser and disassembler which generates Jasmin [10, 12] like text files containing Dalvik instructions instead of Java instructions. We generate Jimple classes, methods and statements from the informations provided by *dedexer*'s dex file parser.

4.2 Overview

Dalvik bytecode instructions are first mapped to Jimple statements and registers mapped to Jimple local variables. The type of local variables is set to `UnknownType`. Then, Soot's Jimple component, *fast typing*, is applied to infer the type of the local variables. The third and last step consists in applying Soot's Jimple pack *jop*, which features components such as *nop eleminitor*, to optimize the generated Jimple code.

[3] dalvik/bytecode/Opcodes.java

4.3 Instruction Mapping

Each Dalvik instruction is mapped to a corresponding (or a group of) Jimple statements. A comprehensive mapping is represented in Table 1 in Appendix A. Unused opcodes are marked as '-' and odex opcodes as '*odex*'. There are five main groups of instructions: move instructions (0x01 to 0x1C), branch instructions (0x27 to 0x3D), getter and setter instructions (0x44 to 0x6D), method invoke instructions (0x6E to 0x78) logic and arithmetic instructions (0x7B to 0xE2).

4.4 Type Inference

The type for local variables is inferred using the *fast typing* Soot component. However, the inference algorithm sometime generates an exception and stop because some Dalvik instructions (such as the constant initialization instructions `0x12` to `0x19`) do not provide enough information and thus confuse the inference engine.

The lack of type is present in the following instructions:

- null initialization instructions (zero or null?)
- initialization instructions (32 bits: integer or float?, 64 bits: long or double?)

Null Initialization Figure 4 illustrates the problem with a bytecode snippet generated from the Java code of Figure 3. Register `v0` is initialized with 0 at `01`. At this point we do not know if `v0` is an integer, a float or a reference to an object. At `02` we still do not have the answer. We have to wait until instruction at `04` to known that the type of `v0` is `Coordinate`. At this point, the Jimple instruction generated for `01` has to be updated to use `NullConstant` instead of the default `IntConstant(0)`. If this is not handled correctly, the *fast typing* component generates an exception and stops.

Numeric Constant Initialization Similarly, *float* constants initialization cannot be distinguished from *int* constants initialization and *double* constants initialization from *long* constants initialization. Thus, we go through the graph of Jimple statements to find how constants are used and correct the initializations Jimple statements when needed. For instance, if a *float/int* constant (initialized by default to *int* in the Jimple statement) is later used in a *float* addition, the constant initialization changes from `IntConstant(c)` to `FloatConstant(Float.intBitsToFloat(c))`.

We implemented the algorithm described by Enck et al. [6]. It is based on algorithms which extract typing information for a variable by looking at how it is used in operations whithin which the type of the operands is knows (ex: the variable is used as a parameter of a method invocation) [13, 19]. For each ambiguous register declaration, the algorithm performs a depth first search in the control flow graph of Jimple statements to find out how the declared local variable *dv* (registers are mapped to Jimple local variables) is used. The type of *dv* is exposed with the following state-

```
Coordinate newCoord = null;
while (newCoord!=null) {
 newCoord = new Coordinate(1,1);
}
if (newCoord == null) {
 [...]
}
```

Figure 3. Illustration of the *null* init problem.

```
00: const/4 v1, #int 1
01: const/4 v0, #int 0
02: if-eqz v0, 000a
04: new-instance v0, LCoordinate;
06: invoke {v0, v1, v1}, LCoordinate;.<init>:(II)V
09: goto 0002
0a: if-nez v0, 0013
[...]
13: ...
```

Figure 4. Resulting Dalvik Bytecode from Figure 3

ments: comparison with a known type, instructions operating only on specific types (ex: neg-float), non-void return instructions and method invocation. The search in a branch of the graph is terminated if either the local variable is reassigned (new declaration) or if there is no more statement that follow the current one (eg: the current statement is a return or throw statement). When the type information is found it is forward propagated to all subsequent ambiguous uses between the target ambiguous declaration of *dv* and any new declaration of *dv*.

4.5 Handling Branches

Dalvik instructions are mapped to Jimple statements. When parsing Dalvik bytecode, we keep a mapping between bytecode instructions addresses and Jimple statements. Thus, when a Dalvik branch instruction is parsed, a Jimple jump instruction is generated and its target is retrieved by fetching the Jimple statement mapped to the Dalvik branch instruction target's address. We add a `nop` instruction as the first instruction of every Jimple methods. This way, if the first Dalvik instruction is a jump or if the jump's target correspond to a non-yet generated Jimple statement, we redirect it to the this `nop` Jimple instruction. We correct those Jimple jump instructions once the whole Dalvik bytecode of the method has been processed: at this point we know the target Jimple statement mapped to the Dalvik jump's target address. The Jimple `nop` instruction we add is removed during the Jimple optimization step.

Branching instructions often rely on the result of a comparison of two registers. Dalvik comparisons between *double* or *float* are explicit and provide typing information. However, when a register r is compared with zero one has to check the type of r. If it is an object, we change the zero value to *null* since it is a comparison between objects. We do this change when the *fast typing* component has finished. Indeed, comparisons do not influence the type inference. For example, the Jimple statement generated from `02` in Figure 4 has to be updated to use `NullConstant` instead of `IntConstant(0)`. If this is not handled correctly the bytecode generated from Jimple statements does not run correctly and generates an exception similar to the following one: `Exception in thread "main" java.lang.VerifyError: Expecting to find integer on stack`.

Dexpler enables us to transform Dalvik bytecode to Jimple representation. From there, Soot can be used as a static analysis tool to analyze the code. The next Section evaluates Dexpler.

5. Evaluation

We evaluate Dexpler using test cases, and one Android application: Snake.

5.1 Test Cases

The first step is to generate the Dalvik bytecode for every test case. The test cases are written in Java then compiled into Java bytecode using *javac* and finally converted into Dalvik bytecode using *dx*. The second step is to execute Dexpler on every generated Dalvik bytecode test case. This generates `.jimple` and `.class` files. We then compare the execution result from of the versions produced from the original Java bytecode and the Java bytecode produced by Soot from the Dalvik bytecode. Executions of the `.class` files give the correct result.

We wrote test cases for arithmetic operations, branches, method calls, array initialization, string manipulation, null and zero usage, exceptions and casts.

Since simple test cases do not reflect a real application we also evaluated our tool on one Android application.

5.2 Android Application

The snake application is a demonstration application developed by the Android team to showcase the Android platform.[4] It features 11 classes, 39 methods and was written in 550 lines of Java code. The generated Dalvik bytecode takes 14 KiB and contains 884 Dalvik instructions.

From the Dalvik bytecode of the Snake application we generate Jimple code in one second (duration for the Dalvik to Jimple conversion only). Then we ask Soot to generate Java bytecode from the Jimple representation. We convert the Java bytecode back to Dalvik, repackage an Android application and launch it on the Android emulator.

The application runs smoothly and the game is working.

5.3 Static Analysis on Snake

We use Soot to generate a call graph of the Snake application as well as a control flow graph represented in Figure 5 in 14

[4] `http://developer.android.com/resources/samples/Snake/index.html`

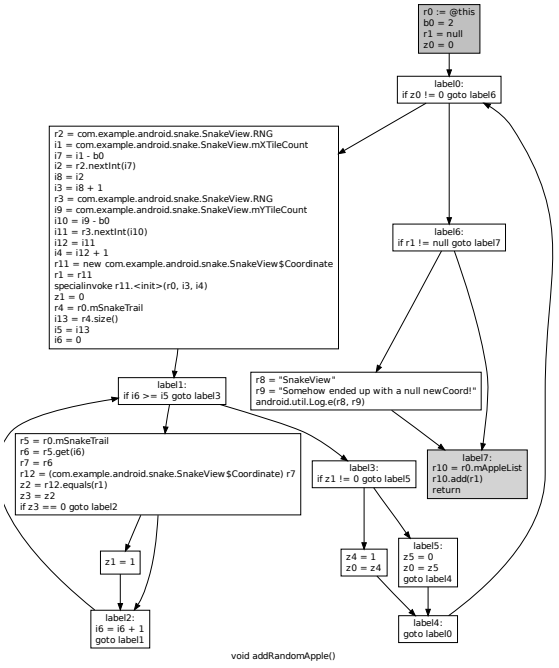


Figure 5. Control Flow Graph for `addRandomApple` Method Extracted from the Generated Jimple Representation.

seconds (duration from the launch time of Soot until Soot has finished). We perform this to check that the generated call graph and CFG correspond to the original code meaning that the conversion from Dalvik to Jimple is correct for this code.

We have successfully tested our prototype tool on test cases as well as on an Android application.

6. Current Liminations

The current version of Dexpler is able to transform Android applications such as the Snake game.

However, it does not handle optimized Dalvik (odex) opcodes.

Moreover, when inferring types for ambiguous declarations the algorithm supposes that the Dalvik bytecode is legal in the sense that it was generated from Java source code and not hand-crafted by malicious developers. In the later case assumptions such as "comparisons always involve variables with the same type" may not hold anymore and may make Dexpler to infer wrong types.

7. Related Work

To our knowledge no existing tool directly converts Dalvik bytecode to Jimple. We either found tools to convert Dalvik bytecode to Java bytecode or tools to disassemble and/or assemble Dalvik bytecode using an intermediate representation.

Dalvik to Java Bytecode Converter Ded [7] is a Dalvik bytecode to Java bytecode converter. Once the Java bytecode is generated, Soot is used to optimize the code. Dex2jar [16] also generates Java bytecode from Dalvik bytecode but no not use any external tool to optimize the resulting Java bytecode. Undx [17] is also a Dalvik to Java bytecode converter but seems to be unavailable.

We on the other hand do not directly generate Java bytecode but Jimple code. From there, since the Jimple code is within Soot, we can generate Java bytecode as well.

Dalvik Assembler/Disassembler Smali [2] or Androguard [5] can be used to reverse engineer Dalvik bytecode. They use their own representation of the Dalvik bytecode: they can not leverage existing analysis tools.

Our tool, use Soot's internal representation which allows existing tools to analyze/transform the Dalvik bytecode.

8. Conclusion

We have presented Dexpler a Soot modification with enables Soot to analyse Dalvik bytecode and thus Android applications. This tool leverages *dedexer* for the parsing of Dalvik `dex` files and Soot's *fast typing* component for the type inference.

Dexpler converts every Dalvik instruction to Jimple. We are working on improving Dexpler to make it robust to yet unhandled typing issues. Once this step is done we will look at the performance of this tool compared to current Java bytecode generation and analysis tools.

Acknowledgments

This research is supported by the National Research Fund, Luxembourg.

References

[1] B. Bellamy, P. Avgustinov, O. de Moor, and D. Sereni. Efficient local type inference. In G. E. Harris, editor, *OOPSLA*, pages 475–492. ACM, 2008. ISBN 978-1-60558-215-3.

[2] Ben Gruver, et al. Smali: An assembler/disassembler for android's dex format. `http://code.google.com/p/smali/`, Last accessed: March 20, 2012.

[3] P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. *ACM SIGPLAN Notices*, 39(1):173–185, Jan. 2004. ISSN 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic).

[4] M. Dahm. Byte code engineering. In *Proceedings of Java-Informations-Tage (JIT'99)*, pages 267–277, Düsseldorf, Deutchland, Sept. 1999. ISBN 3-540-66464-5.

[5] A. Desnos and G. Gueguen. Android: From reversing to decompilation. In *Blackhat*, 2011.

[6] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX Security Symposium*, San Francisco, CA, aug 2011.

[7] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proc. USENIX Security' 11*, pages 21–21, Berkeley, CA, USA, 2011.

[8] L.-Å. Fredlund. Guaranteeing correctness properties of a java card applet. *Electr. Notes Theor. Comput. Sci*, 113:217–233, 2005.

[9] IBM. The T.J. Watson Libraries for Analysis (Wala). `http://wala.sourceforge.net`, Last accessed: March 20, 2012.

[10] Jonathan Meyer, Daniel Reynaud. Jasmin. `http://jasmin.sourceforge.net`.

[11] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Oct. 2011. URL `http://www.bodden.de/pubs/lblh11soot.pdf`.

[12] J. Meyer and T. Downing. *Java Virtual Machine*. O'Reilly, 1997.

[13] R. Milner. A theory of type polymorphism in programming. *JCSS*, 17:348–375, 1978.

[14] G. Paller. Dedexer. `http://dedexer.sourceforge.net/`, Last accessed: March 20, 2012.

[15] G. Paller. Dalvik opcodes. `http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html`, Last accessed: March 20, 2012.

[16] Panxiaobo, et al. Dex2jar: Tools to work with android .dex and java .class files. `http://code.google.com/p/dex2jar/`, Last accessed: March 20, 2012.

[17] M. Schnefeld. Reconstructing dalvik applications. In *CONFidence*, 2009.

[18] M. Sosonkin, G. Naumovich, and N. Memon. Obfuscation of design intent in object-oriented applications. In M. Yung, editor, *Proceedings of the 2003 ACM workshop on Digital rights management (DRM-03)*, pages 142–153, New York, Oct. 27 2003. ACM Press.

[19] J. Tiuryn. Type inference problems: A survey. In *MFCS*, pages 105–120, 1990.

[20] R. Vallée-Rai, L. Hendren, V. Sundaresan, E. G. Patrick Lam, and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[21] A. Varma and S. S. Bhattacharyya. Java-through-C compilation: An enabling technology for java in embedded systems. In *DATE*, pages 161–167. IEEE Computer Society, 2004. ISBN 0-7695-2085-5.

[22] E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. *ACM SIGPLAN Notices*, 39(6):25–34, May 2004. ISSN 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic).

[23] C. Zhang, D. Yan, J. Zhao, Y. Chen, and S. Yang. BPGen: an automated breakpoint generator for debugging. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *ICSE (2)*, pages 271–274. ACM, 2010. ISBN 978-1-60558-719-6.

A. Jimple Code

Table 1: Jimple Code representation of Dalvik Instructions

Opcode	Opcode name	Jimple Code
0x00	nop	nop
0x01	move vx,vy	vx = vy
0x02	move/from16 vx,vy	vx = vy
0x03	move/16	vx = vy
0x04	move-wide	vx = vy
0x05	move-wide/from16 vx,vy	vx = vy
0x06	move-wide/16	vx = vy
0x07	move-object vx,vy	vx = vy
0x08	move-object/from16 vx,vy	vx = vy
0x09	move-object/16	vx = vy
0x0A	move-result vx	vx = mres
0x0B	move-result-wide vx	vx = mres
0x0C	move-result-object vx	vx = mres
0x0D	move-exception vx	vx = mres
0x0E	return-void	return
0x0F	return vx	return vx
0x10	return-wide vx	return vx
0x11	return-object vx	return vx
0x12	const/4 vx,lit4	vx = lit4
0x13	const/16 vx,lit16	vx = lit16
0x14	const vx, lit32	vx = lit32
0x15	const/high16 v0, lit16	vx = lit16 << 16
0x16	const-wide/16 vx, lit16	vx = lit16
0x17	const-wide/32 vx, lit32	vx = lit32
0x18	const-wide vx, lit64	vx = lit64
0x19	const-wide/high16 vx,lit16	vx = lit16 << 48
0x1A	const-string vx,string_id	vx = string
0x1B	const-string-jumbo vx,string	vx = string
0x1C	const-class vx,type_id	vx = class ”type”
0x1D	monitor-enter vx	monitorenter vx
0x1E	monitor-exit vx	monitorexit vx
0x1F	check-cast vx, type_id	checkcast = (type) vx
0x20	instance-of vx,vy,type_id	vx = vy instanceof type
0x21	array-length vx,vy	vx = length(vy)
0x22	new-instance vx,type	vx = new type
0x23	new-array vx,vy,type_id	vx = new type[vy]
0x24	filled-new-array {parameters},type_id	vx = new array_type[size]; vx[0] = e1; ... vx[N] = eN;
0x25	filled-new-array-range {vx..vy},type_id	vx = new array_type[size]; vx[0] = e1; ... vx[N] = eN;
0x26	fill-array-data vx,array_data_offset	vx[0] = e1; ... vx[N] = eN;
0x27	throw vx	throw vx
0x28	goto target	goto target
0x29	goto/16 target	goto target
0x2A	goto/32 target	goto target
0x2B	packed-switch vx,table	switch (vx) { case C1: goto target1; ... case CN: goto targetN; }

Table 1: Jimple Code representation of Dalvik Instructions

Opcode	Opcode name	Jimple Code
0x2C	sparse-switch vx,table	switch (vx) { case C1: goto target1; ... case CN: goto targetN; }
0x2D	cmpl-float	vx = vy cmpl vz
0x2E	cmpg-float vx, vy, vz	vx = vy cmpg vz
0x2F	cmpl-double vx,vy,vz	vx = vy cmpl vz
0x30	cmpg-double vx, vy, vz	vx = vy cmpg vz
0x31	cmp-long vx, vy, vz	vx = vy cmp vz
0x32	if-eq vx,vy,target	if (vx == vy) goto target;
0x33	if-ne vx,vy,target	if (vx != vy) goto target;
0x34	if-lt vx,vy,target	if (vx < vy) goto target;
0x35	if-ge vx, vy,target	if (vx >= vy) goto target;
0x36	if-gt vx,vy,target	if (vx > vy) goto target;
0x37	if-le vx,vy,target	if (vx <= vy) goto target;
0x38	if-eqz vx,target	if (vx == 0) goto target;
0x39	if-nez vx,target	if (vx != 0) goto target;
0x3A	if-ltz vx,target	if (vx < 0) goto target;
0x3B	if-gez vx,target	if (vx >= 0) goto target;
0x3C	if-gtz vx,target	if (vx > 0) goto target;
0x3D	if-lez vx,target	if (vx <= 0) goto target;
0x3E	unused_3E	-
0x3F	unused_3F	-
0x40	unused_40	-
0x41	unused_41	-
0x42	unused_42	-
0x43	unused_43	-
0x44	aget vx,vy,vz	vx = vy[vz]
0x45	aget-wide vx,vy,vz	vx = vy[vz]
0x46	aget-object vx,vy,vz	vx = vy[vz]
0x47	aget-boolean vx,vy,vz	vx = vy[vz]
0x48	aget-byte vx,vy,vz	vx = vy[vz]
0x49	aget-char vx, vy,vz	vx = vy[vz]
0x4A	aget-short vx,vy,vz	vx = vy[vz]
0x4B	aput vx,vy,vz	vy[vz] = vx
0x4C	aput-wide vx,vy,vz	vy[vz] = vx
0x4D	aput-object vx,vy,vz	vy[vz] = vx
0x4E	aput-boolean vx,vy,vz	vy[vz] = vx
0x4F	aput-byte vx,vy,vz	vy[vz] = vx
0x50	aput-char vx,vy,vz	vy[vz] = vx
0x51	aput-short vx,vy,vz	vy[vz] = vx
0x52	iget vx, vy, field_id	vx = field_id
0x53	iget-wide vx,vy,field_id	vx = field_id
0x54	iget-object vx,vy,field_id	vx = field_id
0x55	iget-boolean vx,vy,field_id	vx = field_id
0x56	iget-byte vx,vy,field_id	vx = field_id
0x57	iget-char vx,vy,field_id	vx = field_id
0x58	iget-short vx,vy,field_id	vx = field_id
0x59	iput vx,vy, field_id	field_id = vx
0x5A	iput-wide vx,vy, field_id	field_id = vx
0x5B	iput-object vx,vy,field_id	field_id = vx
0x5C	iput-boolean vx,vy, field_id	field_id = vx

Table 1: Jimple Code representation of Dalvik Instructions

Opcode	Opcode name	Jimple Code
0x5D	iput-byte vx,vy,field_id	field_id = vx
0x5E	iput-char vx,vy,field_id	field_id = vx
0x5F	iput-short vx,vy,field_id	field_id = vx
0x60	sget vx,field_id	vx = field_id
0x61	sget-wide vx, field_id	vx = field_id
0x62	sget-object vx,field_id	vx = field_id
0x63	sget-boolean vx,field_id	vx = field_id
0x64	sget-byte vx,field_id	vx = field_id
0x65	sget-char vx,field_id	vx = field_id
0x66	sget-short vx,field_id	vx = field_id
0x67	sput vx, field_id	field_id = vx
0x68	sput-wide vx, field_id	field_id = vx
0x69	sput-object vx,field_id	field_id = vx
0x6A	sput-boolean vx,field_id	field_id = vx
0x6B	sput-byte vx,field_id	field_id = vx
0x6C	sput-char vx,field_id	field_id = vx
0x6D	sput-short vx,field_id	field_id = vx
0x6E	invoke-virtual { parameters }, methodtocall	invoke-virtual
0x6F	invoke-super {parameter},methodtocall	invoke-special
0x70	invoke-direct { parameters }, methodtocall	invoke-special
0x71	invoke-static {parameters}, methodtocall	invoke-static
0x72	invoke-interface {parameters},methodtocall	invoke-interface
0x73	unused_73	-
0x74	invoke-virtual/range {vx..vy},methodtocall	invoke-virtual
0x75	invoke-super/range	invoke-special
0x76	invoke-direct/range {vx..vy},methodtocall	invoke-special
0x77	invoke-static/range {vx..vy},methodtocall	invoke-static
0x78	invoke-interface-range	invoke-interface
0x79	unused_79	-
0x7A	unused_7A	-
0x7B	neg-int vx,vy	vx = 0 -vy
0x7C	not-int vx,vy	vx = vy ^ (-1)
0x7D	neg-long vx,vy	vx = 0 -vy
0x7E	not-long vx,vy	vx = vy ^ (-1)
0x7F	neg-float vx,vy	vx = 0 -vy
0x80	neg-double vx,vy	vx = 0 -vy
0x81	int-to-long vx, vy	vx = (long) vy
0x82	int-to-float vx, vy	vx = (float) vy
0x83	int-to-double vx, vy	vx = (double) vy
0x84	long-to-int vx,vy	vx = (int) vy
0x85	long-to-float vx, vy	vx = (float) vy

Table 1: Jimple Code representation of Dalvik Instructions

Opcode	Opcode name	Jimple Code
0x86	long-to-double vx, vy	vx = (double) vy
0x87	float-to-int vx, vy	vx = (int) vy
0x88	float-to-long vx,vy	vx = (long) vy
0x89	float-to-double vx, vy	vx = (double) vy
0x8A	double-to-int vx, vy	vx = (int) vy
0x8B	double-to-long vx, vy	vx = (long) vy
0x8C	double-to-float vx, vy	vx = (float) vy
0x8D	int-to-byte vx,vy	vx = (byte) vy
0x8E	int-to-char vx,vy	vx = (char) vy
0x8F	int-to-short vx,vy	vx = (short) vy
0x90	add-int vx,vy,vz	vx = vy + vz
0x91	sub-int vx,vy,vz	vx = vy - vz
0x92	mul-int vx, vy, vz	vx = vy * vz
0x93	div-int vx,vy,vz	vx = vy / vz
0x94	rem-int vx,vy,vz	vx = vy % vz
0x95	and-int vx, vy, vz	vx = vy & vz
0x96	or-int vx, vy, vz	vx = vy | vz
0x97	xor-int vx, vy, vz	vx = vy ^ vz
0x98	shl-int vx, vy, vz	vx = vy << vz
0x99	shr-int vx, vy, vz	vx = vy >> vz
0x9A	ushr-int vx, vy, vz	vx = vy >> vz
0x9B	add-long vx, vy, vz	vx = vy + vz
0x9C	sub-long vx,vy,vz	vx = vy - vz
0x9D	mul-long vx,vy,vz	vx = vy * vz
0x9E	div-long vx, vy, vz	vx = vy / vz
0x9F	rem-long vx,vy,vz	vx = vy % vz
0xA0	and-long vx, vy, vz	vx = vy & vz
0xA1	or-long vx, vy, vz	vx = vy | vz
0xA2	xor-long vx, vy, vz	vx = vy ^ vz
0xA3	shl-long vx, vy, vz	vx = vy << vz
0xA4	shr-long vx,vy,vz	vx = vy >> vz
0xA5	ushr-long vx, vy, vz	vx = vy >> vz
0xA6	add-float vx,vy,vz	vx = vy + vz
0xA7	sub-float vx,vy,vz	vx = vy - vz
0xA8	mul-float vx, vy, vz	vx = vy * vz
0xA9	div-float vx, vy, vz	vx = vy / vz
0xAA	rem-float vx,vy,vz	vx = vy % vz
0xAB	add-double vx,vy,vz	vx = vy + vz
0xAC	sub-double vx,vy,vz	vx = vy - vz
0xAD	mul-double vx, vy, vz	vx = vy * vz
0xAE	div-double vx, vy, vz	vx = vy / vz
0xAF	rem-double vx,vy,vz	vx = vy % vz
0xB0	add-int/2addr vx,vy	vx = vx + vy
0xB1	sub-int/2addr vx,vy	vx = vx - vy
0xB2	mul-int/2addr vx,vy	vx = vx * vy
0xB3	div-int/2addr vx,vy	vx = vx / vy
0xB4	rem-int/2addr vx,vy	vx = vx % vy
0xB5	and-int/2addr vx, vy	vx = vx & vy
0xB6	or-int/2addr vx, vy	vx = vx | vy

Table 1: Jimple Code representation of Dalvik Instructions

Opcode	Opcode name	Jimple Code
0xB7	xor-int/2addr vx, vy	vx = vx ^ vy
0xB8	shl-int/2addr vx, vy	vx = vx << vy
0xB9	shr-int/2addr vx, vy	vx = vx >> vy
0xBA	ushr-int/2addr vx, vy	vx = vx >> vy
0xBB	add-long/2addr vx,vy	vx = vx + vy
0xBC	sub-long/2addr vx,vy	vx = vx - vy
0xBD	mul-long/2addr vx,vy	vx = vx * vy
0xBE	div-long/2addr vx, vy	vx = vx / vy
0xBF	rem-long/2addr vx,vy	vx = vx % vy
0xC0	and-long/2addr vx, vy	vx = vx & vy
0xC1	or-long/2addr vx, vy	vx = vx | vy
0xC2	xor-long/2addr vx, vy	vx = vx ^ vy
0xC3	shl-long/2addr vx, vy	vx = vx << vy
0xC4	shr-long/2addr vx, vy	vx = vx >> vy
0xC5	ushr-long/2addr vx, vy	vx = vx >> vy
0xC6	add-float/2addr vx,vy	vx = vx + vy
0xC7	sub-float/2addr vx,vy	vx = vx - vy
0xC8	mul-float/2addr vx, vy	vx = vx * vy
0xC9	div-float/2addr vx, vy	vx = vx / vy
0xCA	rem-float/2addr vx,vy	vx = vx % vy
0xCB	add-double/2addr vx, vy	vx = vx + vy
0xCC	sub-double/2addr vx, vy	vx = vx - vy
0xCD	mul-double/2addr vx, vy	vx = vx * vy
0xCE	div-double/2addr vx, vy	vx = vx / vy
0xCF	rem-double/2addr vx,vy	vx = vx % vy
0xD0	add-int/lit16 vx,vy,lit16	vx = vy + lit16
0xD1	sub-int/lit16 vx,vy,lit16	vx = vy - lit16
0xD2	mul-int/lit16 vx,vy,lit16	vx = vy * lit16
0xD3	div-int/lit16 vx,vy,lit16	vx = vy / lit16
0xD4	rem-int/lit16 vx,vy,lit16	vx = vy % lit16
0xD5	and-int/lit16 vx,vy,lit16	vx = vy & lit16
0xD6	or-int/lit16 vx,vy,lit16	vx = vy | lit16
0xD7	xor-int/lit16 vx,vy,lit16	vx = vy ^ lit16
0xD8	add-int/lit8 vx,vy,lit8	vx = vy + lit8
0xD9	sub-int/lit8 vx,vy,lit8	vx = vy - lit8
0xDA	mul-int/lit-8 vx,vy,lit8	vx = vy * lit8
0xDB	div-int/lit8 vx,vy,lit8	vx = vy / lit8
0xDC	rem-int/lit8 vx,vy,lit8	vx = vy % lit8
0xDD	and-int/lit8 vx,vy,lit8	vx = vy & lit8
0xDE	or-int/lit8 vx, vy, lit8	vx = vy | lit8
0xDF	xor-int/lit8 vx, vy, lit8	vx = vy ^ lit8
0xE0	shl-int/lit8 vx, vy, lit8	vx = vy + lit8
0xE1	shr-int/lit8 vx, vy, lit8	vx = vy + lit8
0xE2	ushr-int/lit8 vx, vy, lit8	vx = vy + lit8
0xE3	unused_E3	-
0xE4	unused_E4	-
0xE5	unused_E5	-
0xE6	unused_E6	-
0xE7	unused_E7	-

Table 1: Jimple Code representation of Dalvik Instructions

Opcode	Opcode name	Jimple Code
0xE8	unused_E8	-
0xE9	unused_E9	-
0xEA	unused_EA	-
0xEB	unused_EB	-
0xEC	unused_EC	-
0xED	unused_ED	-
0xEE	execute-inline {parameters},inline ID	odex
0xEF	unused_EF	-
0xF0	invoke-direct-empty	odex
0xF1	unused_F1	-
0xF2	iget-quick vx,vy,offset	odex
0xF3	iget-wide-quick vx,vy,offset	odex
0xF4	iget-object-quick vx,vy,offset	odex
0xF5	iput-quick vx,vy,offset	odex
0xF6	iput-wide-quick vx,vy,offset	odex
0xF7	iput-object-quick vx,vy,offset	odex
0xF8	invoke-virtual-quick {parameters},vtable offset	odex
0xF9	invoke-virtual-quick/range {parameter range},vtable offset	odex
0xFA	invoke-super-quick {parameters},vtable offset	odex
0xFB	invoke-super-quick/range {register range},vtable offset	odex
0xFC	unused_FC	-
0xFD	unused_FD	-
0xFE	unused_FE	-
0xFF	unused_FF	-