Retargeting Android Applications to Java Bytecode

Damien Octeau
Department of Computer Science and Engineering
Pennsylvania State University
octeau@cse.psu.edu

Somesh Jha
Computer Sciences Department
University of Wisconsin
jha@cs.wisc.edu

Patrick McDaniel
Department of Computer Science and Engineering
Pennsylvania State University
mcdaniel@cse.psu.edu

ABSTRACT

The Android OS has emerged as the leading platform for SmartPhone applications. However, because Android applications are compiled from Java source into platform-specific Dalvik bytecode, existing program analysis tools cannot be used to evaluate their behavior. This paper develops and evaluates algorithms for retargeting Android applications received from markets to Java class files. The resulting `Dare` *tool uses a new intermediate representation to enable fast and accurate retargeting.* `Dare` *further applies strong constraint solving to infer typing information and translates the 257 DVM opcodes using only 9 translation rules. It also handles cases where the input Dalvik bytecode is unverifiable. We evaluate* `Dare` *on 1,100 of the top applications found in the free section of the Android market and successfully retarget 99.99% of the 262,110 associated classes. Further, whereas existing tools can only fully retarget about half of these applications,* `Dare` *can recover over 99% of them. In this way, we open the door to users, developers and markets to use the vast array of program analysis tools to ensure the correct operation of Android applications.*

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

Keywords

Android, Dalvik bytecode, Dalvik retargeting

1. INTRODUCTION

Android now hosts more smartphones worldwide than any other mobile platform, and its market share is increasing quickly [11, 18]. The corresponding markets are delivering an astonishing array of applications – as of March 2012, the number of applications available from the Android Market alone doubled to 400,000 in just eight months [3]. However, existing markets provide little meaningful security or privacy guarantees because market providers have neither the tools nor the resources to perform detailed analysis of submitted applications [22]. Thus, users fall victim to bad applications with moderate to devastating results [4, 8, 13, 17].

At the time of registration, Android developers submit an "application package" containing the program bytecode, resources and an XML manifest to the market. The submitted applications are initially developed in Java, but compiled by the developer into Dalvik bytecode [12]. Android runs each application on the phone in its own instance of the Dalvik virtual machine (DVM). The DVM has some major differences with traditional JVM. For example, the DVM is a register-based architecture and has ambiguous register typing (see Section 2). These different bytecode and program structures make it impossible to leverage existing Java tools such as Soot [37] or WALA [15] for program analysis of Android applications. Thus, in the absence of usable analysis tools, markets can do little to vet applications.

In this paper, we develop and evaluate algorithms for retargetting Dalvik to Java bytecode. The resulting tool is called `Dare` (DAlvik REtargeting)[1]. `Dare` was motivated by our past analysis of a large corpus of applications found in the Android market. In this study, we used a rudimentary retargeting tool to perform a study of security properties of Android applications [9]. The output of that retargeting tool was input to Dava [24]–a Java decompiler integrated with the Soot toolkit [37]–and the resulting source code analyzed using Fortify SCA [14]. We found that the *ad hoc* methods used for retargeting were often unreliable or failed outright. These failures limited the visibility of the code (and thus the coverage of the analysis), and prevented conclusive results. More specifically, while we were often able to retarget and eventually decompile portions of the application code, about half the applications had classes which were unrecoverable, which made program analysis of complete applications impossible. Further, the *ad hoc* tool was targeting decompilers and its success rate was measured in terms of decompilation success. For `Dare`, we do not target a specific tool but instead seek to produce verifiable Java bytecode, which ensures that it is accepted by analysis tools.

By providing the Java bytecode of Android applications via `Dare`, we provide a path for users, developers, application market providers (such as Amazon) to perform analysis on Android applications. The following sections detail the structure of `Dare`. Principally, we focus on solutions that address the key challenges of retargeting Dalvik bytecode. Our paper makes the following contributions:

- We introduce the Tyde intermediate representation for structured semantic mapping between the VMs. All 257 Dalvik instructions are translated using only 9 translation rules.
- Because sound bytecode typing is necessary for verifiability, we use a strong constraint-based type inference algorithm.
- We introduce code transformations to fix unverifiable input bytecode. In addition to making the code verifiable, these transformations accurately mirror VM runtime behavior.

[1]Source code and documentation for `Dare` are available at [32].

- We evaluate our algorithms on a sample of 1,100 applications. We successfully retarget 99.99% of the 262,110 classes. Further, while previous tools were able to completely recover less than 60% of the applications in the corpus, we recover over 99%. Retargeting is efficient, taking less than 20 minutes for the entire sample. Finally, our experiments reveal that over 20% of applications in the sample have unverifiable Dalvik bytecode in at least one class.

The remainder of this paper explores the algorithms and structure of `Dare`. The next two sections provide background on the retargeting challenges and outline the `Dare` retargeting process. Next, we describe how DVM bytecode is translated into the Tyde intermediate representation (Sections 4 and 5) and then converted to Java bytecode (Section 6). Next, we show the causes of unverifiability in Dalvik bytecode and how to reliably retarget unverifiable bytecode (Section 7). We then present the empirical study of `Dare` (Section 8), show related work (Section 9) and conclude.

2. RETARGETING CHALLENGES

Android applications are developed in Java and compiled into JVM class files. Then, the various class files comprising the application are retargeted to the DVM and coalesced into a single `.dex` file. Thus, Java bytecode is an intermediate representation for DVM bytecode. `Dare` reverses the lossy JVM to DVM bytecode compilation to allow subsequent program analysis, e.g., [9].

In order to make sure that the bytecode we generate can be used by existing analysis tools, we aim to generate *verifiable* bytecode. Since not all code that is loaded by a VM necessarily comes from a trusted compiler, the verification process ensures that code can be safely executed. The Java verification process is precisely described in the JVM specification [21, §4.9]. The Dalvik verification process is mostly defined in the Dalvik source code documentation [12] (we inferred part of it directly from the source code). The JVM and DVM verifiers both check for similar static constraints (i.e., answering the question: "is the file well-formed?") and structural constraints (i.e., relationships between instructions). They also both implement a similar type inference algorithm, which consists in symbolically executing instructions one by one to infer the influence that each instruction has on the types of the local variables (Java) or registers (Dalvik). This algorithm follows all possible execution paths and iterates as long as new type information is found for a variable or a register. This part of the verification process ensures that the bytecode is type safe.

The differences between the JVM and DVM make the retargeting process difficult. To aid the following discussions, we give some background on the main retargeting challenges.

Instruction Set - Dalvik instructions are vastly different from Java instructions. DVM bytecode has 257 different instructions and 3 pseudo-instructions. Dalvik instructions are two to ten bytes long, and pseudo-instructions have a variable length. The DVM has substantially more instruction formats (over 20) than the JVM.

Pseudo-instructions are used to store extra information related to other instructions (and thus are never executed). Specifically, the Dalvik switch instructions (`packed-switch` and `sparse-switch`) store an offset to a pseudo-instruction. The data describing the switch statement (case values and targets) is stored in a pseudo-instruction placed at the end of the bytecode block. The `fill-array-data` instruction fills an array of primitive elements with values stored in a pseudo-instruction.

The DVM is register-based, whereas the JVM is stack-based. Thus, the DVM uses registers to manage local variables rather than pushing them onto a stack. For example, in Dalvik, `add-int` v_1, v_2, v_3 adds the contents of registers v_2 and v_3 and stores the result to v_1. In contrast, Java bytecode would first push the integer variables onto the stack with `iload` 2 and `iload` 3, perform the addition with `iadd` and store the result using `istore` 1.

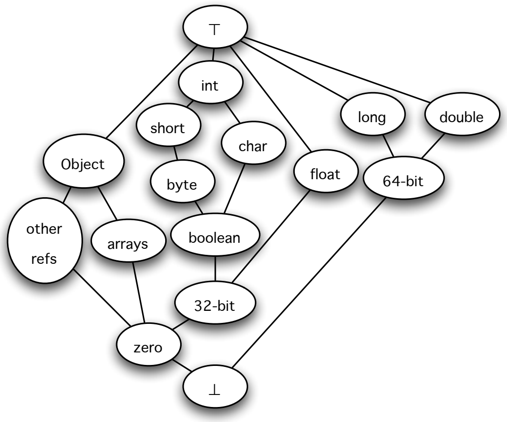


Figure 1: Dalvik Type Lattice

It is challenging and potentially cumbersome to find a semantic mapping for all 257 instructions. To allow structured mapping between the VMs, we introduce the Tyde intermediate representation (IR) in **Section 4**. The generation of Java bytecode from Tyde is shown in **Section 6**. In particular, we show how to make the transition from register-based bytecode to stack-based bytecode.

Exceptions - There is a significant difference in the type inference algorithm used by the verifiers, related to how they handle exceptions. During the path-sensitive type verification process, the Java verifier considers that any instruction in a `try` block may throw an exception. In reality, not all instructions in each try block are able to throw exceptions. Therefore, the Java verifier considers some unfeasible execution paths. On the other hand, the Dalvik verifier does not consider these unfeasible paths. Occasionally, an unfeasible path leads from a register assignment to a register use with an incompatible type (e.g., an int register assignment reaches a use with float type). It is not an issue in the DVM, since the spurious execution path is not considered by the verifier. However, since the Java verifier follows the unfeasible path during type inference, it leads to unverifiable Java bytecode if nothing is done to remove it. We explain how we deal with this issue in **Section 5.1**.

Bytecode Type System - DVM typing is very different than that of JVM bytecode. The primary differences include:

- *Primitive Assignments* - Dalvik primitive constant assignments specify only the width of the constant (32 or 64 bits). Thus, no distinction is made between int and float or between long and double. In contrast, primitive constants in Java are fully typed.
- *Array Load/Store Instructions* - The DVM has common array-specific load and store instructions for int and float arrays (`aget` and `aput`) and for long and double arrays (`aget-wide` and `aput-wide`). Here again, this introduces type ambiguity.
- *Object References* - Java bytecode uses the `null` reference type to track and detect undefined object references. Conversely, Dalvik uses an integer constant with value 0 to represent both the number zero and the `null` reference. Adding to this ambiguity, a comparison between two integers uses the same instructions as a comparison between object references.

Figure 1 shows the lattice of types in the Dalvik architecture. It depicts subtyping relations between types. We have collapsed array and other reference types. The zero, 32-bit and 64-bit types are not valid Java types. While our previous tool, `ded` [29, 31], managed to generate valid types in most cases using a simple type

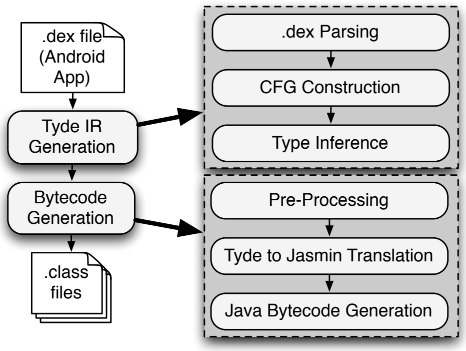


Figure 2: Verifiable Dalvik Bytecode Retargeting Overview

inference algorithm, a more sophisticated algorithm must be used to systematically generate type-safe Java bytecode. We describe the constraint-based algorithm we use for retargeting in **Section 5.2**.

Unverifiable Dalvik Bytecode - Occasionally, part of the Dalvik bytecode of applications found in the markets is unverifiable. As the DVM and JVM verification processes are similar, retargeting unverifiable Dalvik bytecode usually leads to unverifiable Java bytecode. Our goal is to generate verifiable bytecode on any input, therefore properly dealing with unverifiable input code is an important challenge. We address it in **Section 7**.

3. RETARGETING VERIFIABLE DALVIK BYTECODE: OVERVIEW

Figure 2 describes the `Dare` retargeting process for verifiable Dalvik bytecode. We address the issue of unverifiable Dalvik bytecode in Section 7. The application bytecode is initially translated into the Tyde intermediate representation (IR) in three steps: *a*) the `.dex` file is parsed and code structures, methods and the global constant pool are interpreted and annotated, *b*) a control flow graph is generated and *c*) register types used in ambiguous instructions are inferred. The Java bytecode is thereafter generated from this IR in three phases: *d*) a pre-processing step generates labels and maps registers to local variables, *e*) the IR is translated to Jasmin [23] code, and *f*) the Jasmin tool generates the final `.class` files.

To illustrate, Figure 3(a) shows the source code for a hypothetical method `m2` and Figure 3(b) shows Java bytecode generated by the Java compiler. The `iload_1` instruction loads local variable 1 (variable `a` in the source) onto the stack. The next instruction compares its value to 0. If it is not 0, then `dconst_1` loads double value 1.0 onto the stack and `dreturn` returns it. Otherwise, `ifeq` branches to offset 6. `ldc2_w` loads a constant with value 2.5 from the constant pool; the constant is then returned with `dreturn`.

Figure 3(c) shows the Dalvik bytecode for `m2`. `if-eqz` compares the value of register v_3 to 0. If it is not 0, then a 64-bit constant is assigned to register v_0 with `const-wide/high16` and returned with `return-wide`. If v_3 is 0, then the instruction at offset 5 is executed and assigns a different 64-bit constant to v_0. Next, `goto` transfers control to offset 4. The 64-bit constants are detected as long by default by the disassembler we used (dexdump), which is why they appear as long (instead of double) in Figure 3(c).

Figures 3(d), 3(e) and 3(f) show the stages of retargeting. The Tyde representation of `m2` is generated by mapping the Dalvik structures and generated control flow graph into the IR and performing type inference on the ambiguous register references. Once in Tyde, all registers are fully typed in accordance with the Java type system. Figure 3(f) shows the retargeted Java bytecode after remapping and Jasmin assembly. This bytecode is functionally equivalent to the

```
public double m2(int a)
{
  if (a != 0)
    return 1.0;
  else
    return 2.5;
}
```

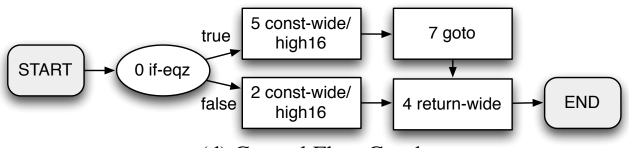
(a) Source Code

```
public double m2(int);
0: iload_1
1: ifeq 6
4: dconst_1
5: dreturn
6: ldc2_w #double 2.5d
9: dreturn
```

(b) Original Java Bytecode

```
public double m2(int);
0: if-eqz v3, 5 // +5
2: const-wide/high16 v0, #long 4607182...
4: return-wide v0
5: const-wide/high16 v0, #long 4612811...
7: goto 4 // -3
```

(c) Dalvik Bytecode



(d) Control Flow Graph

```
public double m2(int);
0: if-eqz (3, int, δs), 5
2: const-wide/high16 (0, double, δd) #double 1.0
4: return-wide (0, double, δs)
5: const-wide/high16 (0, double, δd) #double 2.5
7: goto 4
```

(e) Tyde Representation - δ_s (resp. δ_d) indicates a source (resp. destination) register.

```
public double m2(int);       8:  dload_2
0:  iload_1                  9:  dreturn
1:  ifeq 10                  10: ldc2_w #double 2.5d
4:  ldc2_w #double 1.0d      13: dstore_2
7:  dstore_2                 14: goto 8
```

(f) Retargeted Java Bytecode

Figure 3: Stages of Retargeting for Method `m2`

one in Figure 3(b), albeit longer. That is mostly due to the presence of spurious store/load instructions. However, we are not concerned with optimality but only with semantic equivalence. Tools such as Soot [37] can optimize the resulting bytecode if necessary.

4. THE TYDE REPRESENTATION

The DVM recognizes 257 different instructions. A naïve approach to converting Dalvik bytecode to Java bytecode would be to have 257 translation rules, which is very cumbersome. Moreover, analyzing the equivalence of the semantics of the translation rules would be very time consuming. The naïve approach would also make the implementation error-prone and hard to maintain.

In this section we describe a typed IR called *Tyde* (for *Ty*ped *de*x) whose main purpose is to enable easy translation of Dalvik bytecode to Java bytecode. As described in Section 6, translating the Tyde IR to Java bytecode is done *with only 9 translation rules* for all 257 Dalvik opcodes. The corresponding semantic mapping is much easier to analyze than 257 translation rules. Moreover, this approach also leads to a cleaner and maintainable implementation.

The insight behind Tyde is that, by typing all instruction arguments, load/store operations can be translated independently of opcodes. For instance, let us consider instructions `add-int` v_0, v_1, v_2 (integer addition) and `add-float` v_3, v_4, v_5 (float addition). By typing all registers and specifying if they are source or destination, we can use a single translation rule for both instructions: first translate all source register loads, then translate the opcode and finally translate the destination register store. If we did not determine the type information about the registers, retargeting those instructions would require two different translation rules.

Another advantage of Tyde is that Dalvik pseudo-instructions (such as `packed-switch-payload`) are not used, which leads

```
public double m3(int a) {
  switch (a) { case 0: return 1.0;
               case 1: return 2.5;
               default: return 4.0; }}
```

(a) Source Code

```
public double m3(int);
0:  packed-switch v3, 12 // +12
3:  const-wide/high16 v0, #long 4616189...
5:  return-wide v0
6:  const-wide/high16 v0, #long 4607182...
8:  goto 5 // -3
9:  const-wide/high16 v0, #long 4612811...
11: goto 5 // -6
12: packed-switch-payload
      entries: 2 - first key: 0 - targets: 6, 9
```

(b) Dalvik Bytecode

```
public double m3(int);
0:  packed-switch (3, int, $\delta_s$)
      first key: 0 - targets: 6, 9 - default: 3
3:  const-wide/high16 (0, double, $\delta_d$), #double 4.0
5:  return-wide (0, double, $\delta_s$)
6:  const-wide/high16 (0, double, $\delta_d$), #double 1.0
8:  goto 5 // -3
9:  const-wide/high16 (0, double, $\delta_d$), #double 2.5
11: goto 5 // -6
```

(c) Tyde Representation

Figure 4: Method `m3`

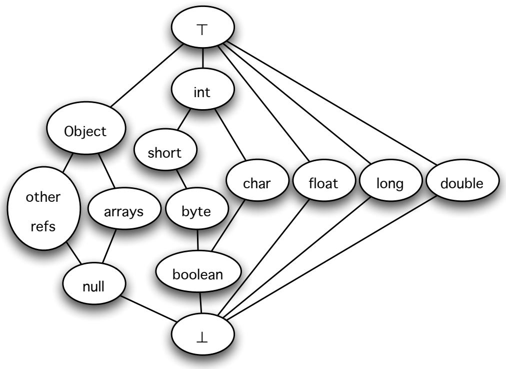


Figure 5: Tyde Type Lattice

to a more compact representation. Let us consider method `m3`, whose source code is presented in Figure 4(a). Figure 4(b) shows the corresponding Dalvik bytecode. It uses a `packed-switch-payload` pseudo-instruction which contains data about the switch statement at offset 0 (note that the default case is implicit). Moreover, in the Dalvik bytecode the registers used as arguments are not typed. Figure 4(c) shows the Tyde IR for `m3`. In the Tyde representation, the `packed-switch` instruction has all the necessary data (with an explicit default case) and no pseudo-instruction is used.

4.1 Specification

Figure 5 presents the type lattice used by the Tyde IR. All types on this lattice are valid Java types. In Tyde, we introduce the notion of typed registers. It adds two elements to Dalvik registers: a type τ and information about whether the register is a source or destination register (represented by terminals δ_s and δ_d).

The notion of typed value is used for all source or destination operands. Typed source values can either be typed source registers or integer literals. Typed destination values are defined as typed destination registers, ρ_p or ρ_{2p}. ρ_p (resp. ρ_{2p}) represents the case where a single-word (resp. double-word) return value is ignored after a method invocation. This is summarized in Figure 6, in which τ is any type from the lattice in Figure 5.

Tyde only defines proper instructions, i.e., no pseudo-instructions are used. Also, instead of constant pool indices and most inline numeric literals, Tyde directly uses Java constants. Note that while several constant types are used, we use a generic constant C for ease of exposition. Finally, Tyde does not use offsets to refer to other instructions to represent branches. Instead, Tyde instructions use pointers to other instructions, represented as $ptr_{\mathcal{T}}$.

Register Indices	$r ::= \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{\ldots}$
Typed Source Registers	$\rho_s ::= (r, \tau, \delta_s)$
Typed Dest. Registers	$\rho_d ::= (r, \tau, \delta_d)$
Typed Registers	$\rho ::= \rho_s \mid \rho_d$
Integer Literals	$l_i ::= \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{\ldots} \mid \mathbf{-1} \mid \mathbf{-2} \mid \mathbf{\ldots}$
Typed Source Values	$v_s ::= \rho_s \mid l_i$
Typed Destination Values	$v_d ::= \rho_d \mid \rho_p \mid \rho_{2p}$

Figure 6: Tyde Typed Registers and Values

Name	Syntax	Dalvik Instructions
$\mathcal{T}_{uo}$	$\mathcal{O}_{uo}, [v_d], \{v_s\}, [C]$	222
$\mathcal{T}_{ao}$	$\mathcal{O}_{ao}, [v_d], \{v_s\}, [C]$	12
$\mathcal{T}_{ub}$	$\mathcal{O}_{ub}, \{\rho_s\}_0^2, ptr_{\mathcal{T}}$	11
$\mathcal{T}_{ab}$	$\mathcal{O}_{ab}, \{\rho_s\}_1^2, ptr_{\mathcal{T}}$	4
$\mathcal{T}_{no}$	$\mathcal{O}_{no}, \rho_d, \rho_s$	2
$\mathcal{T}_{fna}$	$\mathcal{O}_{fna}, \rho_d, \{\rho_s\}, \tau$	3
$\mathcal{T}_{fad}$	`fill-array-data`, $\rho_s, \{C\}$	1
$\mathcal{T}_{ps}$	`packed-switch`, $\rho_s, l, ptr_{\mathcal{T}}, \{ptr_{\mathcal{T}}\}$	1
$\mathcal{T}_{ss}$	`sparse-switch`, $\rho_s, \{l\}_m, ptr_{\mathcal{T}}, \{ptr_{\mathcal{T}}\}_m$	1

Table 1: Simplified Syntax of Tyde Instructions.

Table 1 shows the syntax of Tyde instructions. There are 9 formats, each of which is later translated to Java using a single rule (see Section 6). $\{A\}_a^b$ indicates that symbol A is repeated between a and b times. We note $\{A\}_a = \{A\}_a^a$ and $[A] = \{A\}_0^1$. Finally, $\{A\}$ indicates that A is repeated zero or more times.

$\mathcal{T}_{uo}$ represents instructions which have zero or one typed destination value v_d, zero or more source values v_s, and zero or one Java constant C. Finally, their opcode has an unambiguous semantic equivalent in the JVM. Examples include a vast majority of unary and binary operators and method invocations. $\mathcal{T}_{ao}$ is almost the same format: the only difference is that the corresponding Java opcode is ambiguous. For example, `return-wide` is included in that format because it can be used to return a long (`lreturn` in Java) or a double (`dreturn` in Java). Opcode set $\mathcal{O}_{uo}$ is partially shown in the first column of Table 3(a); the complete lists for $\mathcal{O}_{uo}$ and other partially defined sets are available in our technical report [30]. Set $\mathcal{O}_{ao}$ is partially shown in Table 3(b) (first column).

$\mathcal{T}_{ub}$ represents branching instructions whose opcode have an unambiguous semantically equivalent Java opcode. In addition to an opcode, they are composed of zero, one or two typed source registers ρ_s and a pointer to a target Tyde instruction $ptr_{\mathcal{T}}$. $\mathcal{T}_{ab}$ is similar: the two differences are the number of source registers and the fact that the corresponding Java opcode is ambiguous. Sets $\mathcal{O}_{ub}$ and $\mathcal{O}_{ab}$ are partially shown in Tables 4(a) and 4(b) (first column).

$\mathcal{T}_{no}$ represents the `not-int` and `not-long` unary operators defined in the DVM, which do not have a trivial semantically equivalent opcode in the JVM. Set $\mathcal{O}_{no}$ is { `not-int`, `not-long` }.

$\mathcal{T}_{fna}$ instructions are the `filled-new-array` instructions. They are used to create a new array and fill it with the contents of registers. In addition to their opcode, they are composed of a destination register ρ_d, an arbitrary number of sources registers ρ_s and a type τ. Set $\mathcal{O}_{fna}$ is { `filled-new-array`, `filled-new-array/range`, `filled-new-array/jumbo` }. $\mathcal{T}_{fad}$ instructions are `fill-array-data` instructions (presented in Section 2). In addition to their opcode and a typed source register ρ_s, they have an arbitrary number of Java constants.

$\mathcal{T}_{ps}$ are the `packed-switch` instructions. They are composed of a `packed-switch` opcode, a typed source register ρ_s, an integer literal l_i (switch lowest case value) and a strictly positive number of pointers to Tyde instructions $ptr_{\mathcal{T}}$ (switch targets, including the default case handler). $\mathcal{T}_{ss}$ are the `sparse-switch` instruc-

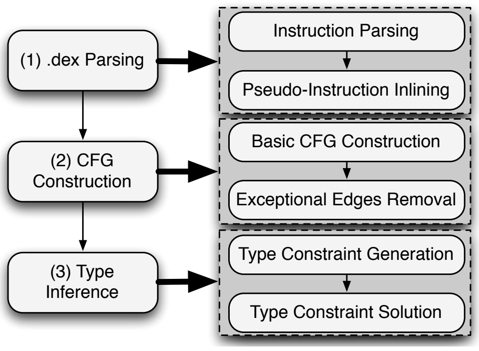


Figure 7: Tyde IR Construction Overview

tions. They are composed of a `sparse-switch` opcode, a typed source register ρ_s, m integer literals l_i (switch case values) and $m+1$ pointers to Tyde instructions $ptr_{\mathcal{T}}$ (switch targets, including the default case handler), for some integer m.

Note that with this representation, not all instructions correspond to valid instructions. For example, with an `add-int` instruction in $\mathcal{T}_{uo}$, there would only be two source values, even though the format accepts an arbitrary number of source values. The third column of Table 1 indicates how many types of Dalvik instructions are mapped to each Tyde instruction format.

5. FROM DALVIK BYTECODE TO TYDE

As depicted in Figure 7, Tyde IR is generated in three steps. The first step consists of parsing the `.dex` file, which involves parsing data related to classes and methods (e.g., access flags and names), as well as fields and method instructions. In the next step, a Control Flow Graph (CFG) is constructed. In the final step, a type inferencing algorithm infers types of registers that are ambiguous.

Parsing - Since parsing class and method data and fields is straightforward, we focus our description on instruction parsing. While parsing instructions, type information for registers is determined. For example, the types of several unary and binary operators can be known from their opcode, e.g., an `add-long` instruction takes two long integers as sources and a long integer as destination. Also, during this parsing step, for every instruction which uses a constant pool reference, a new Java constant is generated on the fly. The only exception is when the instruction is an ambiguous numeric constant assignment. In that case, type information is needed before the constant can be created. The parsing step also removes pseudo-instructions. As mentioned before, Dalvik bytecode uses pseudo-instructions to store complementary data about a proper instruction. In the parsing step, we simply store the contents of each pseudo-instruction in the proper instruction which refers to it and eliminate the pseudo-instruction.

5.1 Building a CFG

In the second step, we build a CFG from the Dalvik bytecode. Techniques used for constructing CFGs are quite standard, so we focus on details that are specific to our system. As required by the Tyde IR, relative offsets in branching instructions are converted to pointers. Also, exception tables are used to generate CFG edges. When an instruction I is protected by a `try` block, we add a CFG edge from its predecessors to the appropriate exception handlers. Standard CFG construction often also includes edges from I to exception handlers if I has side effects. This is done in order to account for the case in which the side effects are committed before the exception is thrown. For our purposes, we do not include these edges. They are not included by the Dalvik or Java verifiers: even though I might commit side effects before throwing an exception, the type state will not be changed. This entire process is quite straightforward, since all relative offsets are statically known.

5.1.1 Removing Unfeasible Exceptional CFG Edges

As we previously described, DVM and JVM verifiers differ in the way they handle CFG edges related to exceptions: while the Java verifier considers that all instructions inside a `try` block may throw an exception, the Dalvik verifier only considers edges for instructions which might actually throw an exception. For our CFG construction, we adopt the Dalvik approach. In order to generate verifiable Java bytecode, we also modify the exception tables (which describe `try/catch/finally` blocks) to only include instructions which might throw an exception in `try` blocks. We now explain why these modification do not modify the semantics of the program.

The Java platform has two types of exceptions [21, §2.16]. *Synchronous* exceptions occur as a result of the execution of a particular instruction. For example, if an integer division instruction has a divisor with value 0, it will throw an `ArithmeticException` when it is executed. On the other hand, *asynchronous* exceptions can happen at any point in the execution of a method. Asynchronous exceptions only have two possible causes. First, an asynchronous exception can be thrown when the deprecated `Thread.stop()` method is called (the thread on which it is called will immediately throw an exception). The only other case where an asynchronous exception can be thrown is when an internal error in the virtual machine implementation occurs.

Removing CFG edges for instructions which cannot throw synchronous exceptions does not violate synchronous exception handling semantics; as we know which instructions can throw synchronous exceptions, we just have to make sure to leave them in their `try` blocks. Asynchronous exceptions may seem less trivial, since they might in theory be thrown by any instruction. However, exceptions cannot be thrown asynchronously in the DVM (in particular, `Thread.stop()` is not supported). That is the reason why the Dalvik verifier only considers synchronous exceptions in its CFG construction. Therefore, removing CFG edges corresponding to asynchronous exception handling does not modify the semantics of the program.

5.2 Type Inference

The problem we are trying to solve is the following: given a method with typing following the lattice on Figure 1, find a typing which is valid with respect to the lattice on Figure 5.

We defer formal proofs to future work, but the type inference algorithm we present below was able to find a valid Java typing for all 1.6 million retargeted methods from our sample. There are several reasons why our algorithm works well in practice. The type system enforced by the Dalvik verifier is quite similar to the type system enforced by the Java verifier. For example, consider a register assignment which might be for a float or an int. If it is used as a float, it cannot be subsequently used as an int on the same branch. However, the Dalvik verifier does not back-propagate unambiguous use type information to ambiguous assignments. Therefore, if an ambiguous assignment (e.g., 32-bit constant) reaches two uses with incompatible types through two different branches (e.g., int on one branch and float on another branch), the code could still be Dalvik-verifiable even though it does not have a valid Java typing. However, we did not find any occurrence of this, most likely because Dalvik code is compiled from Java code and the analysis required to merge registers with different types but identical bit patterns would be costly and provide very minimal gain.

Type inference for Dalvik bytecode uses the following approach: First we generate constraints on types based on definitions and uses. These constraints are then solved to infer unknown types. Note that our goal is not to determine types for all variables, unlike previous work [5]. In particular, with the exception of array types we do not need to know precise types for references.

In this section, $\tau_c(v_i, I_j)$ and $\tau_v(v_i, I_j)$ denotes the type of register v_i at instruction I_j. We use τ_c to denote a type constant and τ_v to denote a type variable.

5.2.1 Constraint Generation

Constraints are generated by traversing the CFG starting at ambiguous register definitions (respectively uses), looking for uses (respectively assignments) of the same register. For example, let us consider method m2, defined in Figure 3. Its CFG is shown in Figure 3(d). Method m2 generates the following constraints: $\tau_v(v_0, I_2) \leq$ double, $\tau_v(v_0, I_5) \leq$ double, and int $\leq \tau_v(v_3, I_0)$. It has two ambiguous definitions (the two `const-wide/high16` instructions) and one ambiguous use (the `if-eqz` instruction). The first two constraints are generated because ambiguous definitions reach instruction `return-wide`, whose type is known from the method signature. The last constraint (on instruction I_0) is given by the method signature, according to which register v_3 is assigned an integer argument before method execution starts.

The constraints related to variable definitions and uses induce inequality constraints. Instructions dealing with arrays introduce another kind of constraint on their operands. For example, consider instruction aget v_0, v_1, v_2, which loads element at index v_2 from the array referenced by v_1 into v_0. If it is determined that the array referenced by v_1 is an array of integers, then v_0 is an integer. Eventually, we obtain four types of constraints: type (1) $\tau_c \leq \tau_v$, type (2) $\tau_v \leq \tau_c$, type (3) $\tau_{v_1} \leq \tau_{v_2}$ and type (4) $\tau_{v_1} = [\tau_{v_2}$ (where $[\tau_{v_2}$ means "array of τ_{v_2}").

5.2.2 Constraint Solution

Solving these constraints is performed in three phases. Throughout these three phases, whenever a type variable involved in a type (4) constraint is determined, the other side of the type (4) equality is also determined (or checked if it is already known).

Phase 1 - First, we use Algorithm D by Rehof and Mogensen [36] to find the least solution to type (1) and (3) constraints. The Rehof-Mogensen (RM) algorithm finds the least solution τ. The RM-algorithm also checks that the solution τ satisfies type (2) constraints. For the constraint system shown earlier, the RM-algorithm finds the solution $\tau_v(v_3, I_0) =$ int. However, we still need to find values for $\tau_v(v_0, I_2)$ and $\tau_v(v_0, I_5)$, which is handled in phase 2.

Phase 2 - After finishing phase 1, there are variables whose type is $\perp$ (unknown). We need to infer the types of these variables. In our example, after phase 1 $\tau_v(v_0, I_2)$ and $\tau_v(v_0, I_5)$ are $\perp$. Phase 2 finds the types of these variables. Assume that type variable τ_v has value $\perp$ after phase 1, but has the following constraints of type (2): $\tau_v \leq \tau_{c_1}, \tau_v \leq \tau_{c_2}, \cdots, \tau_v \leq \tau_{c_k}$. Then we set $\tau_v = \tau_{c_1} \wedge \tau_{c_2} \wedge \cdots \wedge \tau_{c_k}$ (that is, the greatest common subtype of $\tau_{c_1}, \tau_{c_2}, \cdots, \tau_{c_k}$). This ensures that all type (2) inequalities involving τ_v are satisfied. For our example, this yields the following types: $\tau_v(v_0, I_2) =$ double and $\tau_v(v_0, I_5) =$ double. In general, this process may determine the type of the left-hand side of some type (3) inequalities. In order to solve these constraints properly, we run the algorithm by Rehof and Mogensen again.

Phase 3 - After phase 1 and 2 there are some types that are still undetermined (e.g., ambiguous assignment reaching only an ambiguous use, which is not itself reached by any unambiguous assignment). In that case, we set these types to safe default types. In Section 2, we explained that all instructions with ambiguous typing have a limited set of types that they can possibly take. For example, a `const-wide` instruction can only take types long or double. After this, all variable types are safe Java types.

Typed Value	Java Instruction
l_i	sipush l_i
(r, τ_i, δ_s)	iload $m(r)$
$(r$, float, $\delta_s)$	fload $m(r)$
$(r$, long, $\delta_s)$	lload $m(r)$
$(r$, double, $\delta_s)$	dload $m(r)$
(r, τ_r, δ_s)	aload $m(r)$

Typed Value	Java Instruction
ρ_p	pop
ρ_{2p}	pop2
(r, τ_i, δ_d)	istore $m(r)$
$(r$, float, $\delta_d)$	fstore $m(r)$
$(r$, long, $\delta_d)$	lstore $m(r)$
$(r$, double, $\delta_d)$	dstore $m(r)$
(r, τ_r, δ_d)	astore $m(r)$

Table 2: Opcode Map f_{ds} for Typed Values.

6. GENERATING JAVA BYTECODE

A Dalvik code in Tyde IR is translated into Java bytecode in three steps. In the first step, registers are mapped to Java local variables and labels are generated to support control-flow instructions. In the second step, instructions in Tyde IR are converted to Jasmin instructions (Jasmin is a Java bytecode assembler). The third step is to use Jasmin to generate Java `.class` bytecode. We will provide a brief description of the first two steps.

6.1 First Step (Pre-Processing)

Register Mapping - Tyde is a register-based representation, therefore we need to map every register to a Java local variable. A *register map m* is a function that maps a Tyde IR register to a Java local variable. In the JVM, the first local variable indices are reserved for the `this` reference (for non-static methods) and the method arguments [21, §3.6.1]. In the DVM, these variables use the last register indices, so our register map has to respect this constraint.

Label Generation - In Java bytecode, branching instructions include relative offsets to their targets. Also, exceptions tables describe the boundaries of `try/catch/finally` blocks using absolute offsets. However, Jasmin uses labels for these, which is why we need to generate labels before we can generate Jasmin code. We define b such that, for any pointer $ptr_{\mathcal{T}}$ to a Tyde instruction, $b(ptr_{\mathcal{T}})$ is the label of the instruction corresponding to $ptr_{\mathcal{T}}$.

6.2 Second Step (Translating Instructions)

In order to map a Tyde method to a Java method, we introduce helper functions. The general idea is that each Tyde instruction format is mapped to Java instructions using a single pattern. In this section, we use the symbol $\mathcal{J}_\epsilon$ to represent the "null" Java instruction (which in case of Jasmin is the empty string).

Typed Values - In Table 2, we define a function f_{ds} which maps typed values to Java instructions. For this definition, we use the register map m. τ_i is any single-word integer type (boolean, char, byte, short or int) and τ_r is any reference (array or non-array) type. Essentially $f_{ds}(v)$ corresponds to the Java instruction to load or store the Tyde typed value v. These Java instructions can be divided into two categories: *a)* pushing values onto the operand stack before an operation is applied to them (`load` instructions), and *b)* popping values resulting from an operation from the stack, either to store them in a local variable (`store` instructions) or to balance the stack if a method's return value is discarded (`pop` instructions). Since these values are typed and their register use (source or destination) is known, they can be translated independently from other parts of the Tyde instruction, i.e., knowing the Dalvik opcode is not necessary. One of the advantages of having types on sources and destinations in Tyde is that Java instructions to load/store/pop typed values can be generated independently of the instruction opcode.

Tyde Opcode	Java Opcode
nop	nop
move	$\mathcal{J}_\epsilon$
const-wide/16	ldc2_w
monitor-exit	monitorexit

(a) f_{uo} (partial definition)

Tyde Op.	Tyde Type	Java Op.
return	τ_i	ireturn
	float	freturn
const/4,	τ_i, float	ldc
const	τ_r	aconst_null

(b) f_{ao} (partial definition)

Table 3: Opcode Maps f_{uo} and f_{ao} for Sets $\mathcal{O}_{uo}$ and $\mathcal{O}_{ao}$.

Tyde Op.	Java Op.
goto	goto
if-lt	if-icmplt
if-ge	if-icmpge
if-ltz	iflt

(a) f_{ub} (partial definition)

Tyde Op.	Tyde Type	Java Op.
if-eq	τ_i	if_icmpeq
	τ_r	if_acmpeq
if-ne	τ_i	if_icmpne
	τ_r	if_acmpne

(a) f_{ab} (partial definition)

Table 4: Opcode Maps f_{ub} and f_{ab} for Sets $\mathcal{O}_{ub}$ and $\mathcal{O}_{ab}$.

Constants - We use $f_{\mathcal{C}}$ to represent the translation of a constant pool reference. If we were generating binary bytecode, it would simply be the index of the constant in the constant pool. Since we generate instructions for Jasmin, it is a textual description of the constant (e.g., value for an integer constant). If there is no constant, $f_{\mathcal{C}}$ simply returns an empty string.

Opcodes - In general, unambiguous opcodes in Tyde IR have a corresponding opcode in Jasmin. For ambiguous opcodes, the types of the operands in Tyde IR are used to determine the corresponding opcode in Jasmin. In Table 3(a), we define a function f_{uo} which maps unambiguous Tyde operators to semantically equivalent Java opcodes. A vast majority of Dalvik opcodes (222 out of 257) are in this class $\mathcal{O}_{uo}$. Their semantic mapping to a Java opcode is trivial and the equivalent Java opcode can be known by only knowing the Dalvik opcode. Table 3 only shows a subset of the mappings. The complete definition is available in the technical report [30], along with complete definitions of other partially defined functions. In Table 3(b), we define a function f_{ao} which maps ambiguous opcodes and Tyde types to Java opcodes. τ_p is any primitive type, i.e. τ_i, float, long or double. In these cases, one Dalvik opcode maps to several Java opcodes and thus an argument type is needed to disambiguate the mapping.

In Table 4(a), we define a function f_{ub} which maps unambiguous branching opcodes to semantically equivalent Java opcodes. As with f_{uo}, this mapping is trivial and the equivalent Java opcode is completely determined by the Dalvik opcode. Table 4(b) defines a function f_{ab} which maps ambiguous branching opcodes and Tyde types to Java opcodes. As with $\mathcal{O}_{ao}$, one Dalvik opcode maps to several Java opcodes and an argument type is needed to disambiguate the mapping.

Instructions - Using the register map m and the various functions defined earlier, we can describe the translation for each of the Tyde instruction classes shown earlier. This translation is shown in Table 5. In addition to the functions defined earlier, our translation also uses function $f_{xastore}$ for mapping store instructions defined in Table 6. Due to space restrictions, we will only describe translation rules for a few interesting Tyde instructions.

`not` **Instructions** $\mathcal{T}_{no}$ - The Dalvik and Tyde instruction sets include `not` instructions for integers. While there is no trivial semantic equivalent in the Java instruction set, we can use a combination of Java instructions and take advantage of the equivalence of bitwise binary operators. Given an integer i, we define $1_{|i|}$ the integer whose bit pattern is all ones with the same width as i (32 or 64 bits). If the NOT (resp. XOR) bitwise operator is represented as $\neg$ (resp. $\oplus$), then we have $\neg i = 1_{|i|} \oplus i$. Therefore, a valid Java instruction pattern consists in pushing constant $1_{|i|}$ onto the stack (with `ldc` or `ldc2_w` depending on integer width). After pushing i onto the stack, the `ixor` (or `lxor`) opcode should be applied. Finally, the result should be popped from the stack. This pattern defines map j_{no} in Table 5. For this function, we use function f_{no} defined over $\mathcal{O}_{no}$ such that $f_{no}(\texttt{not-int}) = \texttt{ixor}$ and $f_{no}(\texttt{not-long}) = \texttt{lxor}$. We also define $|o_{no}| = 32$ if $o_{no} = \texttt{not-int}$ and $|o_{no}| = 64$ if $o_{no} = \texttt{not-long}$.

Tyde Type	Java Opcode
[boolean, [byte	bastore
[char	castore
[short	sastore
[int	iastore
[float	fastore
[τ_r	aastore

Table 6: Map $f_{xastore}$.

`filled-new-array` **Instructions** $\mathcal{T}_{fna}$ - Dalvik and Tyde bytecode have instructions which create a new array and fill it with the contents of registers given as arguments. While Java does not have a direct equivalent, a semantically equivalent sequence of Java instructions is as follows. First, a `newarray` (primitive type) or `anewarray` (reference type) instruction will create a new array with the appropriate type and return a reference to the array on the stack. To fill the array with the proper values, we use a sequence of the following pattern: *i)* a `dup` instruction duplicates the array reference on the stack, then *ii)* the proper array index $l \in \mathcal{L}_i$ is pushed onto the stack, next *iii)* the array element value is pushed onto the stack, then *d)* an appropriate `xastore` instruction pops the duplicated reference, the index and the element and performs the array storage. Finally, an `astore` instruction stores the array reference to a local variable. This patterns defines map j_{fna} in Table 5. For this function, we use function f_{fna} such that $f_{fna}(\tau_p) = \texttt{newarray}$ and $f_{fna}(\tau_r) = \texttt{anewarray}$.

`packed-switch` **Instructions** $\mathcal{T}_{ps}$ - The semantic equivalent of a Tyde `packed-switch` instruction is a Jasmin `tableswitch` instruction. The arguments of a Tyde `packed-switch` are a typed source register (integer used to switch), a literal corresponding to the lowest case value, a pointer to the default case handler and a set of pointers to case handlers. The corresponding Jasmin instruction is similar, except that it uses labels instead of pointers.

Example - Let us consider method `m2` introduced in Figure 3(a). Figure 3(e) shows its Tyde representation. For ease of exposition, we assume that the label of each instruction is simply its original offset and the register mapping m is such that $m(0) = 2$, $m(1) = 3$, $m(2) = 0$ and $m(3) = 1$. The first Tyde instruction has format $\mathcal{T}_{ab}$ (see Table 5). Using the notation from Table 5, o_{ab} is `if-eqz`, $\rho_s^0 = (3, \text{int}, \delta_s)$, ρ_s^1 is empty and $ptr_{\mathcal{T}}$ is a pointer to instruction at offset 5. $f_{ds}(3, \text{int}, \delta_s) = \texttt{iload}\ m(3) = \texttt{iload}\ 1$ is the first Java instruction. Then we have $f_{ds}(\rho_s^1) = \mathcal{J}_\epsilon$, $f_{ab}(o_{ab}, \text{int}) = \texttt{ifeq}$ and $b(ptr_{\mathcal{T}}) = 5$ (note that the 5 value is just a label and not the offset in the final Java bytecode). The second Java instruction is `ifeq`, label 5. The remaining Tyde instructions are translated in a similar manner to obtain the bytecode shown on Figure 3(f).

7. UNVERIFIABLE DALVIK BYTECODE

In the previous sections, we have described how to generate verifiable Java bytecode from verifiable Dalvik bytecode. As we explore below, some bytecode from real-world applications is not Dalvik-verifiable. In this section, we show the errors we encountered with real bytecode. We also show how we modified our retargeting process to handle bytecode that is not Dalvik-verifiable in order to generate verifiable Java bytecode.

Tyde Instruction Set	Tyde Instruction	Equivalent Java Instructions
$\mathcal{T}_{uo}$	$t_{uo} = (o_{uo}, v_d, v_s^0, v_s^1, \cdots, v_s^k, C)$	$j_{uo}(t_{uo}) = f_{ds}(v_s^0) \Vert f_{ds}(v_s^1) \Vert \cdots \Vert f_{ds}(v_s^k) \Vert (f_{uo}(o_{uo}), f_{\mathcal{C}}(C)) \Vert f_{ds}(v_d)$
$\mathcal{T}_{ao}$	$t_{ao} = (o_{ao}, v_d, v_s^0, v_s^1, \cdots, v_s^k, C)$	$j_{ao}(t_{ao}) = f_{ds}(v_s^0) \Vert f_{ds}(v_s^1) \Vert \cdots \Vert f_{ds}(v_s^k) \Vert (f_{ao}(o_{ao}, \tau), f_{\mathcal{C}}(C)) \Vert f_{ds}(v_d)$
$\mathcal{T}_{ub}$	$t_{ub} = (o_{ub}, \rho_s^0, \rho_s^1, ptr_{\mathcal{T}})$	$j_{ub}(t_{ub}) = f_{ds}(\rho_s^0) \Vert f_{ds}(\rho_s^1) \Vert (f_{ub}(o_{ub}), b(ptr_{\mathcal{T}}))$
$\mathcal{T}_{ab}$	$t_{ab} = (o_{ab}, \rho_s^0, \rho_s^1, ptr_{\mathcal{T}})$	$j_{ab}(t_{ab}) = f_{ds}(\rho_s^0) \Vert f_{ds}(\rho_s^1) \Vert (f_{ab}(o_{ab}, \tau), b(ptr_{\mathcal{T}}))$
$\mathcal{T}_{no}$	$t_{no} = (o_{no}, \rho_d, \rho_s)$	$j_{no}(t_{no}) = (\text{ldc/ldc2\_w}, f_{\mathcal{C}}(1_{\lvert o_{no}\rvert})) \Vert f_{ds}(\rho_s) \Vert f_{no}(o_{no}) \Vert f_{ds}(\rho_d)$
$\mathcal{T}_{fna}$	$t_{fna} = (o_{fna}, \rho_d, \rho_s^0, \rho_s^1, \cdots, , \rho_s^k, \tau)$	$j_{fna}(t_{fna}) = (f_{fna}(\tau), \tau) \Vert \text{dup} \Vert f_{ds}(0) \Vert f_{ds}(\rho_s^0) \Vert f_{xastore}(\tau) \Vert \text{dup} \Vert f_{ds}(1) \Vert f_{ds}(\rho_s^1) \Vert f_{xastore}(\tau) \Vert \cdots \Vert \text{dup} \Vert f_{ds}(k) \Vert f_{ds}(\rho_s^k) \Vert f_{xastore}(\tau) \Vert f_{ds}(\rho_d)$
$\mathcal{T}_{fad}$	$t_{fad} = (\text{fill-array-data}, \rho_s, C_0, C_1, \cdots, C_k)$	$j_{fad}(t_{fad}) = f_{ds}(\rho_s) \Vert f_{ds}(0) \Vert (\text{ldc/ldc2\_w}, f_{\mathcal{C}}(C_0)) \Vert f_{xastore}(\tau(\rho_s)) \Vert f_{ds}(\rho_s) \Vert f_{ds}(1) \Vert (\text{ldc/ldc2\_w}, f_{\mathcal{C}}(C_1)) \Vert f_{xastore}(\tau(\rho_s)) \Vert \cdots \Vert f_{ds}(\rho_s) \Vert f_{ds}(k) \Vert (\text{ldc/ldc2\_w}, f_{\mathcal{C}}(C_k)) \Vert f_{xastore}(\tau(\rho_s))$
$\mathcal{T}_{ps}$	$t_{ps} = (\text{packed-switch}, \rho_s, l, ptr_{\mathcal{T}}^{default}, ptr_{\mathcal{T}}^0, ptr_{\mathcal{T}}^1, \cdots, ptr_{\mathcal{T}}^k)$	$j_{ps}(t_{ps}) = f_{ds}(\rho_s) \Vert (\text{tableswitch}, l, b(ptr_{\mathcal{T}}^{default}), b(ptr_{\mathcal{T}}^0), b(ptr_{\mathcal{T}}^1), \cdots, b(ptr_{\mathcal{T}}^k))$
$\mathcal{T}_{ss}$	$t_{ss} = (\text{sparse-switch}, \rho_s, l_1, l_2, \cdots, l_k, ptr_{\mathcal{T}}^{default}, ptr_{\mathcal{T}}^1, ptr_{\mathcal{T}}^2, \cdots, ptr_{\mathcal{T}}^k)$	$j_{ss}(t_{ss}) = f_{ds}(\rho_s) \Vert (\text{lookupswitch}, b(ptr_{\mathcal{T}}^{default}), l_1, b(ptr_{\mathcal{T}}^1), l_2, b(ptr_{\mathcal{T}}^2), \cdots, l_k, b(ptr_{\mathcal{T}}^k))$

Table 5: Tyde Maps.

7.1 Observed Errors

Improper references - The main source of unverifiable bytecode is the presence of bad method, field, interface or class references. Two different cases were encountered:

- References to classes which are not available within the application or in the core Android classes. A special case is when the superclass of a class is missing; then the class is trivially unverifiable and is not even linked by the DVM.
- References to methods or fields which are non-existent or not accessible (e.g., private member).

There are two reasons for these missing references. The first reason is that applications commonly use private Android APIs. The Android platform includes public APIs which are documented and always backward compatible (an application using only public APIs will still work after an OS update). Android also includes private APIs which are meant for internal use by OS components. Unlike public APIs, private ones are not officially documented and backward compatibility is not guaranteed. Using them is strongly discouraged, as a simple OS update may break an application making calls to private APIs. In our experiments, we checked verifiability using recent Android core classes whereas the application sample was about a year older. Doing so allowed us to point out potential problems applications could have after an OS update.

The second reason is that, as we pointed out in previous work [9], developers often include entire libraries to be able to use some classes from these libraries. Parts of the included libraries sometimes make calls to other libraries, which are not themselves included with the Android application. In practice, it is not an issue, as long as these parts of the included library are not used anywhere in the application. However, the unused part of the library code making these calls will not be verifiable.

Typing and other issues - The second source of unverifiability for Dalvik bytecode is the presence of invalid typing. Other issues are a marginal cause of verification problems. These can have a very wide variety of causes (e.g., malformed class or member identifier, illegal access flag, etc.). More analysis is needed to understand how these problems can make their way to released Dalvik bytecode.

7.2 Handling Unverifiable Dalvik Bytecode

In this section, we describe our approach to handle unverifiable Dalvik bytecode in an application using Dalvik pre-verification. First, we verify the application using the Dalvik verifier which is part of the Android OS. We modified it in order to make it generate a detailed report describing all verifiability issues with the application. For each problem in the application, the report describes the class name, the method name and signature, the problematic code offset and the type of verification error.

Then, the report is input into `Dare` with the application package. The `Dare` parsing step (see Section 5) is modified as follows:

1. If the entire class could not be linked by the verifier (e.g., because its superclass is missing), then skip the entire class. None of the code of the class will be retargeted. At runtime, the class would also be missing, so not retargeting it does not change the semantics of the program.
2. If a method is entirely unverifiable because of a serious issue (e.g., typing issue), then the parsing step replaces the method with code that throws a `VerifyError`. All other verifiable methods in the class will be retargeted without modification.
3. If a method is unverifiable because of a less serious issue (e.g., missing class reference), then only replace the faulty code location with code that throws an appropriate error (for example, `NoClassDefFoundError`). That is exactly the behavior of the Dalvik VM at runtime: the faulty code locations are also rewritten during the verification process. An interesting case is when we encounter a reference to a class that was ignored in Step 1. The reference is then replaced with code throwing a `NoClassDefFoundError`, which is the runtime behavior.

These code transformations serve two important purposes:

- They mirror the runtime behavior of the program and therefore do not cause any loss of semantics.
- They make the entire program Dalvik-verifiable.

In order to account for some subtle differences between the Java and Dalvik verifiers, we also had to consider two cases where a method was Dalvik-verifiable but it was not Java-verifiable after retargeting. The first difference involves `aget-object` instructions, which are used to access a component in an array of references. If it is used with an array reference which is known to always be null at verification time, then the verifier sets the array component to be null as well. But as we described above, in Dalvik, null and int/float with value 0 are the same type. Thus, if the register is subsequently used as an int or a float, the code will be Dalvik-verifiable, but no valid Java typing will exist for it. In order to fix this, we replace the `aget-object` instruction with code throwing a `NullPointerException`. This mirrors the runtime behavior of the program and also allows us to find a Java typing (since the null register will no longer be used as an int or a float).

Second, when a field with reference type is accessed in the Dalvik architecture, the method accessing it can still be verifiable even if the field type cannot be resolved (the type of the register storing the field is set to `java.lang.Object`). The method is verifiable only if the register is subsequently used in trivial ways. On

	Bad Ref.	Missing Ref.	Typing Issue	Other Issue
Apps	93	168	73	13
Code Locations	1,335	6,413	488	54

Table 7: Verification results for partially verifiable classes

the other hand, the Java verifier will reject a method if a field reference type cannot be resolved. In order to make the retargeted code verifiable, we had Dare automatically generate class stubs for the unresolvable field types. Since those fields are only used in trivial ways, these stub classes do not need to contain any field or method.

8. EVALUATION

8.1 Dalvik Bytecode Verification

In this section, we describe the Dalvik verification issues we found in a sample of 1,100 applications. We use the 50 most popular applications in the 22 application categories as of September 1, 2010[2]. The 1,100 applications contained 262,110 classes. We used the Dalvik verifier and the core Android classes included in Android version 4.0.3.0.2.0.1.0.

A surprising result of our Dalvik verification experiments is that 247 applications contained unverifiable code. 181 applications contained at least one class which could not be linked by the Dalvik VM (e.g., because of a missing superclass), totalling 905 trivially unverifiable classes. 214 applications had at least one unverifiable code location in non-trivially unverifiable classes. Table 7 presents the results of the Dalvik verification process for the classes which were not completely Dalvik-unverifiable. For each issue, we show the number of faulty code locations and the number of applications with at least one bad location. Occasionally, several locations in a single method can be unverifiable.

The first column shows the number of bad references, i.e. references to an inaccessible (e.g., private) or nonexistent class member. It is unlikely for an application to make a bad reference to its own code, so these are most likely references to private Android APIs which were modified after the application was developed. The second column shows the number of references to a missing class. These are caused by unlinkable application classes and by (supposedly unused) code in included libraries making references to other libraries which are not included with the application. Finally, we show the number of typing and other issues, which account for 6.46% of all unverifiable code.

8.2 Retargeting

The empirical evaluation described in this section attempts to answer two central questions: 1) are the computational costs of retargeting feasible in practice? and 2) can Dare successfully retarget market applications? The answers to these questions will determine the degree to which this is a useful tool for extracting code for further analysis. Highlights of the study include:

- After some additional code optimizations in some isolated cases, the output of Dare is verifiable for all methods for 99.64% of the applications in the corpus, and over 99.999% of methods overall. This is a substantial increase over existing tools.
- Dare can, on average, retarget each class in 4.20 msec, and was able to retarget the entire corpus of 1,100 applications containing over 260,000 classes in less than 20 min.
- The complete processing of all applications including Dalvik pre-verification (modified Dalvik verifier), retargeting (Dare) and assembly (Jasmin) took less than 70 compute-minutes.

We compare Dare against dex2jar [1], the most popular tool for retargeting Dalvik bytecode to Java. We do not report bytecode verification results for our previous tool: ded [29, 31] was built for the purpose of decompilation and did not include all information that is required for a verifiable class file, for example the maximum stack size for each method (which was set to a default value of 0). As a consequence, while the output of ded can typically be processed by decompilers and accurately captures the semantics of the original program, it is generally trivially unverifiable.

[2]Experiments run on a smaller sample from March 2012 show near-identical results.

	Total	Removed Code	Modified Code	Unverif. Classes	Unverif. Code	Verifiable Code (%)
Apps	1,100	181	214	7	3	99.09%
Classes	262,110	905	3,354	14	4	99.99%
Methods	1,620,813	9,658	6,858	100	4	99.99%

Table 8: Dare retargeting success rates

We evaluate Dare on two keys metrics: performance and retargeting success rate. We retargeted the entire corpus of applications described in the previous section.We used the latest version of Jasmin (2.4.0) for Java bytecode assembly.

Performance - The total processing time was 4,198 seconds, with Dalvik pre-verification consuming 229 seconds (5.45%), Dare 1,101 seconds (26.23%) and Jasmin 2,868 seconds (68.32%). Dare processing was dominated by the file output operations. They are performed at the same time as the translation of Tyde to Jasmin. Together they take 85% of the total processing time. The type inference algorithm accounts for 5% of the total processing time. Other parts of the retargeting process take less than 5% each. Retargeting is efficient and can be a fast first step before application analysis.

Retargeting - The success metrics reported below measure the ability of Dare to generate valid bytecode. A method is said to be successfully retargeted when Dare generates bytecode that is verifiable (and thereafter is ready for inspection and analysis by existing tools). A class is said to be successfully retargeted if all the methods it contains are successfully retargeted. Finally, an application is said to be successfully retargeted if all classes within the application are retargeted. For the Java bytecode verification experiments, we used the Oracle Labs Maxine VM verifier [33].

Table 8 shows the resuts of the Dare retargeting. The first column shows the total number of classes and non-abstract methods in our sample. The second column shows how many classes were safely removed using the Dalvik verification reports as described in Section 7 (the application count is the number of apps in which at least one class was removed). The next column presents the number of classes and methods which were modified following the Dalvik verification reports. The next column shows the number of retargeted classes which were completely unverifiable (the application count is the number of apps in which at least one class was completely unverifiable). For each of these classes, the issue was caused by a single method which had a code size over the maximum allowed size of 65536 bytes. We were able to fix 13 of these 14 issues by running the Soot optimizations on these classes. Only one of these 14 failures could not be fixed: the bytecode optimizations did not sufficiently reduce the code size. After optimization, only one of the 100 methods was not verifiable.

The next column shows the number of methods which had an issue that did not cause Maxine to reject the entire class but only a single method. One of these failures was caused by a reference to one of the classes which was not verifiable because of code size, as described above. It was fixed after the code optimizations reduced the code size and made the referenced class verifiable. The other 3 failures were related to a pathological difference between the Dalvik and Java verifiers. In the case of Java, when a method in a class C tries to access a protected method from a superclass D which is in a different package, it can only do so if the instance on which the method invocation occurs is an instance of a subclass

	Total	Completely Unverif. Classes	Unverif. Code	Verifiable Code (%)
Apps	1,100	422	206	59.64%
Classes	262,110	1,405	776	99.17%
Methods	1,620,813	25,972	1272	98.32%

Table 9: `dex2jar` **retargeting success rates**

of `C`. The Dalvik verifier, however, does not enforce this rule and only checks that `C` is a subclass of `D`. As a consequence, the Dalvik verification step accepted the 3 methods, which were subsequently rejected by the Java verifier after retargeting. Since a Java compiler would not generate code with this issue, the issue is most likely due to a private API method which was public when the applications were created and was later changed to be protected (all 3 failures occurred in a wrapper for the same Android API class and involve a call to the same protected method).

The final column shows the overall success rates as a percentage of the retargeted code. While we do not implement a solution for the 4 failures which did not have a trivial fix, we do not consider them to be significant. They only represent less than 0.00025% of the methods in our sample. Moreover, we were able to check that the issues with the code in these 4 methods do not necessarily prevent them from being processed by analysis tools: all 4 were successfully optimized by Soot. Typing problems, on the other hand, would prevent any serious analysis. No type issue was found by the Java verifier, which strongly validates our type inference algorithm.

Table 9 shows the retargeting results using the latest version of `dex2jar` (0.0.9.8), currently the most widely used retargeting tool. There are two main reasons why `dex2jar` performs less well at the retargeting experiments than `Dare` . First, it does not handle unverifiable Dalvik bytecode: the result of retargeting unverifiable Dalvik bytecode is unverifiable Java bytecode. The second reason is that, similarly to `ded`, `dex2jar` aims to be used for decompilation and typically decompilers can decompile unverifiable code if the cause for unverifiability is not too serious. Verifiability is a stronger criterion for success and ensures that the application is processed by analysis tools (and not only decompilers). In the case of `dex2jar`, a number of classes are completely unverifiable for trivial reasons (e.g., illegal member access flags). In addition, several methods are unverifiable for various reasons ranging from bad references to illegal typing. As a result, even though the class and method retargeting success rates are high (respectively over 99% and over 98%), less than 60% of applications are completely verifiable. It is a serious obstacle to whole-program analysis.

9. RELATED WORK

Java decompilers have been around for almost as long as the language itself. Krakatoa [34] was introduced in 1997 and Mocha [38] shortly thereafter. These earlier tools failed on a number of common bytecode structures. Dava [25, 24, 28] significantly improved the state of the art in Java decompilation in 2001. Dava is built upon the Soot framework [37]. Other tools such as Jad [2] and JD [7] are available but their algorithms are not published. Decompilation of other languages such as C/C++ [6] differs vastly from the challenges that we face in `Dare`, in that the residual information in the executable code is substantially lower than DVM or JVM bytecode.

Retargeting Dalvik bytecode is closely related to the Java decompilation. In both cases variable types need to be inferred from bytecode and the instructions interpreted and translated into the targeted language (in this case Java bytecode). A major difference is that decompiling Java bytecode requires recovering control flow statements from bytecode. In the case of Dava, this is done by making transformations to the abstract syntax tree of individual methods. An alternate approach recognizes bytecode patterns generated by known compilers. However, `Dare` avoids these complexities by mirroring the original Dalvik control flow statements directly.

Type inference in Java has been widely studied [10, 19, 5]. Conceptually, these systems generate and solve constraints on types. A solution for the constraints is found using a variety of algorithms: graph heuristics [10], solving assignment constraints before use constraints [5] or introducing new types [19]. These approaches achieve different guarantees, e.g., optimal typing [5] or polynomial time solution [10]. Our algorithm is similar to the first steps of the one in [19], in the sense that we generate constraints and then use *all* of them to find a typing. A fundamental difference between these algorithms and ours is that we only infer a subset of all types. In particular, we do not seek to determine all reference types. This makes our problem simpler, as we do not have to deal with some of the more complex issues the other algorithms solve, such as typing in the presence of multiple interface inheritance. For instance, the subtype completion of [19] or the variable splitting of [10] are not necessary in our case. The constraint-solving algorithm we use for type inference has been used by other authors [27, 16].

The idea of having a rich intermediate language (with types or semantics) is not new. For example, Lim and Reps [20] developed a language called TSL for describing the semantics of an instruction set, along with a run-time system to support the static analysis of executables written in that instruction set. A language similar to TSL called λ-RTL was developed by Ramsey and Davidson [35]. Typed Assembly Language (TAL) extends traditional untyped assembly languages with typing annotations, memory management primitives, and a sound set of typing rules. These typing rules guarantee the memory safety, control flow safety, and type safety of TAL programs [26]. The goal of the Tyde IR was to support compact retargetting rules, and so the types in Tyde IR are geared towards reducing the number of translation rules.

10. CONCLUSION

We have presented algorithms to retarget Android applications to Java bytecode. The retargeting is done in two steps. First, the Dalvik application package is converted to the Tyde intermediate representation. The Tyde IR is then translated to Java bytecode and structured into discrete class files. The associated `Dare` tool employs strong type inference algorithms and translates the complex Tyde IR to Java bytecode using only nine translation rules. Further, `Dare` can output verifiable Java bytecode even when the input is Dalvik-unverifiable.

`Dare` represents a substantial step forward in the quality retargeting of Android applications. We evaluated `Dare` against the state of the art Dalvik retargeting tools over a corpus of 1,100 applications. These experiments show that `Dare` retargeted over 99% of these applications – a 40% increase over the next best tool. By recovering application class files, `Dare` enables the use of existing Java frameworks and tools for Android application analysis, and enables future studies of smartphone applications.

11. ACKNOWLEDGEMENTS

We would like to thank Matt Fredrikson, Stephen McLaughlin and Dinghao Wu for editorial comments during the writing of this paper. We also thank Drew Davidson for editorial comments and for testing the released version of Dare. This material is based upon work supported by the National Science Foundation Grants No. CNS-0905447, CNS-1064944 and CNS-0643907. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

12. REFERENCES

[1] dex2jar. https://code.google.com/p/dex2jar/.

[2] Jad java decompiler download mirror. http://http://www.varaneckas.com/jad.

[3] Android apps on the Android market, March 15, 2012. http://www.appbrain.com/stats/number-of-android-apps, 2012.

[4] I. Asrar. Android.counterclank found in official android market. http://www.symantec.com/connect/blogs/androidcounterclank-found-official-android-market/.

[5] B. Bellamy, P. Avgustinov, O. de Moor, and D. Sereni. Efficient local type inference. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 475–492, 2008.

[6] C. Cifuentes. Interprocedural data flow decompilation. *Journal of Programming Languages*, 4:77–99, 1996.

[7] E. Dupuy. Jd java decompiler. http://java.decompiler.free.fr/.

[8] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of the 9th USENIX conf. on OS design and implementation*, 2010.

[9] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, pages 21–21, 2011.

[10] E. Gagnon, L. J. Hendren, and G. Marceau. Efficient inference of static types for java bytecode. In *SAS'00*, pages 199–219, 2000.

[11] Gartner. Gartner says android to command nearly half of worldwide smartphone operating system market by year-end 2012. http://www.gartner.com/it/page.jsp?id=1622614, April 2011.

[12] Google. Dalvik vm: Code and documentation. http://code.google.com/p/dalvik/.

[13] J. Hamada. New android threat gives phone a root canal. http://www.symantec.com/connect/blogs/new-android-threat-gives-phone-root-canal, March 2011.

[14] HP-Fortify. Source Code Analyzer (SCA). https://www.fortify.com/products/hpfssc/source-code-analyzer.html.

[15] IBM. Wala: T.J. watson libraries for analysis. http://wala.sourceforge.net/wiki/index.php/Main_Page, October 2011.

[16] R. T. Johnson. *Verifying Security Properties using Type-Qualifier Inference*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2007.

[17] Kaspersky Lab. First SMS Trojan detected for smartphones running Android. http://www.kaspersky.com/about/news/virus/2010/First_SMS_Trojan_detected_for_smartphones_running_Android, Aug. 2010.

[18] D. Kellogg. In U.S. Market, New Smartphone Buyers Increasingly Embracing Android. http://blog.nielsen.com/nielsenwire/online_mobile/in-u-s-market-new-smartphone-buyers-increasingly-embracing-android/, Sep. 2011.

[19] T. B. Knoblock and J. Rehof. Type elaboration and subtype completion for java bytecode. *ACM Trans. Program. Lang. Syst.*, 23:243–272, March 2001.

[20] J. Lim and T. Reps. A system for generating static analyzers for machine instructions. In *Proc. International Conference on Compiler Construction (CC)*, 2008.

[21] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[22] P. McDaniel and W. Enck. Not So Great Expectations: Why Application Markets Haven't Failed Security. *IEEE Security & Privacy Magazine*, 8(5):76–78, September/October 2010.

[23] J. Meyer, D. Reynaud, and I. Kharon. Jasmin home page. http://jasmin.sourceforge.net/, 2004.

[24] J. Miecznikowski and L. Hendren. Decompiling java using staged encapsulation. In *WCRE '01: Proc. of the 8th Working Conf. on Reverse Engineering*. IEEE Computer Society.

[25] J. Miecznikowski and L. J. Hendren. Decompiling java bytecode: Problems, traps and pitfalls. In *CC '02: Proc. of the 11th International Conference on Compiler Construction*, pages 111–127, London, UK, 2002. Springer-Verlag.

[26] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system f to typed assembly language. In *The Twenty-Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.

[27] A. C. Myers. Jflow: practical mostly-static information flow control. In *Proc. of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.

[28] N. A. Naeem and L. Hendren. Programmer-friendly decompiled java. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 327–336. IEEE Computer Society, 2006.

[29] D. Octeau, W. Enck, and P. McDaniel. The ded Decompiler. Technical Report NAS-TR-0140-2010, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, Sep. 2010.

[30] D. Octeau, S. Jha, and P. McDaniel. Retargeting Android Applications to Java Bytecode. Technical Report NAS-TR-0150-2011, NSRC, Dept. of CSE, Pennsylvania State University, Sep. 2011.

[31] D. Octeau, P. McDaniel, and W. Enck. ded: Decompiling Android Applications. http://siis.cse.psu.edu/ded/, 2011.

[32] D. Octeau, P. McDaniel, and S. Jha. Dare: Dalvik retargeting. http://siis.cse.psu.edu/dare/, 2012.

[33] Oracle. Maxine vm. https://wikis.oracle.com/display/MaxineVM/Home, 2004.

[34] T. A. Proebsting and S. A. Watterson. Krakatoa: Decompilation in java (does bytecode reveal source?). In *3rd USENIX Conference on Object-Oriented Technologies and Systems*, 1997.

[35] N. Ramsey and J. Davidson. Specifying instruction semantics using λ–RTL. Unpublished manuscript, 1999.

[36] J. Rehof and T. A. Mogensen. Tractable constraints in finite semilattices. In *Science of Computer Programming*, pages 285–300. Springer-Verlag, 1996.

[37] R. Vallee-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *International Conference on Compiler Construction, LNCS 1781*, pages 18–34, 2000.

[38] H. van Vliet. Mocha, the java decompiler. http://www.brouhaha.com/~eric/software/mocha/, 2001.