The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

AUTOMATED CERTIFICATION OF ANDROID APPLICATIONS

A Thesis in

Computer Science and Engineering

by

Damien Octeau

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2010

The thesis of Damien Octeau was reviewed and approved* by the following:

Patrick McDaniel
Associate Professor of Computer Science and Engineering
Thesis Adviser

Trent Jaeger
Associate Professor of Computer Science and Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

ABSTRACT

Smart phone applications are often incompletely vetted, poorly isolated, and installed by users without restraint. Such behavior is fraught with peril: applications containing malicious logic or critical vulnerabilities are likely to be identified only after substantial damage has already occurred. Unfortunately, the limitations of application markets makes them a poor agent for certifying that applications are secure. This thesis presents a certification process that allows the consumers of applications to validate applications' security directly. Built for the Android cellular platform, we reverse engineer downloaded application images into application source code and thereafter use static analysis to detect potential security vulnerabilities. We develop a multi-stage process for VM retargeting and code recovery and detail their implementation within our automated tools. A study of the top 1,100 free Android market applications recovers source code for over 95% of the 143 thousand class files containing over 12 million lines of code. A preliminary analysis of the recovered source code identified over 3,100 potential vulnerabilities indicating issues over a broad range of features.

Table of Contents

List of Tables

List of Figures

Acknowledgments

First, I would like to thank my advisor, Patrick McDaniel. This work would not have been possible without his patience and his advice, and his help in writing this thesis was priceless.

Also, I would like to thank the members of the SIIS Laboratory, who were always a source of good advice. In particular, I would like to express my gratitude to Will for his guidance at the start of the project, as well as his considerable help in writing this thesis. I am also very thankful to Tom for his help with day-to-day issues, such as the occasional system crashes.

Most importantly, I would like to thank my parents, as well as my extended family, for the support they have given me throughout these two years. They have made my stay far from home a little easier.

Chapter 1

Introduction

1.1 Introduction

The explosion of smartphones has led to a cottage industry in phone applications. Developers use phone APIs to build applications sold for as little as 99¢ through *application markets*. For example, the Apple iPhone [1] and Android [28] platforms provide easy to use on-phone interfaces for browsing literally thousands of applications. Users simply click a button to download and install an application. For technical and logistical reasons, existing application markets do little to detect malicious or vulnerable applications. Thus, "*caveat emptor* "–users have little information on which to judge the trustworthiness of downloaded applications. Misplaced trust in these applications has left users' phone data and interfaces vulnerable to malicious code [30, 26].

In response to the limitations of current markets, recent experimental systems have explored other means of vetting smart phone applications. The Kirin system [27] inspects the permissions associated with each application as it is installed. Based on a formal analysis of a program's manifest, applications that govern sensitive interfaces poorly or attempt or request unsafe sets of permissions are rejected. In contrast, the Saint system [40] seeks to regulate how smartphone applications share information by enforcing safe workflows between coordinated applications through mandatory system policies. These techniques are limited in that focus solely on the analysis of policy associated with a particular phone and application. Identifying vulnerabilities or malicious misuse of permissions is not within the scope of their analysis. Thus, otherwise valid and correctly regulated applications may, for example, leak sensitive data [30].

This thesis explores an alternative to current smart phone application market analysis. In this model, platform developers, users, cellular providers, consumer protection agencies, enterprises, professional software certification consultants, and anyone else with an interest in the security of the phones are free to certify the applications meet whatever security criteria they choose. This model changes the nature of the market by altering the oft-broken incentives– developers and markets will be encouraged to create secure code and consumers will have the tools to detect and ultimately punish those who fail to do so. Users will "vote with their feet" by avoiding developers with poor security practices and gravitate towards those with good ones.

Our certification approach automates the reverse engineering process by extracting the application source *solely from its installation image*. Thereafter, static analysis tools are used to detect potential vulnerabilities based on the evaluator's criteria. Our novel *ded* tool retargets the Dalvik virtual machine bytecode of candidate Android applications into Java `.class` files. The application's original Java source code is then recovered from the retargeted bytecode using existing Java tools (i.e., Soot [44]). Finally, commercial off-the-shelf (COTS) static analysis tools are applied to validate security related invariants defining safe phone behavior. Applications adhering to these invariants are deemed safe and can be installed, and those that fail are rejected.

We make the following contributions in this thesis:

- *We develop an automated system for Android application certification* - We develop techniques for extracting source code from installation images and leverage existing analysis tools to measure the code's adherence to desired security policies. While we explore the use of tools for analysis, the majority of the technical research contributions focus on algorithms and tools for retargeting Dalvik bytecode into valid class files.

- *We demonstrate the viability of the approach on existing market applications* - Our study of the recent top 1,100 free Android applications recovered 12 million SLOC in 143 thousand Java classes–a 95% success rate. The analysis tools flag over 3,100 locations within code as

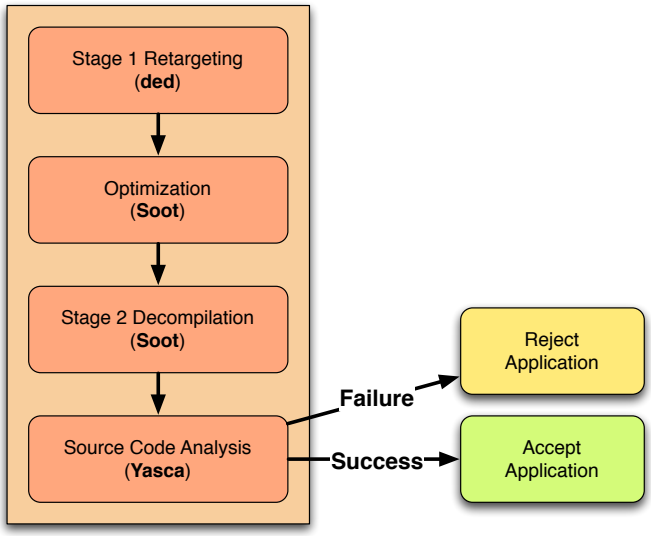


Fig. 1.1. Install-time Application Certification

potential vulnerabilities. A preliminary analysis of the vulnerabilities shows that the flags represent both common security failures and false-positives.

Note that the certification process is not a replacement for policy analysis techniques such as Kirin or Saint, but is complementary to them. Such tools inspect or regulate how applications can exercise system permissions. We aim to detect vulnerable interfaces and code, as well as malicious behaviors within the applications themselves. Thus, prior tools sought to analyze different aspects of the system than those discussed here, the former relating to the application behaviors within the phone and the latter about the structure and behaviors of the application itself.

1.2 Certification Approach

Illustrated in Figure 1.1, certification consists of two phases; *a*) source code recovery via multi-stage decompilation, and *b*) analysis of recovered code against safety-preserving invariants.

Java bytecode decompilation has been studied since its introduction in the 1990s. Tools such as Mocha [11] date back over a decade, with many other techniques being developed [41, 36, 35, 9, 8, 5] (see Chapter 5). Unfortunately, there exists no functional tool for the Dalvik bytecode[1]. As described in the next chapter, the JVM and DVM are vastly different architectures, thus simple modification of existing decompilers was not possible. Discussed in detail in the following chapters, we developed *ded*[2] for this purpose. Rather than directly recover Java source code from Dalvik bytecode, ded retargets Dalvik `.dex` files to Java `.class` files. The ded retargeting process yields complex, unoptimized Java bytecode. We use the Soot framework [44] to optimize the ded output, rendering Java bytecode from which the application source code can be recovered. The `.class` files are then decompiled using the optimized bytecode also using Soot.

The output of the decompilation process is the application source code. At this stage, a certifying party can use any code analysis tool at their disposal. In this thesis, we use the Yasca (Yet Another Source Code Analyzer [19]) tool with the PMD plugin [12] to evaluate the recovered source code. These tools search the code for poor coding practices, errors, and exploitable code. In the study presented in Section 4.4, with one exception, we use default tests that identify simple security issues. However, the tools are designed to allow the user to create custom analyses testing—thus allowing them to enforce whatever security criteria they deem necessary (within the limitations of the analysis process).

Note that the substance of the "security-preserving invariants" validated via analysis is critical to certification; it defines the security policy being enforced. This thesis does not attempt to identify what a good application policy is, as it is a complex context-sensitive issue. We only seek to demonstrate that the code recovery and subsequent analysis process is viable. In practice,

[1] The undx and dex2jar tools attempt to decompile `.dex` files. As discussed in Chapter 5 the tools are currently incapable of reliably recovering source code.

[2] While the dex decompiler (ded) only performs retargeting, it is named such for historical reasons. Together, ded and Soot provide decompilation of Android applications.

we expect that the kinds of analysis needed to vet application code will be specific to the platform, the users, and the security goals of the evaluator.

Chapter 2

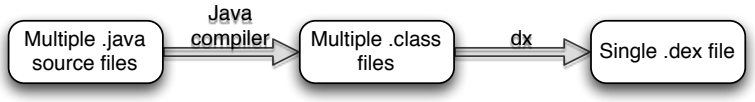
The Dalvik Virtual Machine

The Dalvik Virtual Machine (DVM) was designed for the resource constrained Android cell-phones. While the software run in DVM is compiled from Java, its bytecode and run-time environment differ substantially from that for existing JVMs. We highlight several key differences between the JVM and DVM as they relate to code recovery and certification below. We simplify the description of the VMs for ease of exposition.

2.1 Application Structure

Java applications are composed of one or more `.class` files, one file for each class. The JVM loads the bytecode in the `.class` file associated with each Java class as it is referenced using the Java class loader at run time. Conversely, a Dalvik application consists of a single `.dex` file containing all classes composing the application. The entirety of the application is loaded when it is launched by the DVM.

To support this, Android extends the compilation process:



Here, the Java compiler operates normally to produce a collection of `.class` files. The Dalvik `dx` compiler then consumes the classes, recompiles them to Dalvik bytecode, and writes the resulting application into a single `.dex` file.

Figure 2.1 provides a conceptual view of the `dx .class` to `.dex` compilation process. This process consists of the translation, reconstruction, and interpretation of three basic elements of

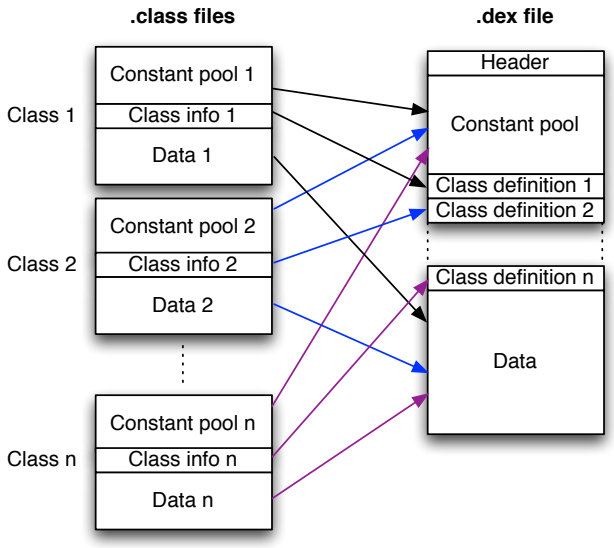


Fig. 2.1. `dx` compilation of Java classes to a DVM application (simplified view).

the application: the constant pools, the class definitions, and the data segment. A constant pool describes, not surprisingly, the constants used by a class. This includes among other items references to other classes, method names and numerical constants. The class definitions consist in the basic information such as access flags and class names. The data element contains the method code executed by the target VM, as well as other information related to methods (e.g., number of DVM registers used, local variable table and operand stack sizes) and to class and instance variables.

2.2 Register architecture

The DVM is register-based, whereas existing JVMs are stack-based. Java bytecode can assign local variables to a local variable table before pushing them onto an operand stack for manipulation by opcodes, but it can also just work on the stack without explicitly storing variables in the table. Dalvik bytecode assigns local variables to any of the 2^{16} available registers.

The Dalvik opcodes directly manipulate registers, rather than accessing elements on a program stack. For example, Figure 2.2 shows how a typical operation ("add" function Figure 2.2(a)) is implemented in Java (stack) versus Dalvik (register) virtual machine. The Java bytecode shown in Figure 2.2(b) pushes the local variables `a` and `b` onto the operand stack using the `iload` opcode, and returns the result from the stack via the `ireturn` opcode. By comparison, the Dalvik bytecode shown in Figure 2.2(c) simply references the registers directly.

2.3 Instruction set

The Dalvik bytecode instruction set is substantially different than that of Java: 218 opcodes vs. 200, respectively. The nature of the opcodes is very different: for example, Java has tens of opcodes dedicated to pushing and pulling elements between the stack and local variable table. Obviously the Dalvik bytecode instruction set does not require any such opcodes. Moreover, as illustrated in the example in Figure 2.2, Dalvik instructions tend to be longer than Java instructions as they include the source and destination registers (when needed). As a result, Dalvik applications require fewer instructions. Applications encoded in Dalvik bytecode have on average 30% fewer instructions than in Java, but have a 35% larger code size (bytes) [21]. This increased code size has limited impact on performance, as the DVM reads instructions by units of two bytes.

2.4 Constant pool structure

Java applications necessarily replicate elements in constant pools within the multiple `.class` files, e.g., referrer and referent method names. The `dx` compiler attempts to reduce application size by eliminating much of this replication. Dalvik uses a single large constant pool that all classes simultaneously reference. Additionally, `dx` eliminates some constants by inlining their values directly into the bytecode. In practice, pool elements for integers, long integers,

```
public int add(int a, int b)
{
    return a + b;
}
```

(a) Source Code

```
public int add(int, int)
  0:  iload_1
  1:  iload_2
  2:  iadd
  3:  ireturn
```

(b) Java (stack) bytecode

```
public int add(int, int)
  0:  add-int   v0,v2,v3
  2:  return    v0
```

(c) Dalvik (register) bytecode

Fig. 2.2. Register vs. Stack Opcodes

and single and double precision floating-point constants simply disappear as bytecode constants during the transformation process.

2.5 Control flow Structure

Programmatic control flow elements such as loops, switch statements and exception handlers are structured very differently by Dalvik and Java bytecode. To simplify, Java bytecode structure loosely mirrors the source code, whereas Dalvik bytecode does not. The restructuring is likely performed to increase performance, reduce code size, or address changes in the way the underlying architecture handles variable types and registers.

Figure 2.3 shows an example of such differences for a simple switch statement. The Java bytecode on Figure 2.3(b) follows the order of the source code of Figure 2.3(a). It starts with a single opcode for the switch statement (`tableswitch`), which describes all the cases and the offsets to corresponding handlers. It is followed by the code for the handlers. After that, we find the rest of the code, which translates the `return` instruction. On the other hand, the Dalvik bytecode on Figure 2.3(c) is quite different. First, the switch instruction (`packed-switch`) does not describe the cases and offsets to handlers; instead, it branches to a `packed-switch-data` pseudo-instruction at the end of the method code. This pseudo-instruction describes the cases and corresponding offsets to handlers. Note that unlike with Java bytecode, the default case is implicit: it is the instruction located right after the `packed-switch` instruction. Note also that a `nop` instruction is inserted before the `packed-switch-data` pseudo-instruction to make the latter 4-byte aligned.

```
int do_operation(int a, int b, int op)
{
    int result = 0;

    switch(op)
    {
    case 1:
        result = a + b;
        break;
    case 2:
        result = a - b;
        break;
    case 3:
        result = a * b;
        break;
    default:
        result = a;
        break;
    }

    return result;
}
```

(a) Source Code

```
int do_operation(int, int, int)
   0:   iconst_0
   1:   istore  4
   3:   iload_3
   4:   tableswitch{ //1 to 3          } Switch instruction with
                1: 32;                  } complete description
                2: 40;                  } of the values and
                3: 48;                  } targets
                default: 56 }
  32:  iload_1                          }
  33:  iload_2                          }
  34:  iadd                             } Handler 1
  35:  istore  4                        }
  37:  goto    59                       }
  40:  iload_1                          }
  41:  iload_2                          }
  42:  isub                             } Handler 2
  43:  istore  4                        }
  45:  goto    59                       }
  48:  iload_1                          }
  49:  iload_2                          }
  50:  imul                             } Handler 3
  51:  istore  4                        }
  53:  goto    59                       }
  56:  iload_1                          } Default handler
  57:  istore  4                        }
  59:  iload   4
  61:  ireturn
```

(b) Java (stack) bytecode

```
public int do_operation(int, int, int)
   0:   const/4 v0,0
   1:   packed-switch v4,16             } Switch instruction
   4:   move v0,v2                      } Default handler
   5:   return v0
   6:   add-int v0,v2,v3                } Handler 1
   8:   goto 4                          }
   9:   sub-int v0,v2,v3                } Handler 2
   11:  goto 4                          }
   12:  mul-int v0,v2,v3                } Handler 3
   14:  goto 4                          }
   15:  nop
   16:  packed-switch-data{ size: 3             } Switch description
                            lowest value: 1     }
                            targets: 6, 9, 12 } }
```

(c) Dalvik (register) bytecode

Fig. 2.3. Register vs. Stack Opcodes for a Switch Statement

2.6 Ambiguous typing of primitive types

Java bytecode variable assignments distinguish between integer (int) and single-precision floating-point (float) constants and between long integer (long) and double-precision floating-point (double) constants. However, Dalvik constant assignments (int/float and long/double) use the same opcodes for integers and floats, e.g., the opcodes are untyped beyond specifying precision. This complicates decompilation of Dalvik bytecode because the variable type is not indicated by its declaration. Thus, the decompilation process must observe a variable creation and inspect its subsequent use to infer its type to create accurate correct Java bytecode and constant pools. This is a specific instance of a broad class of widely-studied *type inference* problems. Note that an incorrect inference (and thus type) may result in incorrect behavior (and analysis) of the decompiled program.

2.7 Null references

The Dalvik bytecode does not specify a `null` type, instead opting to use a zero value constant. Thus, constant zero values present in the Dalvik bytecode have ambiguous typing that must be recovered. The decompilation process must recover the `null` type by inspecting the variable's use *in situ*. If the `null` type is not correctly recovered, the resulting bytecode can have illegal integer zero assignments to object references, and vice-versa.

2.8 Comparison of object references

The Java bytecode uses typed opcodes for the comparison of object references (`if_acmpeq` and `if_acmpne`) and for null comparison of object references (`ifnull` and `ifnonnull`). The Dalvik bytecode uses a more simplistic integer comparison for these purposes–respectively the comparison between two integers and the comparison of integer to zero. This requires the decompilation process to recover types for integer comparisons. Again, incorrect or missed inferences

could result in illegal bytecode, typing violations, and ultimately inaccurate decompiled source code.

2.9 Example

Figure 2.4 illustrates several of the differences described in this chapter with a somewhat contrived example. The assignment `const/4 v1,1` at offset 0 is ambiguous: it could either correspond to a 32-bit integer (int) or a 32-bit floating-point constant (float). Then the `if-eq v4,v1,9` instruction at offset 1 is also ambiguous: it could either be a comparison between two integers or between two object references. The next instruction, `const/4 v0,0`, could either be an assignment for a null reference, or integer or floating-point constant with value 0. Then the `if-nez v0,11` instruction can either be a comparison to 0 or to a null reference. Also, the `const-wide/high16 v1,16388` and `const-wide v1,4608083138725491507` instructions at offsets 6 and 11 could either be assignments for 64-bit integers (long) or 64-bit floating-point constants (double). Similarly, the `return-wide v1` instruction at offset 8 has the same ambiguity: it could either return a long or a double. None of these ambiguities can be found in the original Java bytecode, therefore we have to recover this type information for the code retargeting.

This example also illustrates how registers can get reassigned at runtime: here register v1 is first assigned to a 32-bit constant. Then later in the code it is becomes the first half of a 64-bit constant.

The next chapter explains how these issues are solved, as well as all other differences described in this chapter.

```
double return_a_double(int a)
{
  Object x;

    if(a != 1) {
        x = null;
    }
    else {
        x = this;
    }

    if(x == null)
        return 2.5;

    return 1.2;
}
```

(a) Source Code

```
double return_a_double(int)
  0:  const/4     v1,1
  1:  if-eq       v4,v1,9
  3:  const/4     v0,0
  4:  if-nez      v0,11
  6:  const-wide/high16 v1,16388
  8:  return-wide v1
  9:  move-object v0,v3
  10: goto        4
  11: const-wide v1,4608083138725491507
  16: goto        8
```

(b) Dalvik bytecode

Fig. 2.4. Example of ambiguous assignment and comparison, and of register reassignment

Chapter 3

Application Retargeting

The first step in the decompilation process is to retarget the application .apk to Java classes. Figure 3.1 shows the retargeting process with a focus on the three central research challenges we faced; recovering typing information, translating the contant pool based in part on inferred types, and the retargeting of the DVM bytecode. Note that for brevity we omit many important issues that relate to the implementation of ded, but rather focus on the key research challenges faced in its creation. For example, parsing the original Dalvik executable is straightforward, as the `.dex` format is clearly defined in the Android SDK documentation.

3.1 Type Inference

The first step in retargeting is to identify class and method constants and variables. However, the Dalvik bytecode does not always provide enough information to determine the type of a variable or constant from its register declaration. There are two generalized cases where variable types are ambiguous: 1) constant and variable declaration only specifies the variable width (e.g., 32 or 64 bits), but not whether it is a float, integer, or null reference; and 2) comparison operators do not distinguish between integer and object reference comparison (i.e., null reference checks).

Type inference has been widely studied [43]. The seminal Hindley-Milner [37] algorithm provides the basis for type inference algorithms used by many languages such as Haskell and ML. Broadly speaking, the approach is to determine unknown types by observing how variables are used in operations with known type operands. Other languages such as OCAML [22] perform strong type inference and others like Perl [45] employ weaker algorithms using similar techniques.

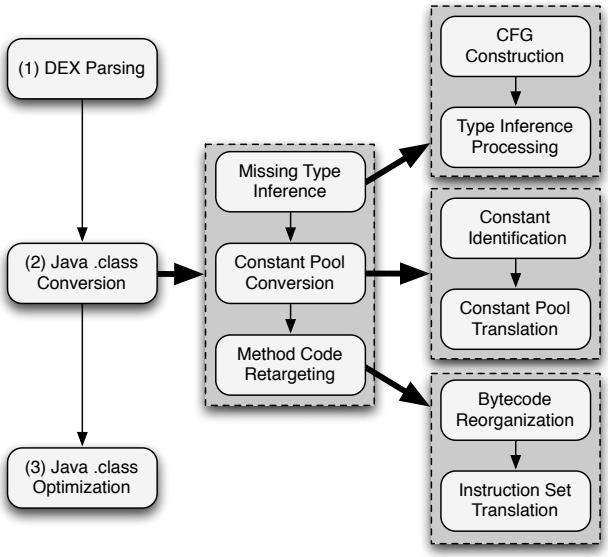


Fig. 3.1. DVM retargeting

ded adopts the accepted approach: it infers register types by observing how they are used in subsequent operations with known type operands. Dalvik registers loosely correspond to Java variables. Because Dalvik bytecode reuses registers whose variables are no longer in scope, we must evaluate the register type within its context of the method control flow, i.e., inference must be *path-sensitive*. Note further that ded type inference is also *method-local*. Because the types of passed parameters and return values are identified by method signatures, there is no need to search outside the declaring method.

There are three ways ded infers a register's type. First, any comparison of a variable or constant with a known type exposes the type. Comparison of dissimilar types requires type coercion in the original Java code, which is propagated to the Dalvik bytecode. Hence any legal Dalvik comparison must involve registers of the same type. Second, instructions such as `add-int` only operate on specific types, and manifestly expose typing information. Third, any instruction

that passes a register another method or assigns it to a return value exposes the type via the method signature.

The ded type inference algorithm proceeds as follows. After reconstructing the control flow graph, ded identifies any ambiguous register declaration. For each such register, ded walks the instructions in the control flow graph starting from its declaration. Each branch of the control flow encountered is pushed onto a inference stack, e.g., ded performs a depth-first search of the control flow graph looking for type-exposing instructions. If a type-exposing instruction is encountered, the variable is labeled and the process is complete for that variable. There are three events that cause a branch search to terminate: a) when the register is reassigned to another variable (e.g., a new declaration is encountered), b) when a return function is encountered (all DVM methods terminate with a return), and c) when an exception is thrown. After a branch is abandoned, another is popped off the stack and the search continues. Lastly, type information is forward propagated, modulo register reassignment, through the control flow graph from each register declaration to all subsequent ambiguous uses. This algorithm resolves all ambiguous primitive types, except for one isolated case when all paths leading to a type ambiguous instruction originate with ambiguous constant instructions (e.g., all paths leading to an integer comparison originate with registers assigned a constant zero). However, in this case, the exact type does not impact decompilation, and we can safely assign a default type (e.g., integer).

Note that it is sufficient to find *any* type-exposing instruction for a given register assignment. Any two instructions that expose different types for the same register would represent illegal bytecode. If this were to occur, the primitive type would be dependent on the path taken at run time, a clear violation of Java's type system.

Appendix A illustrate this approach by presenting the detailed resolution of the ambiguous types of the method presented on Figure 2.4.

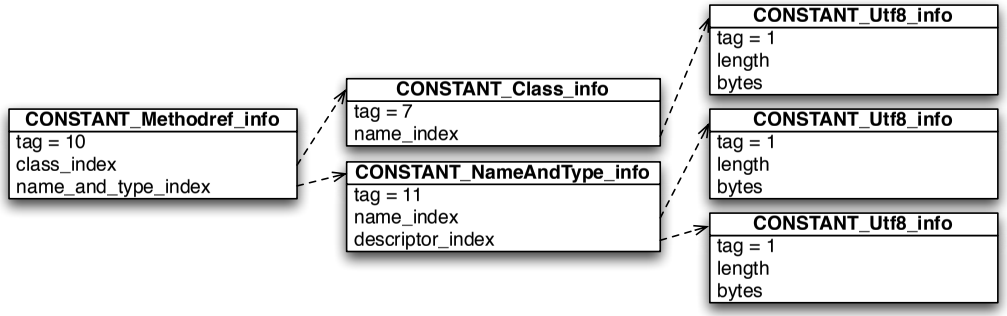
3.2 Constant Pool Conversion

There are two central differences between `.dex` and `.class` file constant pools: 1) Dalvik maintains a single constant pool for the application and Java maintains one for each class, and 2) Dalvik bytecode places primitive type constants directly in the bytecode, whereas Java bytecode uses the constant pool for most references. We convert constant pool information in two steps, as shown in Figure 3.1.

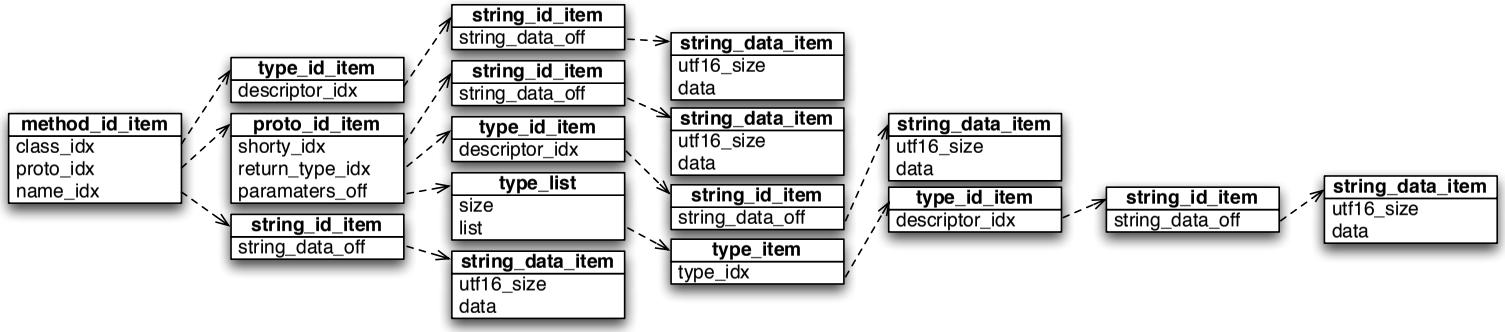
The first step is to identify which constants are needed for a `.class` file. Constants include references to classes, methods, and instance variables. ded traverses the bytecode for each method in a class, noting such references. ded also identifies all constant primitives specified in the Dalvik bytecode. Here, ded notes the types determined during type inference, described in Section 3.1.

Once ded identifies the constants required by a class, it adds them to the target `.class` file. For primitive type constants, new entries are created. For class, method, and instance variable references, we must populate the Java constant pool entry based on information available in the Dalvik constant pool. The two constant pool entries differ in complexity. Specifically, Dalvik constant pool entries use significantly more references to reduce memory overhead.

Figure 3.2 depicts the method entry constant in both Java and Dalvik formats. Other constant pool entry types have similar structures. Each box is a data structure. Index entries (denoted as "idx" for the Dalvik format) are pointers to a data structure. The Java method constant pool entry, Figure 3.2(a), provides three strings: 1) the class name, 2) the method name, and 3) a descriptor string representing the argument and return types. The Dalvik method constant pool entry, Figure 3.2(b), also contains these strings, but uses more indirection. ded ignores Dalvik-specific entries such as "shorty" strings used as simplified method descriptors.



(a) Java constant pool entry



(b) Dalvik constant pool entry

Fig. 3.2. Constant pool entry structure for a method reference

3.3 Method Code Retargeting

The final stage of the retargeting process is the translation of the method code. This is a two stage process, as shown in Figure 3.1. First, we preprocess the bytecode to reorganize structures that cannot be directly retargeted. Second, we linearly translate DVM bytecode to the JVM.

The preprocessing phase considers multidimensional arrays. Both Dalvik and Java use blocks of bytecode instructions to create multidimensional arrays; however, the instructions have different semantics and layout. ded reorders and annotates the bytecode with array size and type information to allow linear instruction translation.

The bytecode translation linearly processes each Dalvik instruction. First, ded maps each referenced register to a Java local variable table index. Second, ded performs an instruction translation for each encountered Dalvik instruction. As Dalvik bytecode is more compact and takes more arguments, one Dalvik instruction frequently expands to multiple Java instructions. Third, ded patches the relative offsets used for branches based on preprocessing annotations. Finally, ded defines exception tables that describe `try/catch/finally` blocks. The result when combined with the constant pool is a Java `.class` file.

Table 3.1 provides two example translation rules. The first example illustrates the translation of an add instruction. For each register, ded creates a corresponding Java local variable, i.e., $d_0 \rightarrow d'_0$, $s_0 \rightarrow s'_0$, etc. The translation creates four Java instructions: two instructions to push the variables onto the stack, one instruction to add, and one to pop the result. The second example, `invoke-virtual`, shows the translation of a virtual method invocation. This example demonstrates two intricate details for which ded must account. First, Dalvik's register method arguments do not always correspond to Java's variable method arguments. In Dalvik, registers are always 32-bits, and 64-bit variables are stored in two adjacent registers. Hence, ded must inspect the method descriptor when translating registers to variables for method invocation.

Second, Dalvik bytecode commonly omits the `move-result` instruction if the return value is not used. However, the Java bytecode must always pop the return value off the stack, regardless of its use. ded includes related logic to ensure stack integrity.

Table 3.1. Example Dalvik to Java bytecode translation rules

Dalvik Bytecode Instructions	Java Bytecode Instructions	Description
`add-int` d_0, s_0, s_1	`iload` s'_0 `iload` s'_1 `iadd` `istore` d'_0	Integer addition
`invoke-virtual` $s_0, \ldots, s_k, m_r$ `move-result` d_0	`iload` s'_0 ... `iload` s'_l `invokevirtual` m'_r `istore` d'_0	Virtual method invocation with assigned return value.

3.4 Optimization and Decompilation

At this stage, the retargeted `.class` files can be decompiled using existing tools, e.g., Fernflower [5] or Soot [44]. However, ded's bytecode translation process yields unoptimized Java code. For example, Java tools often optimize out unnecessary assignments to the local variable table, e.g., unneeded return values. The lack of optimization yields complex decompiled code and frustrates its subsequent analysis. Furthermore, artifacts of the retargeting process lead to potential decompilation errors in some decompilers. To remedy these issues, we post process `.class` files using a bytecode optimizer such as Soot [44].

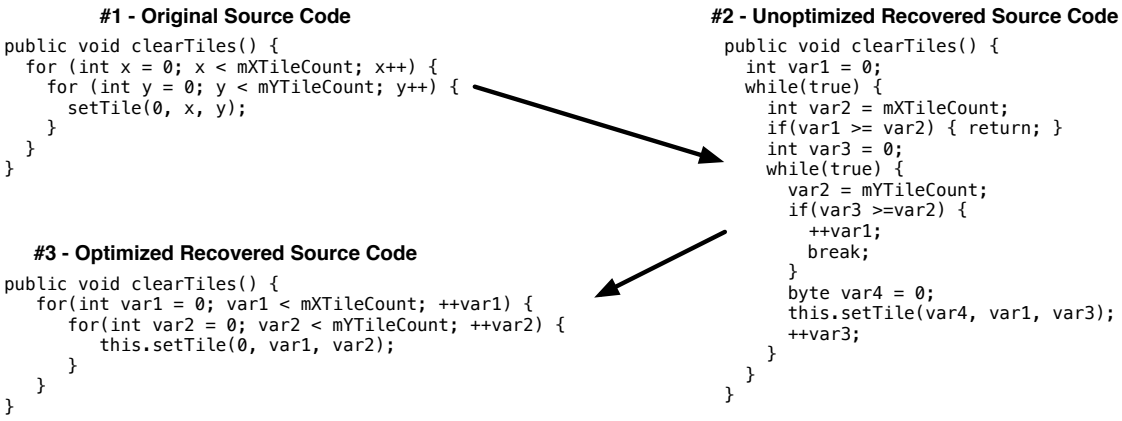


Fig. 3.3. Original, recovered unoptimized, and recovered optimized source code.

The need for bytecode optimization can be demonstrated by looking at decompiled loops. Most decompilers convert `for` loops into infinite loops with `break` instructions. While the resulting source code is functionally equivalent to the original, it is significantly more difficult to understand and analyze, especially for nested loops. Figure 3.3 shows example retargeted source with and without optimization; (#1) shows the original source code, (#2) shows unoptimized decompiled code, and (#3) shows optimized decompiled code. Note that the optimized decompiled code is, modulo variable names, virtually identical to the original source, whereas the unoptimized code is nearly indecipherable.

The Soot tool [44] performs a myriad of optimization, data and control analysis, and decompilation on each program. We use Soot with default optimizations for the next two stages of application processing; optimization and decompilation (of Figure 1.1). Note that these processes can fail due to the limitations of our retargeting. We revisit the frequency and source of these failures in our evaluation of Android applications in the next chapter. Also note that other decompilers such as Fernflower may perform better than Soot. For example, an informal comparison of Soot and Fernflower decompilation suggests that the latter is more robust. We defer investigation of other tools to future work.

Chapter 4

Evaluation

This chapter studies the accuracy and results of the proposed certification process. We begin by validating the accuracy of the recovery process, then analyze the results of a preliminary certification of 1,100 applications downloaded from the Android market.

4.1 Recovery Validation

We first sought to validate the recovered code was faithful to the original source. Although tools exist to check the similarity of Java source code, we desired a much stronger comparison. The only reliable way to perform validation was, sadly, through manual inspection. Five open source applications ranging in size and origin were selected for validation. A percentage of the recovered methods (selected at random) were compared against the original source to ensure it was functionally identical. All aspects of the source code were compared, e.g., control flow, calls, arithmetic operators, etc. The validated applications included; Snake (421 LOC, 100% of methods validated), Radar (581 LOC, 20% checked), Translate (1868 LOC, 20% checked), Countdown by OpenIntents (4450 LOC, 20% checked), and WordPress (11,727 LOC, 5% checked).

The validation process found no errors in the recovery process; every method checked was functionally identical to the original code, with three exceptions. The exceptions occurred due to the limitations of Soot, and included a) incorrect `super` references, b) improper structure in `try/catch` blocks, and c) improper `super` constructor calls. These translation errors are not present in tools such as FernFlower. Note that these errors did not effect our vulnerability detection described below in any meaningful way.

4.2 Certification Evaluation

Our second analysis focuses on the results of the certification itself. The experiments described here use the top 50 free applications available from the Android Market in each of the 22 application categories (API level 4). These 1,100 applications were downloaded using an automated tool from Android Market on December 11th, 2009. Our decision to use only free applications does not meaningfully impact the experimental results (beyond a potential for the paid applications to have undergone a more thorough development process). Android makes no distinction between paid or free applications, and no addition of DRM is applied. For completeness, we successfully recovered and analyzed source code from sample paid applications without error. We omit further discussion of paid applications for brevity.

ded was compiled using gcc version 4.4.1. The code recovery experiments were run on 64-bit, 8 core, 2.33 GHz (8 GB RAM) Dell blades running Linux Ubuntu version 9.10 (kernel version 2.6.31-17). We used a customized version of Soot with default optimization options. We relaxed superfluous (for our purposes) checks on internal string representations and deactivated the elimination of `store/load/load` trios that can yield illegal code on legal input. We further deactivated the code transformations that add unnecessary (non-Android) system support code. Finally, we fixed a bug we found in version 2.3.0 of Soot: in the PackManager class, we modified the incorrect phase name `source_is_javac` to `source-is-javac` (in Soot, a phase is a processing step). Certification tests use Yasca 2.1 with the PMD plugin version 4.2.5 4-core 2.33 using PHP version 5.2.10 on a 3.20 GHz Dell desktop. SLOC counts were generated using SLOCCount version 2.26 [17].

Note that imperfect analysis is subject to false-positives; we may incorrectly flag that source code violates a tested security property. How this is handled by the certifier is subject to

policy; the developer may refactor the code to remove the violation or provide evidence/explanation that the reported error is a false positive. As with iPhone, the likely answer is probably a combination of both.

4.3 Source Code Recovery

The first set of experiments measured the code recovery process. Table 4.1 shows the number of classes that were successfully *retargeted* (converted from Dalvik to Java bytecode) and *recovered* (source code retrieved). Notably, over 99.6% of the `.class` files were successfully retargeted, and the source code for 95% of the classes was recovered.

The computational costs of recovering the source code were not onerous. The total time to recover the source code for all 1,100 applications was about 272 computation-hours, or a little over 14 minutes of processor time per application. Interestingly, the retargeting process was completely dominated by the optimization and decompilation process. The retargeting process took on average about 17 msec per class, whereas the source code recovery took about 6.8 seconds per class. The per-application recovery costs were largely linear in the number of classes (standard deviation 6.694 sec).

The processing time of Soot is mostly spent resolving referenced classes. The issue is that for the decompilation of several classes from the same application, it will often resolve the same classes several times, if classes are individually decompiled. Soot can decompile several classes at the same time, which can save a lot of time in class resolution. However, in that case, whenever a recovery error occurs, none of the classes gets decompiled. We tried to modify Soot in a way which made it possible to decompile several classes in one Soot session without this limitation. These changes consisted in catching thrown exceptions and reverting the side effects which resulted from processing a class causing a failure. We got dramatic performance improvements, but had to revert our changes because they were not always completely reliable.

Table 4.1. Application source code recovery for the top 1,100 free Android Market applications (Dec. 2009).

Category	**Size (bytes)**	**App classes**	**Retargeted classes**	**Recovered classes**	**SLOC**
Comics	3,675,436	2,767	2,764 (99.89%)	2,656 (95.99%)	158,526
Communication	17,984,508	12,976	12,925 (99.61%)	12,010 (92.56%)	991,205
Demo	4,849,012	4,236	4,235 (99.98%)	4,061 (95.87%)	374,433
Entertainment	5,103,224	5,423	5,421 (99.96%)	5,239 (96.61%)	253,033
Finance	7,116,100	5,141	5,132 (99.82%)	4,972 (96.71%)	411,413
Arcade Games	10,244,184	5,401	5,361 (99.26%)	5,114 (94.69%)	610,042
Puzzle Games	4,995,604	3,308	3,290 (99.46%)	3,141 (94.95%)	257,554
Casino Games	9,586,980	4,947	4,920 (99.45%)	4,688 (94.76%)	572,903
Casual Games	5,662,640	3,492	3,483 (99.74%)	3,316 (94.96%)	303,673
Health	7,905,040	5,260	5,251 (99.83%)	5,099 (96.94%)	463,285
Lifestyle	9,547,124	6,692	6,676 (99.76%)	6,289 (93.98%)	544,005
Multimedia	12,889,520	9,902	9,844 (99.41%)	9,336 (94.28%)	784,082
News/Weather	9,565,676	6,610	6,574 (99.46%)	6,160 (93.19%)	466,465
Productivity	18,512,912	12,376	12,324 (99.58%)	11,745 (94.90%)	1,147,235
Reference	7,159,300	4,614	4,599 (99.67%)	4,403 (95.43%)	441,536
Shopping	11,263,896	8,318	8,300 (99.78%)	8,009 (96.29%)	738,213
Social	18,090,512	14,342	14,311 (99.78%)	13,744 (95.83%)	1,176,859
Libraries	4,757,036	3,653	3,646 (99.81%)	3,527 (96.55%)	301,248
Sports	13,580,008	9,589	9,550 (99.59%)	9,288 (96.86%)	1,650,778
Themes	411,568	508	506 (99.61%)	491 (96.65%)	13,079
Tools	5,331,940	3,690	3,670 (99.46%)	3,509 (95.09%)	304,347
Travel	13,906,124	9,845	9,774 (99.28%)	9,231 (93.76%)	877,031
TOTAL	**202,138,344**	**143,090**	**142,556 (99.63%)**	**136,028 (95.06%)**	**12,840,945**

The 534 retargeting failures fell into three classes. The first class of failures was the result of unresolved class references. We confirmed that these errors can be fixed by investigating and supplying Soot precise versions of support classes needed by the failed application. We are investigating ways to automate the labor-intensive process of finding correct support libraries. The second class of errors occurred due to invalid input Dalvik bytecode. We observed a small number of applications whose bytecode had type system violations. This occurred when specific exceptional control flow structures (i.e., specific placement of `try`/`catch` blocks) were present in a method. Such bytecode can be repaired by reducing the scope of a try block to only instructions that are capable of throwing the target exception. While we were able confirm the fix by directly editing the bytecode using a binary editor and retargeting, we have yet to identify an algorithm to do this automatically.[1] The last very infrequent class of errors remains unexplained. We are continuing to explore pathological boundary cases or errors in ded that produce invalid bytecode.

The source code recovery failures also were equally diverse. Soot is an optimization tool with the ability to recover source code in most cases, but does not process certain legal program idioms (code structures) generated by ded. We encountered and refactored many of these idioms while developing ded, but were unable to resolve all of them. In particular, two central problems we have encountered involve interactions between synchronized blocks and exception handling, and complex control flows caused by break statements. The difference between the success rates of retargeted and recovered rates is largely due to Soot's inability to extract source code from these otherwise legal idioms. Note that other tools such as Fernflower [5] can be more resilient in the face of these idioms. We chose not to use Fernflower in this work because it is an online service that requires direct access to the bytecode. Sending retargeted bytecode over the Internet to a third party presents both logistical and potential legal challenges. Experiments are currently in progress to precisely evaluate the success rate of the Soot decompiler on `.class` files which

[1] Soot has a documented feature to perform this code transformation, but it failed on all our sample bytecode.

do not originate from VM retargeting. We have attempted to decompile the `rt.jar` libraries, which are part of Sun's JVM version 1.5. Out of 13260 classes, 12582 were decompiled, which is a success rate of 94.89%. More experiments are in progress, but this first result tends to show that Soot introduces important limitations that cannot be solved by modifying the ded retargeting process.

4.4 Application Vulnerability Analysis

The second battery of tests evaluated application vulnerability detection. Yasca was configured to detect *a*) **weak crypto**: the use of poor crypto algorithms, e.g., DES, unsafe PRNGs; *b*) **bad practices**: general bad programming practices, e.g., empty exception catch blocks, required but uncalled superclass constructors; *c*) **AJAX**: the use of AJAX (which Yasca/PMD deem dangerous); *d*) **file use**: files being created or modified in unsafe ways, e.g., poor log file handling; *e*) **DoS**: exposing applications to denial of service, e.g., reading from arbitrary files; *f*) **process control**: the use of external programs such as "su" or other low-level programs, or loading external Java libraries; and *g*) **hidden permissions**: permissions asserted within code, rather than through policy configuration manifest (see below).

Table 4.2 shows the results of the analysis of the Android applications. Yasca flagged over 3,000 different locations in the recovered source code. An investigation of the flagged code yields some interesting results. One surprising discovery was that there was widespread use of weak cryptographic algorithms. In particular, 56-bit DES, RC4 and MD5 were widely used. While it is unclear what these are used for, their use at all is enough for Yasca to consider the code suspect. In another vein, the vast majority of flagged code for AJAX existed in a single sports-score application. The application was tightly integrated with an online Web system, and thus these flags are likely false positives. The DoS vulnerabilities represented about 32% of the 3,119 vulnerabilities. Most of these flags were due to file reads; Yasca flags blocking I/O read calls as potential vulnerabilities requiring closer inspection to ensure an adversary cannot supply an

Table 4.2. Application Analysis - potential vulnerabilities flagged by Yasca analysis.

Category	**Weak crypto**	**Bad practice**	**AJAX**	**File use**	**DoS**	**Process control**	**Hidden permission**	**Total**
Comics	0	0	0	0	48	1	0	**49**
Communication	22	1	0	3	83	131	235	**475**
Demo	3	0	0	0	6	14	9	**32**
Entertainment	0	0	79	0	42	0	19	**140**
Finance	1	0	0	0	44	23	8	**76**
Arcade Games	0	3	0	0	34	28	0	**65**
Puzzle Games	0	0	0	0	28	24	0	**52**
Casino Games	1	4	0	0	27	8	2	**42**
Casual Games	0	1	0	0	34	3	0	**38**
Health	5	0	0	1	43	64	12	**125**
Lifestyle	24	0	2	0	67	48	28	**169**
Multimedia	15	0	1	0	100	72	156	**344**
News/Weather	5	1	0	1	46	2	50	**105**
Productivity	31	7	6	0	58	19	66	**187**
Reference	5	1	0	0	36	13	4	**59**
Shopping	9	0	0	0	53	49	28	**139**
Social	19	0	0	0	74	166	47	**306**
Libraries	60	5	0	0	30	135	25	**255**
Sports	3	2	0	0	65	22	13	**105**
Themes	8	0	0	0	0	0	0	**8**
Tools	14	3	0	4	28	7	79	**135**
Travel	10	0	0	3	63	119	18	**213**
TOTAL	**235**	**28**	**88**	**12**	**1,009**	**948**	**799**	**3,119**

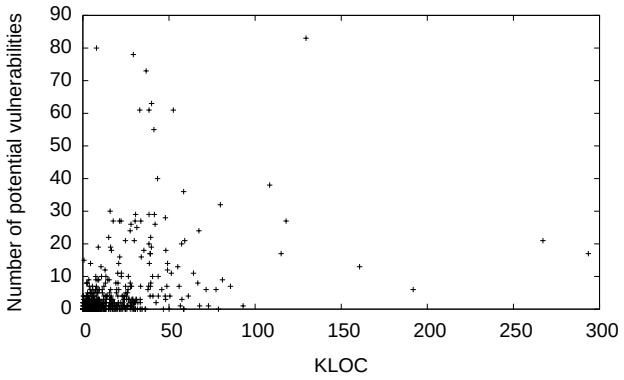


Fig. 4.1. Number of potential vulnerabilities based on lines of code in an application.

input file that consumes unexpected resources or blocks indefinitely. In many cases the adversary will not be able to control the input file, therefore it is reasonable to expect that many errors are false positives. We were also surprised at the pervasiveness of the process control flags; it seems that several programs were using the underlying embedded Linux tools. e.g., "su." Such uses are dangerous under normal circumstances, but in Android they serve to bypass the permission system. Therefore they should be used carefully if at all.

Lastly, our Android-specific test searched for "hidden permissions" used for IPC. We characterize such instances as potential vulnerabilities because it requires developers to correctly enforce policy through code, rather than relying on the platform permission system. Thus, poorly implemented applications may silently violate the system policy. Moreover, because they are embedded in source code, policy analysis tools are not able to evaluate their correctness (e.g., Kirin [27]). Note that our custom permission check is only one (simple) heuristic for identifying potentially vulnerable security policy specification. However, even simple heuristic demonstrates that these intents were used in a small number of applications, but that those that did use them did so frequently. Future work will perform a more sophisticated analysis of security policy embedded within code.

Figure 4.1 plots the number of identified potential vulnerabilities in an application based on number of lines of code in that application. Note that both the application with the most vulnerabilities and the application with the most LOC were removed for presentation purposes. The plot does not indicate any obvious trend of potential vulnerabilities based on the size of the application. We experimented with several different representations (e.g., number of classes, binary size) with similar results.

Finally, we stress here that these are preliminary tests. Beyond the brief notes presented here, we have not systematically evaluated the flagged source code to see which represent real vulnerabilities and which are false positives. The focus of this work was to validate the certification process was feasible in practice. The identification of over 3,000 potential problems within the tested application serves as validation of this. The real certification of 1,100 applications, however, is an orthogonal and substantial effort in its own right.

Chapter 5

Related Work

Java decompilation has been around almost as long as the language itself. Proebsting invented a comprehensive java decompilation method [41] in 1997 shortly after Mocha [11] decompiler was released. This remained the state of the art until 2001 when Dava [36, 35, 39] was introduced. Dava is built on the Soot tool used throughout. The Soot framework uses sophisticated type inference [29, 20, 38] and code analysis techniques to accurately recover the original code. Closed source decompilers such Jad [9], JD [8] and Fernflower [5] are extremely effective at recovering source code, but their associated algorithms are not made public.

Ours is not the first work to use decompilation to enable security analysis [25]. Tools such as Yasca [19] and PMD [12], FindBugs [6], JLint [2], QJ-Pro [14], Lint4j [10], Checkstyle [3] verify best coding practices and the absence or presence of vulnerabilities or bugs. Interested readers are referred to surveys on code level [46] and bytecode-level [32] security analysis. These and similar tools have applied in many domains of security [24]. In particular, [34] presents a technique based on a points-to analysis for the detection of vulnerabilities in Java applications. This work is extended in [33], which presents a range of static and runtime analyses of Web applications. Also, significant work has been done in large scale static analysis of C code [23, 42]. Our preliminary evaluation relies on simple lexical analysis tools [47, 24, 34]. Thus, future work will focus on more powerful analysis tools that will enable stronger certification.

Two parallel efforts are also attempting to recover source code from Android application code files. Like ded, the undx [18] and dex2jar [4] tools attempts to recover Dalvik application class files from the installation image. Our experiments using these tools on a array of applications demonstrated that they only recover about 14% (undx) and 6% (dex2jar) of classes. Further

investigation of the tools showed that they lacked many of the key algorithms and structures needed to accurately retarget Android applications.

Chapter 6

Conclusion and Future Work

6.1 Conclusions

Smartphone application development is one of the fastest growing technology markets in the world. However, apart from weak assurances of market providers, users currently have no way of evaluating whether these applications are safe to use. This thesis has presented a new user-centric certification method for this evaluation; organizations and concerned users extract the application source code from the installation image and perform code-level analysis of its structure. We have illustrated the design and challenges of the ded decompiler and performed a large but preliminary analysis of 1,100 mobile phone applications. The experiment shows that the certification process is feasible in practice.

This thesis has described tools for the execution of application certification. However, this is only half of the story. Defining security policies appropriate for cell phones and the data they manage is a complex and context-sensitive process. Moreover, to enforce those policies, we need to perform deeper validation of the source code recovered using stronger analysis techniques. Our future work will focus centrally on these critical activities.

6.2 Improving the Static Analysis

The research litterature usually presents lexical analysis tools as inherently limited by a high false positive rate [24]. There are two major reasons for this. First, some of the detection rules are not correctly implemented, which prevents an accurate detection of vulnerabilities. The second reason is that the scope of these tools is inherently limited to the detection of code

patterns and does not include any flow analysis. The first problem can be solved by rigorously implementing appropriate rules. However, the second one requires using different, more complex analysis tools.

Future work will seek to improve the detection rules used in the lexical analysis tools. In particular, [46] defines a taxonomy of secure coding heuristics for Java 6. Having detection rules actually follow these heuristics would improve the lexical static analyses. Augmenting the taxonomy with Android-specific rules and implementing them would also improve the analyses. For example, PMD claim that it has rules [13] to check that a program enforces Sun's secure coding guidelines [15]. However, it can only detect a very small subset of these vulnerabilities. In particular, a well-known vulnerability (affecting Sun's JVM until 2001 [16]) due to improper treatment of mutable inputs is not detected. It is important that tools detect such vulnerabilities, as many smartphone application developers often do not respect secure coding practices [7]. It implies a rigorous treatment of all the subtleties [31] that current tools do not take into account.

But since the scope of these tools is clearly limited, we also need to start analyzing information flows in Android. For example, our analysis has detected hidden permissions by using a Yasca rule which is just a somewhat elaborate version of the UNIX `grep` utility. It does not tell us anything about how a permission is used, what method called the method which used it, etc. Building a call graph would be needed to get access to this information. Such an analysis may be used to extend the scope of the Kirin system [27] with permissions buried in the source code.

In order to generate other more elaborate analyses, existing Java tools can be used. For example, Soot allows us to define our own analyses by providing us with useful tools (whole-program call graph construction, etc.). Also, as presented in Chapter 5, there has been extensive work done in static analysis of Java programs. We can leverage this existing work and customize it with Android-specific security requirements.

However, such analyses are completely dependent on the quality of the retargeted code. That implies that existing problems will have to be fixed if we want to get analyses as accurate as possible. Also, some information is not currently used by our retargeting process, namely annotations and debugging information. While we do not expect that these elements will be useful for automated analysis, they would make the decompiled output easier to comprehend and thus facilitate manual code auditing.

Appendix

Type Resolution Example

In this section, we illustrate the type inference approach introduced in Section 3.1 with the example presented in Section 2.9. The resolution method presented here follows the implementation used in ded. Figure A.1 shows the Control Flow Graph (CFG) for the method for which we want to determine ambiguous types.

A.1 First Pass

The method uses five registers, including two for the input parameters: v3 is an object reference (`this` reference to the current instance) and v4 is an integer (the method argument). The forward propagation starts with the state presented in Table A.1.

Table A.1. Type state after type propagation to node 0

Register	v0	v1	v2	v3	v4
Type at node 0				ref	int

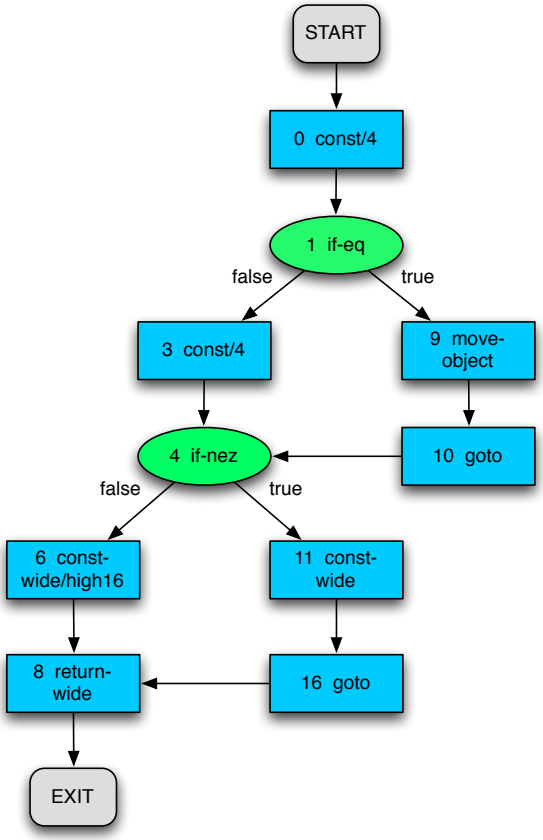
Node 0 is an ambiguous assignment to register v1: `const/4 v1,1`. It could be either an integer or a floating-point constant. This can be resolved by a Depth First Search (DFS) traversal of the CFG looking for a type-exposing opcode. Node 1 takes v1 as a source register; since an `if-eq` instruction can only work with integers or object references, we deduce that v1 is an integer. The updated type state is shown in Table A.2.

```
double return_a_double(int a)
{
  Object x;

    if(a != 1) {
        x = null;
    }
    else {
        x = this;
    }

    if(x == null)
        return 2.5;

    return 1.2;
}
```

(a) Source Code

```
double return_a_double(int)
  0:  const/4     v1,1
  1:  if-eq       v4,v1,9
  3:  const/4     v0,0
  4:  if-nez      v0,11
  6:  const-wide/high16 v1,16388
  8:  return-wide v1
  9:  move-object v0,v3
  10: goto        4
  11: const-wide v1,4608083138725491507
  16: goto        8
```

(b) Dalvik bytecode



(c) Control Flow Graph

Fig. A.1. Method with ambiguous assignment and comparison instructions with its CFG

Table A.2. Type state after determination of an ambiguous type in node 0

Register	v0	v1	v2	v3	v4
Type at node 0		int		ref	int

Types are then propagated to node 1, which does not modify the state (no assignment), as shown in Table A.3.

Table A.3. Type state after type propagation to node 1

Register	v0	v1	v2	v3	v4
Type at node 0		int		ref	int
Type at node 1		int		ref	int

Node 1 is a branching instruction, its successors are nodes 3 and 9. Let us consider the branch which executes node 3. As usual, types are first propagated (see Table A.4).

But node 3 is an ambiguous assignment to register v0: `const/4 v0,0`. It could assign either a null reference, an integer with value 0 or a floating- point constant with value 0. Therefore, we try to determine its type with the usual DFS looking for a type-exposing instruction. First we consider node 4, which compares register v0 with 0. This instruction does not reveal the type of v0, since it can be used either with an integer (comparison to 0) or a reference (comparison to null). Next the DFS continues with node 6 and then node 8. Node 8 is a return instruction, which causes the branch to be abandoned. Then node 11 is considered before node 16. None of them uses v0. Then this branch is abandoned, since the next node (8) has already been visited. Therefore, the type of v0 remains unknown (see Table A.5).

Table A.4. Type state after type propagation to node 3

Register	v0	v1	v2	v3	v4
Type at node 0		int		ref	int
Type at node 1		int		ref	int
Type at node 3		int		ref	int

Table A.5. Type state after attempt to determine an ambiguous type in node 3

Register	v0	v1	v2	v3	v4
Type at node 0		int		ref	int
Type at node 1		int		ref	int
Type at node 3	unknown	int		ref	int

Then we consider node 4 and propagate the current types. Node 4 does not modify the types (no assignment), as shown in Table A.6.

Node 4 has two successors. Let us consider node 6 first. We start with the type propagation (see Table A.7).

Node 6 is an ambiguous assignment to register v1: `const-wide/high16 v1,16388`. It could be used to assign either a long or a double. A DFS traversal considers node 8, which exposes the type of v1: it returns the contents of v1, which can be determined by parsing the return type in the method descriptor. The type is determined to be a double. The updated type state is presented in Table A.8.

Then we consider node 8, which does not change the type state (no assignment), as shown in Table A.9.

This branch has encountered a `return-wide` instruction, therefore it is abandoned. Then we consider the branch which starts at node 11. First, types are propagated to it from node 4 (see Table A.10).

Table A.6. Type state after type propagation to node 4

Register	v0	v1	v2	v3	v4
Type at node 0		int		ref	int
Type at node 1		int		ref	int
Type at node 3	unknown	int		ref	int
Type at node 4	unknown	int		ref	int

Table A.7. Type state after type propagation to node 6

Register	v0	v1	v2	v3	v4
Type at node 0		int		ref	int
Type at node 1		int		ref	int
Type at node 3	unknown	int		ref	int
Type at node 4	unknown	int		ref	int
Type at node 6	unknown	int		ref	int

Table A.8. Type state after determination of an ambiguous type in node 6

Register	v0	v1	v2	v3	v4
Type at node 0		int		ref	int
Type at node 1		int		ref	int
Type at node 3	unknown	int		ref	int
Type at node 4	unknown	int		ref	int
Type at node 6	unknown	double (1st half)	double (2nd half)	ref	int

Table A.9. Type state after type propagation to node 8

Register	v0	v1	v2	v3	v4
Type at node 0		int		ref	int
Type at node 1		int		ref	int
Type at node 3	unknown	int		ref	int
Type at node 4	unknown	int		ref	int
Type at node 6	unknown	double (1st half)	double (2nd half)	ref	int
Type at node 8	unknown	double (1st half)	double (2nd half)	ref	int

Table A.10. Type state after type propagation to node 11

Register	v0	v1	v2	v3	v4
Type at node 0		int		ref	int
Type at node 1		int		ref	int
Type at node 3	unknown	int		ref	int
Type at node 4	unknown	int		ref	int
Type at node 6	unknown	double (1st half)	double (2nd half)	ref	int
Type at node 8	unknown	double (1st half)	double (2nd half)	ref	int
Type at node 11	unknown	double (1st half)	double (2nd half)	ref	int

Node 11 is an ambiguous assignment to register v1: `const-wide v1,4608083138725491507`. As with node 6, the type is determined by finding a type-exposing instruction. The DFS considers node 16 and then node 8, which reveals that v1 is a double. The updated state is presented in Table A.11 (we have included the propagated types for node 16, which does not modify the type state).

Next we consider the branch which starts at node 9. First, types are propagated from node 1 (see Table A.12).

Node 9 is an assignment to register v0: `move-object v0,v3`. This instruction copies the contents of v3 to v0, which means that the type of v0 becomes the same as the type of v3, an object reference (see Table A.13).

Types can then be propagated to node 10 (see Table A.14).

At this point, all the nodes have been considered once. However, the type of v0 could be known in nodes 4, 6, 8, 11, 16. That is why, from node 10, we continue the forward propagation of the type of v0 to nodes 4, 6, 8, 11 and 16. In practice, our implementation only forwards types to the next node if it brings new type information, which allows multiple traversals of a node if necessary. The updated type state is shown in Table A.15.

Table A.11. Type state after determination of an ambiguous type in node 11

Register	v0	v1	v2	v3	v4
Type at node 0		int		ref	int
Type at node 1		int		ref	int
Type at node 3	unknown	int		ref	int
Type at node 4	unknown	int		ref	int
Type at node 6	unknown	double (1st half)	double (2nd half)	ref	int
Type at node 8	unknown	double (1st half)	double (2nd half)	ref	int
Type at node 11	unknown	double (1st half)	double (2nd half)	ref	int
Type at node 16	unknown	double (1st half)	double (2nd half)	ref	int

Table A.12. Type state after type propagation to node 9

Register	v0	v1	v2	v3	v4
Type at node 0		int		ref	int
Type at node 1		int		ref	int
Type at node 3	unknown	int		ref	int
Type at node 4	unknown	int		ref	int
Type at node 6	unknown	double (1st half)	double (2nd half)	ref	int
Type at node 8	unknown	double (1st half)	double (2nd half)	ref	int
Type at node 11	unknown	double (1st half)	double (2nd half)	ref	int
Type at node 16	unknown	double (1st half)	double (2nd half)	ref	int
Type at node 9		int		ref	int

Table A.13. Type state after updating a type in node 9

Register	v0	v1	v2	v3	v4
Type at node 0		int		ref	int
Type at node 1		int		ref	int
Type at node 3	unknown	int		ref	int
Type at node 4	unknown	int		ref	int
Type at node 6	unknown	double (1st half)	double (2nd half)	ref	int
Type at node 8	unknown	double (1st half)	double (2nd half)	ref	int
Type at node 11	unknown	double (1st half)	double (2nd half)	ref	int
Type at node 16	unknown	double (1st half)	double (2nd half)	ref	int
Type at node 9	ref	int		ref	int

Table A.14. Type state after type propagation to node 10

Register	v0	v1	v2	v3	v4
Type at node 0		int		ref	int
Type at node 1		int		ref	int
Type at node 3	unknown	int		ref	int
Type at node 4	unknown	int		ref	int
Type at node 6	unknown	double (1st half)	double (2nd half)	ref	int
Type at node 8	unknown	double (1st half)	double (2nd half)	ref	int
Type at node 11	unknown	double (1st half)	double (2nd half)	ref	int
Type at node 16	unknown	double (1st half)	double (2nd half)	ref	int
Type at node 9	ref	int		ref	int
Type at node 10	ref	int		ref	int

Table A.15. Type state after propagation of the newly determined type of v0

Register	v0	v1	v2	v3	v4
Type at node 0		int		ref	int
Type at node 1		int		ref	int
Type at node 3	unknown	int		ref	int
Type at node 4	ref	int		ref	int
Type at node 6	ref	double (1st half)	double (2nd half)	ref	int
Type at node 8	ref	double (1st half)	double (2nd half)	ref	int
Type at node 11	ref	double (1st half)	double (2nd half)	ref	int
Type at node 16	ref	double (1st half)	double (2nd half)	ref	int
Type at node 9	ref	int		ref	int
Type at node 10	ref	int		ref	int

A.2 Second Pass

There is still one unknown type, which is the type of v0 in node 3. In a second pass, we can determine the type of v0 in node 3 by doing the usual DFS, looking for a type-revealing instruction. Considering node 4 reveals the type: we now know that v0 is used as an object reference as a source register for node 4.

The final type state is presented in Table A.16. All the ambiguous types have been successfully determined.

Table A.16. Final type state

Register	v0	v1	v2	v3	v4
Type at node 0		int		ref	int
Type at node 1		int		ref	int
Type at node 3	ref	int		ref	int
Type at node 4	ref	int		ref	int
Type at node 6	ref	double (1st half)	double (2nd half)	ref	int
Type at node 8	ref	double (1st half)	double (2nd half)	ref	int
Type at node 11	ref	double (1st half)	double (2nd half)	ref	int
Type at node 16	ref	double (1st half)	double (2nd half)	ref	int
Type at node 9	ref	int		ref	int
Type at node 10	ref	int		ref	int

References

[1] Apple iphone - mobile phone, ipod, and internet device. http://www.apple.com/iphone/.

[2] Artho software - jlint. http://artho.com/jlint/.

[3] Checkstyle. http://checkstyle.sourceforge.net/.

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

[5] Fernflower - java decompiler. http://www.reversed-java.com/fernflower/.

[6] Findbugs - find bugs in java programs. http://findbugs.sourceforge.net/.

[7] Fortify software: Fortify warns that smartphone app developers must embrace secure code development strategies. http://4g-wirelessevolution.tmcnet.com/news/2010/03/11/4666957.htm.

[8] Jad - the fast java decompiler. http://www.kpdus.com/jad.html.

[9] Jd java decompiler. http://java.decompiler.free.fr/.

[10] Lint4j. http://www.jutils.com/.

[11] Mocha, the java decompiler. http://www.brouhaha.com/ eric/software/mocha/.

[12] Pmd. http://pmd.sourceforge.net/.

[13] Pmd - rule set: Security code guidelines. http://pmd.sourceforge.net/rules/sunsecure.html.

[14] Qj-pro - code analyzer for java. http://qjpro.sourceforge.net/.

[15] Secure coding guidelines for the java programming language, version 3.0 - sun developer network. http://java.sun.com/security/seccodeguide.html.

[16] Secure your java apps from end to end, part 2 - javaworld. http://www.javaworld.com/javaworld/jw-07-2001/jw-0713-howto.html?page=3.

[17] Sloccount. http://www.dwheeler.com/sloccount/.

[18] Undx, a reconstructor for dalvik bytecode. http://www.illegalaccess.org/undx/.

[19] Yasca - yet another source code analyzer. http://www.yasca.org/.

[20] Ben Bellamy, Pavel Avgustinov, Oege de Moor, and Damien Sereni. Efficient local type inference. In Gregor Kiczales, editor, *OOPSLA'07: 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2008.

[21] Dan Bornstein. Google i/o 2008 - dalvik virtual machine internals. http://www.youtube.com/watch?v=ptjedOZEXPM.

[22] E. Chaillou, P. Manoury, and B. Pagano. *Developing Applications with Objective Caml*. O'Reilly, France, 2000. Available from http://caml.inria.fr/pub/docs/oreilly-book/index.html.

[23] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of c code. In *Proc. of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, 2004.

[24] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security and Privacy*, 2(6):76–79, 2004.

[25] Cristina Cifuentes, Trent Waddington, and Mike Van Emmerik. Computer security analysis through decompilation and high-level debugging. In *Proc. of the Workshop on Decompilation Techniques*, pages 375–380. IEEE Press, 2001.

[26] Graham Cluley. First iphone worm discovered - ikee changes wallpaper to rick astley photo, November 2009. http://www.sophos.com/blogs/gc/g/2009/11/08/iphone-worm-discovered-wallpaper-rick-astley-photo/.

[27] William Enck, Machigar Ongtang, and Patrick McDaniel. On Lightweight Mobile Phone App Certification. In *Proc. of the 16th ACM Conference on Computer and Communications Security (CCS)*, November 2009.

[28] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding android security. *IEEE Security and Privacy*, 7(1):50–57, 2009.

[29] Etienne Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for java bytecode. In *SAS '00: Proc. of the 7th International Symposium on Static Analysis*, pages 199–219. Springer-Verlag, 2000.

[30] Dan Goodin. Backdoor in top iphone games stole user data, suit claims. The Register, November 2009. http://www.theregister.co.uk/2009/11/06/iphone_games_storm8_lawsuit/.

[31] Charlie Lai. Java insecurity: Accounting for subtleties that can compromise code. *IEEE Software*, 25:13–19, 2008.

[32] Xavier Leroy. Java bytecode verification: Algorithms and formalizations. *J. Autom. Reason.*, 30(3-4):235–269, 2003.

[33] Benjamin Livshits. *Improving software security with precise static and runtime analysis.* PhD thesis, Stanford, CA, USA, 2006. Adviser-Lam, Monica.

[34] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proc. of the 14th conference on USENIX Security Symposium.* USENIX Association, 2005.

[35] Jerome Miecznikowski and Laurie Hendren. Decompiling java using staged encapsulation. In *WCRE '01: Proc. of the Eighth Working Conference on Reverse Engineering*, page 368. IEEE Computer Society, 2001.

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

[37] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[38] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *Proc. of the 8th European Symposium on Programming Languages and Systems*, pages 208–223. Springer-Verlag, 1999.

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

[40] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically Rich Application-Centric Security in Android. In *Proc. of the 25th Annual Computer Security Applications Conference (ACSAC)*, December 2009.

[41] Todd A. Proebsting and Scott A. Watterson. Krakatoa: Decompilation in java (does bytecode reveal source?). In *Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, 1997.

[42] Benjamin Schwarz, Hao Chen, David Wagner, Jeremy Lin, Wei Tu, Geoff Morrison, and Jacob West. Model checking an entire linux distribution for security violations. In *ACSAC '05: Proc. of the 21st Annual Computer Security Applications Conference*, pages 13–22. IEEE Computer Society, 2005.

[43] Jerzy Tiuryn. Type inference problems: A survey. In *MFCS '90: Proc. of the Mathematical Foundations of Computer Science 1990*, pages 105–120. Springer-Verlag, 1990.

[44] Raja Vallee-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *International Conference on Compiler Construction, LNCS 1781*, pages 18–34, 2000.

[45] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly & Associates, Inc., 2000.

[46] Michael S. Ware and Christopher J. Fox. Securing java code: heuristics and an evaluation of static analysis tools. In *SAW '08: Proc. of the 2008 workshop on Static analysis*, pages 12–21. ACM, 2008.

[47] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng. Notes*, 29(6):97–106, 2004.