Transparent Program Transformations in the Presence of Opaque Code

Eli Tilevich
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061, USA
tilevich@cs.vt.edu

Yannis Smaragdakis
Department of Computer and Information Science
University of Oregon
Eugene, OR 97403, USA
yannis@cs.uoregon.edu

Abstract

User-level indirection is the automatic rewriting of an application to interpose code that gets executed upon program actions such as object field access, method call, object construction, etc. The approach is constrained by the presence of opaque (native) code that cannot be indirected and can invalidate the assumptions of any indirection transformation. In this paper, we demonstrate the problem of employing user-level indirection in the presence of native code. We then suggest reasonable assumptions on the behavior of native code and a simple analysis to compute the constraints they entail. We show that the type information at the native code interface is often a surprisingly sufficient approximation of native behavior for heuristically estimating when user-level indirection can be applied safely. Furthermore, we introduce a new user-level indirection approach that minimizes the constraints imposed by interactions with native code.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming—program synthesis, program transformation, program verification; D2.3 [**Software Engineering**]: Coding Tools and Techniques—Object-oriented programming;

General Terms: Languages.

Keywords: Program transformation, aspect-oriented programming, program enhancement.

1. Introduction and Motivation

User-level indirection is the automatic transformation of application and system code so that its execution characteristics are modified. Typical user-level indirection schemes instrument the code to interpose extra functionality during specific program events—mainly access to object fields, calls of specific methods and object construction. For instance, we may want to add indirection to all changes to the fields of an object for logging: we may want a permanent log of all state updates in a running system. This is possible by finding all field access instructions in the application and modifying them to log their action before taking it. The logging code is either included inline at the field access site, or a separate method can be called.

Standard applications of the approach include transparent distributed execution [3,5,12,14,15,16], persistence [2,9,11], profiling [6], and logging [10]. Additionally, Aspect-Oriented Programming (AOP) [7] has yielded general purpose program enhancement mechanisms that often rely (behind the scenes) on user-level indirection techniques. Compared to the straightforward approach of modifying the runtime system (e.g., the Java VM or the .NET CLR), user-level indirection has the advantage of portability and ease of deployment on unmodified runtime systems. Running applications on modified versions of a platform-specific runtime system is hard and in some cases (e.g., embedded systems) even impossible. Yet, if we achieve the same effect through code transformation, the resulting code can run on many platforms using standard-issue runtimes.

User-level indirection has to be transparent relative to the behavior of the original code. Nevertheless, all user-level indirection techniques have transparency limitations relating to the presence of native code (a.k.a. *platform-specific binary code*) that an application can access. Native code is opaque: it cannot be analyzed or modified without negating the platform-independence advantages of user-level indirection. Yet, native code has its own state, can hold references to user objects, can remember (alias) these references across invocations, and can use them for destructive updates of user-level state. This renders the code transformation incorrect (i.e., non-semantics-preserving) for all user-level indirection techniques in the literature and for most purposes of user-level indirection.

As background for presenting the problem, we first consider how actual user-level indirection schemes deal with system classes. We use the Java platform in our discussion, but the same ideas apply to other high-level runtimes. Typically, one can apply the same user-level indirection techniques to both user-level code and system classes that are supplied in bytecode form. The standard approach is to create a separate, instrumented version of the system classes [4,15,16]. The instrumented version co-exists with the standard system classes in the same application. In this way, an application can access both the user-level indirected versions of system classes and the original versions without any conflict. This is necessary, since the system classes are often used inside the instrumentation code itself. In original application code, however, all uses of system classes are replaced with uses of their instrumented counterparts. Factor, Schuster and Shagin [4] call this the "Twin Class Hierarchy" approach (TCH) and we will use the name for convenience, although the approach pre-existed. As an example, imagine that the original Java application contains code such as:

```
class A
{ public java.lang.String meth(int i, B b) {...} }
```

The rewritten class would use the instrumented class types:

```
class UP.A
{ public UP.java.lang.String meth(int i, UP.B b) {...} }
```

(`UP` in the above code stands for "user package".) Figure 1 shows the effects on the class hierarchies pictorially.

System classes often contain native code, however. Some of the most fundamental system classes (e.g., the ones dealing with threading, file and network access, GUI, etc.) rely on native code, mainly for reasons of low-level resource access, such as context-switching or fast graphical operations. Changing native code

GPCE'06, October 22-26, 2006, Portland, Oregon, USA.

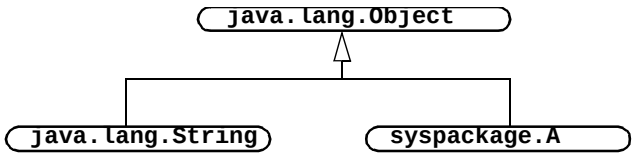

Figure 1(a): Original system classes hierarchy

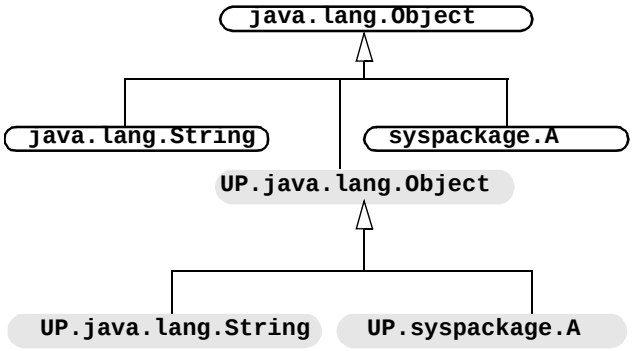

Figure 1(b): Replicating system classes in user package (UP)

requires platform-specific changes and the creation of special versions of the runtime system (either the executable program or its dynamic libraries). Similarly, analyzing native code and relying on its implementation properties is a platform-specific task. Thus, dealing with native code is incompatible with the main motivation for user-level indirection: that of portability and platform independence. Therefore, native code is *opaque* for the purpose of user-level indirection: it cannot be modified or analyzed.

Opaque code presents some obvious limitations for user-level indirection approaches: user-level indirection cannot be used to intercept actions (e.g., field accesses) occurring entirely inside native code. That is, changes to internal system state (e.g., the contents of a low-level window, the scheduling structure of threads, etc.) cannot be intercepted using user-level indirection. E.g., distributed execution systems (such as J-Orchestra [16], Pangaea [14], Addistant [15] or JavaSplit [3]) cannot hope to use the same techniques as for user classes to transparently migrate window or thread objects from one machine to another. Special purpose replacements of this functionality are used instead.

More interestingly, however, the interactions of native code with user-level indirection can be subtle and can affect the correctness of indirection of non-native code. Consider the TCH approach described earlier (Figure 1) for instrumenting standard Java libraries. If a class `A` has a native method, an instrumented version of `A` delegates calls to the native method of an internal `A` object. This technique is used because a native method implementation in Java is bound to a particular class name and cannot be reused for a different class. For instance, consider original code as follows:

```
class File {  ...
  public native void write(byte b);
}
```

The instrumented version of this class would be:

```
class UP.File {  ...
  private File origImpl_;
  // delegate to native method
  public void write(byte b) {origImpl_.write(b);}
}
```

This `UP.File` class cannot use arbitrary user-level indirection even for its non-native methods. Imagine that the `File` class also has a non-native method `newLine`:

```
class File {  ...
  public native void write(byte b);
  public void newLine() { ... }
}
```

It is not safe to indirect method `newLine` (e.g., to track its changes to fields of a `File` object) yet simply delegate method `write`. To see this, consider the re-written code:

```
class UP.File {
  private File origImpl_;
  ...
  // delegate to native method
  public void write(byte b) { origImpl_.write(b); }
  public void newLine() {...} // instrumented body
}
```

The problem is that any call to method `write` affects the `origImpl_` object, while any call to method `newLine` affects the current object of type `UP.File`. The two objects have separate copies of their data fields that get changed, but the two objects were one in the original program. This destroys the transparency of user-level indirection.

Further problems occur because object fields can be read and written inside native code. For instance, consider user-level indirection approaches that capture all updates to fields of an object (e.g., to implement transparent persistence or distributed execution). In this case, all objects that can ever be referenced by native code cannot be fully indirected using user-level indirection techniques. That is, even if an object's class has no native methods, if the object is ever referenced by some other class's native code, then we cannot indirect *all* access to the object's fields. (Although we can intercept the subset of the accesses performed inside platform independent code.)

Furthermore, some restrictions on the use of user-level indirection are caused by the structure of the user-level indirection scheme itself. For instance, consider again the TCH rewrite. Without any special provisions, the limitations on the use of indirection propagate to all subclasses. A subclass `ROFile` of the original `File` class may have no native methods, yet its methods cannot be instrumented. If the instrumentation were performed, the `UP.ROFile` class would be a subclass of `UP.File` and not of `File`. Thus, `UP.ROFile` would not be able to access non-public members of `File`. We later discuss how to remove this limitation.

This paper serves two purposes. First, it describes the limitations of user-level indirection techniques in a general setting. Such limitations often go unnoticed even by experts. For instance, the authors of the TCH approach argue that "TCH can be used automatically by any general instrumentation"; "[TCH has] the ability [...] to instrument all system classes"; "TCH allows even system classes with native dependencies to be rewritten for distributed execution" [4]. Second, our work offers heuristic techniques for recognizing these limitations with high accuracy in practice. These techniques generalize, extend, and refine the ones of our J-Orchestra system [16,17], which automatically rewrites Java applications for distributed execution. The practical benefit of such heuristics is that they give the user guidance on which classes are safe to indirect. Without such guidance, the user would be entirely responsible for the handling of native code. This is not a scalable approach as Saff, Artzi, Perkins and Ernst observe [13]:

> *... [the TCH] approach does not scale. The most serious problem is that wrappers must be written by hand for each native method, of which there are a great many used by any realistic program.*

With our approach, the user rarely needs to explicitly specify special-case handling of native methods.

2. Assumptions and Analysis

To determine which program actions can be safely indirected, we would need to analyze the implementation of native methods. It is

highly complicated to obtain the source code for all platform-specific runtimes, or to require VM implementors to export a model of the behavior of their system just for the purpose of enabling safe user-level indirection. Thus, we examine a more pragmatic approach. We use the type information at the native code interface to derive a "poor-man's native code behavior model". This model is correct only under some (overall reasonable) assumptions on native code behavior.

2.1 Type Analysis + Weak Assumptions

The majority (~97%) of Java system classes (all numbers given are from Sun JDK 1.4.2 unless stated otherwise) have no native methods. Such classes encode useful reusable libraries and not system-level functionality. It is, thus, of broad importance to automatically recognize system classes that do not interact with native code and to support correct user-level indirection for them. In general, this task is impossible without making assumptions regarding native code behavior. For instance, all classes in Java are subclasses of `java.lang.Object`, which has native code. In theory, any native method can be receiving an `Object`-typed argument, discovering its actual type using reflection and performing on the object some action (e.g., reading fields) that would be undetected by any user-level indirection mechanism.

As we discussed earlier, there is no way to detect events (e.g., field writes) that occur entirely inside opaque code. The interesting case, however, is that of events concerning user-level (i.e., non-opaque) entities and the question of whether these can occur inside opaque code. For instance, we may want to capture all updates to an object field that is declared in a Java system class implemented in plain bytecode. We need to ask if this field is ever accessed inside native code. In this section we assume the full gamut of user-level indirection events, including access and modification of fields. For capturing method and constructor calls only, the restrictions are typically far less severe. Nevertheless, most interesting applications of user-level indirection (esp. distributed execution and persistence) need to capture field accesses.

Our previous work on the J-Orchestra system used some weak assumptions on the behavior of native code and a simple type-based analysis to distinguish code that is likely to safely employ user-level indirection. In this way, we can exploit the rich type information of the Java system classes API. J-Orchestra uses user-level indirection in order to execute monolithic Java applications over a network of machines. Typically, the application is split in parts (consisting of user code and system classes) so that each machine handles a different hardware or software resource—e.g., the graphical input/output code may run on one machine, while the processing is done on a second and database access on a third.

Here we abstract away the specifics of the J-Orchestra approach so that it can be generalized to different domains. The approach makes two main heuristic assumptions regarding system classes:

- Classes without native methods have no special semantics. (Native code never treats their objects any differently from user-defined objects.)
- Native methods do not use dynamic type discovery (reflection, downcasting, or any low-level type information recovery) on objects supplied through method arguments.

These assumptions generally hold true with only few exceptions. The first assumption does not hold, for instance, for classes in the `java.lang.ref` package. The second assumption does not hold in the implementation of reflection classes themselves. In Section 3 we discuss a study of the Sun implementation of Java system classes and how it supports our assumptions.

The first assumption essentially states that the JVM is not allowed to handle different types of objects specially when the objects just use plain bytecode instructions. For instance, the JVM is not allowed to detect the construction of an object of a "special" type and keep a reference to this object that native code can later use for destructive state updates. This is a reasonable assumption, conforming to good software design practices. The second assumption states that native code is strongly typed: if a reference is declared to be of type `T`, it can never be used to access fields (method calls are fine) of a subclass of `T`. For instance, the assumption prohibits native methods from taking an `Object`-typed argument, checking if it is actually of a more specific type (e.g., `Thread` or `Window`), casting the object to that type and directly accessing fields or methods defined by the more specific type. This assumption also encodes a good design practice: code exploits the static type system as much as possible for correctness checking. Although the assumption may be violated locally, the hope is that it is rarely violated over the bytecode-native code boundary.

With the above two assumptions, we can perform a classification of Java system classes with respect to whether they can employ user-level indirection transparently or not, based on their usage of native code. We will use the term *NUI* (for *non-user-indirectible*) to describe classes that cannot employ user-level indirection transparently. The base J-Orchestra rules for inferring the classes that have user-level indirection limitations are as follows:

1) A system class with native methods is NUI.
2) A system class used as a parameter or return type for a method or static method in a NUI class is NUI.
3) If a system class is NUI, then all class types of its fields or static fields are NUI.
4) If a system class, other than `java.lang.Object`, is NUI, then its subclasses and superclasses are NUI.

(The above rules represent the essence of the analysis but not its entirety. For instance, we do not discuss arrays or exceptions—these are handled similarly to regular classes holding references to the array element type and method return types, respectively. The numbers we later report are for the full version of the rules, however. Note that interface access does not impose restrictions since an interface cannot be used to directly access state.)

Rule 1 above is justified because no user-indirection technique can guarantee to capture all field updates of an instance of a class with a native method. The native method can always perform updates without any indirection.

Rule 2 is justified with a similar argument: if an object can be passed to native code, native code can alias it and (either during the native method execution or during a later invocation) change its state. Furthermore, the rule can be applied transitively: if a class is NUI then we cannot replace all its uses with uses of an instrumented version in a user package `UP`. Then all objects used as arguments of any method (even non-native) may have their fields accessed directly.

Rule 3 is like Rule 2 but for fields: native code can access objects transitively reachable from an object that leaks to native code.

Rule 4 is justified by the specifics of the aforementioned user-level indirection scheme. We saw an instance of this restriction earlier: if a class cannot be indirected, its uses in the application cannot instead employ a modified copy of the class in a user-level package. Thus, all subclasses and superclasses also cannot be copied to a user level package, as they may need to access non-public fields of their superclass.

These rules enable user-level indirection to be used safely for many Java system classes. Specifically, 37% of the Java system classes are classified as having no dependencies to native code and, thus, being able to employ user-level indirection safely. Still, however, these rules are too conservative, as 63% of the system

classes are deemed non-indirectible. Nevertheless, the rules are a good starting point and can be weakened to be made practical for specific applications of user-level indirection. For instance, in the context of J-Orchestra, one more assumption is made relating to the way native code in different libraries can share state. The extra assumption allows placing different pieces of native code on separate machines [16].

Next, we show one important general-purpose weakening of the rules. Rules 2 and 4 can be weakened significantly if we are allowed to modify system packages (still without touching native code) and we employ a more sophisticated user-level indirection scheme than that of J-Orchestra or TCH.

2.2 More Sophisticated User-Level Indirection

The rules of the previous section are conservative because they assume that all code in system packages (be it native or not) is opaque. See, for instance, Rule 2: although any object that is used as a parameter of a native method can have its fields accessed with no indirection, there is no need to recursively propagate this constraint to the non-native methods of this object as well. If the object class is in pure bytecode, we can edit it and introduce indirection for accesses to its parameters. This, however, relies on a low-level assumption: we assume that the user-level indirection technique can modify system packages in order to edit the bytecode of existing system classes or add a new class in a system package. This is undesirable in some settings because it requires control over the startup environment of the JVM. Such control is not always possible, e.g., for deploying applets that random users will download and use inside a browser, or in systems in which the user cannot modify or extend the system package for security.

Under this assumption, we can weaken Rules 2 and 4.

1) A system class with native methods is NUI.

2') A system class used as a parameter or return type for a native method is NUI.

3) If a system class is NUI, then all class types of its fields or static fields are NUI.

4') If a system class is NUI, then its superclasses are NUI.

The weaker rules push the limits of user-level indirection further: fewer than 8% of the Java system classes are classified as unable to employ user-level indirection (i.e., NUI). This means that a general-purpose user-level indirection technique can apply to more than 92% of the Java system classes with no special handling.

We already discussed how the new version of Rule 2 is a result of instrumenting the bytecode of bytecode-only NUI classes. The weakening of Rule 4 is more interesting. In the new Rule 4, a class does not impose any restrictions on its subclasses. This also eliminates any special handling of the `java.lang.Object` class, which is a common singularity in user-level indirection schemes.

To use the weaker version of Rule 4, we need to make sure that every system class `C` that cannot employ user-level indirection transparently is replicated in a user-level package. The replica class will just delegate all method calls to the original. Subclasses of `C` that have no native dependencies will employ full user-level indirection: an instrumented copy will be created in a user package and all references to the original class will become references to the instrumented version. The problem is that the instrumented class will not be able to access non-public members of `C`, as it is not in the same package as `C`. One solution is to make public all non-public members of class `C` by editing the class bytecode. (Or, equivalently, to create a subclass of `C` that exports the non-public members of `C`—see later.) A safer approach would be to emulate the Java access control at run-time using a technique such as that proposed by Bhowmik and Pugh [1] for the Java inner classes rewrite. At load time, class `C` creates a secret key and passes it to the instrumented version of its subclass. When objects of the instrumented class need to access `C` members, they call a public method that also receives and checks the secret key. This is a safe emulation of the Java access protection, yet it avoids the need to place classes in the same package.

```
class File {
 SomeT field1;
 ...
 public native void write(byte b);
 public void newLine() {...}
}

class TXFile extends File {
 ...
 public void writeString(String s) {...foo(field1)...}
}
```

Figure 2(a): Original system class File (with a native method) and subclass TXFile (without native dependencies).

```
class File {
 SomeT field1;
 // Allow free access to field1 only
 // to class UP.File (and children)
 private static final Object key_ = new Object();
 static { UP.File.setKey (key_); }
 public SomeT get_field1(Object key) {
   if (key!= key_) throw new IllegalAccessException();
   return field1;
 }
 ...
 public native void write(byte b);
 public void newLine() {...}
}

//Just delegates to File.
//Only used for correct subtype hierarchy.
class UP.File {
 protected File origImpl_;
 protected static Object  key_;
 public static void setKey(Object key) {  key_ = key; }
 ...
 // delegate to native method
 public void write(byte b) { origImpl_.write(b); }
 public void newLine() { origImpl_.newLine(); }
}

class UP.TXFile extends UP.File {
 ...
 // methods of this class can employ any
 // user-level indirection scheme
 public void writeString(String s)
 { ...foo(origImpl_.get_field1(key_))... }
}
```

Figure 2(b). Result of the indirection transformation, with safe access to non-public fields of class File.

An example application of this technique is shown in Figure 2. The example class `File` of Section 1 is now shown with a non-public field `field1`. `File` has a subclass `TXFile` with no native dependencies. Figure 2(b) shows the transformed classes so that `UP.File` and `UP.TXFile` can correctly replace all uses of `File` and `TXFile`, respectively, yet `UP.TXFile` can employ fully transparent user-level indirection. (As a low-level note, this transformation means that the instrumented system package, `UP`, needs to be loaded by the bootstrap class loader, since there is a call to method `UP.File.setKey` inside the `File` system class.) The effects of the transformation on the example class hierarchy are shown pictorially in Figure 3.

3. Validation

We validate the assumptions and analysis of the previous section in three ways: first we measure how many of the system classes used in actual Java applications are classified as safely indirectible

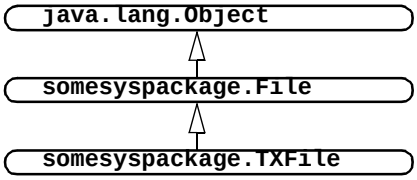

Figure 3(a): A File class hierarchy

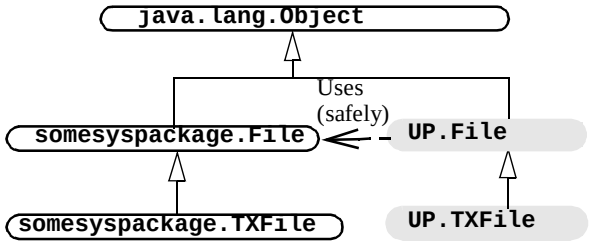

Figure 3(b): Removing subclassing restrictions

under our analysis. Next we examine by code inspection an actual native code implementation of system methods and check whether it satisfies our assumptions. Finally, we perform a dynamic analysis of the applications and show that they do not violate the results of our type-based analysis during their execution.

3.1 Impact on Real Applications

Table 1 shows how many of the system classes actually used by different Java applications are classified as NUI under our analysis of Section 2.2. The table also shows how many of the used system classes have native methods themselves—this is a lower bound on the number of NUI classes under any analysis. (We find the used classes by dynamically observing the loaded classes, minus JVM bootstrap classes. We then run our type-based analysis with the set of used classes as a universe set—any NUI dependencies introduced by classes that were not loaded are ignored.)

Three of the applications (javac, jess, mpegaudio) are standard benchmarks from SPEC JVM'98. (The rest of the SPEC JVM'98 programs yield practically identical numbers.) Unsurprisingly, these benchmarks are old and exercise few of the Java system classes. Nevertheless, we still see that more than 62% of the system classes used can employ user-level indirection. The next seven applications (antlr, bloat, chart, hsqldb, jython, ps, xalan) are from the more modern DaCapo benchmark suite (version beta050224). These applications are more realistic, yet they still do not exercise a large part of the Java system libraries. We see that our analysis enables 66-85% of the system classes used in the DaCapo benchmark programs to be safely indirected. (The DaCapo suite has 3 more applications that we did not manage to run by paper submission time due to setup issues, such as library dependencies or unclear input files.) For applications that exercise more of the Java system classes, we examined the Sun demo application SwingSet2 and the JBits FPGA simulator by Xilinx. The inputs used for these two applications were interactive and consisted of navigating extensively through the application's GUI and performing standard program actions (e.g., loading a simulator and an FPGA configuration and performing simulation steps). Both of these applications exercise over 1400 Java system classes. Only 21 and 16% (for JBits and SwingSet2, respectively) of these classes were found to be NUI under our analysis: the rest can employ user-level indirection without any special treatment. Finally, we include in our suite the RMIServer sample application from Sun, in order to exercise networking system classes.

Thus, Table 1 confirms that native code is not a negligible part of real applications. Additionally, although the type analysis assumes the most general native code behavior that respects its assumptions, it is still sufficient for enabling safe indirection for the large majority of Java system classes used in actual programs.

Table 1. Type-based analysis of used system classes

Application	#classes	#native	%native	#NUI	%NUI
javac	167	21	13	62	37
jess	165	21	13	61	37
Mpeg audio	158	21	13	60	38
Antlr	209	21	10	67	32
Bloat	275	25	9	80	29
Chart	601	69	11	194	32
Hsqldb	295	26	9	83	28
Jython	263	20	8	76	29
Ps	175	18	10	60	34
Xalan	505	21	4	74	15
SwingSet2	1887	120	6	303	16
JBits	1442	124	9	306	21
RMI Server	415	37	9	109	26

3.2 Accuracy of Type Information

Recall that one of the heuristic assumptions of our type-based analysis is that the APIs to system functionality offer accurate type information. That is, if a native method signature refers to type `A`, then it does not attempt to dynamically discover which particular subtype of `A` is the actual type of the object and to use fields or methods specific to that subtype. It is certainly common to pass instances of subtypes of `A` to the native method, but these should only be accessed using the general interface defined by the supertype `A`. This assumption is in line with good OO design.

Although the assumption is soundly motivated, there are certainly exceptions in real code. To validate the assumption, we examined part of the implementation of native methods in Sun's JDK 1.4.2. We searched for the use of specific idioms throughout native method implementations and we examined in detail all native methods (109 of them) accepting as argument or returning as result an object with declared type `java.lang.Object` (the root of the Java inheritance hierarchy). In our study, we observed few violations of our assumptions. The most important ones are:

- reflection functionality routinely circumvents the type system, as expected. Reflection requires special handling.
- passing primitive arrays to native code is typically invisible to the type system. Several native methods accept an `Object` reference but implicitly assume that they are really passed a Java array of bytes or integers. This does not affect our analysis, as we consider primitive types and their arrays to be non-indirectible.
- a handful of methods have poor type information and violate our type accuracy assumptions. For instance, `socketGetOption` in class `java.net.PlainSocketImpl` takes an `Object` as argument, casts it into a `java.net.InetAddress` and then sets one of its fields. (The `addr` field is set when the method returns the bind address for its socket implementation.) Similarly, native method `getPrivateKey` in class `sun.awt.SunToolkit` assumes that its `Object` argument is really a `java.awt.Component` or a `java.awt.MenuComponent` and dynamically discovers its actual type.

  Nevertheless, overall, native code rarely uses dynamic type discovery. A quick search of all native code in Java system libraries (for all platforms together) reveals just 69 uses of the JNI function `IsInstanceOf`, which is the main way to do dynamic type discovery in native code. For comparison, there are about 5900 uses of the Java counterpart, `instanceof`, in plain Java

code in the system libraries. (The total size of Java code in system libraries is roughly twice the size of C/C++ native code, so the discrepancy is not justified by the code size.)

3.3 Testing Correctness

Our type-based analysis attempts a heuristic solution to an unsolvable problem, as mentioned in Section 2.1. If we treat native code as an adversary, there are no safe assumptions we can make, other than "all native code can directly access and modify all objects". Nevertheless, in practice our heuristic, type-based approach works well, as our past experience with J-Orchestra suggested. To see this in a controlled experiment, we dynamically analyzed the applications of Section 3.1. We instrumented a Java VM to observe all reads and writes to object fields performed inside native code. Then we checked whether fields of a class that we did not consider NUI are ever read or written inside native code. Of course, this experiment is just a test under specific inputs. Our analysis results could still be violated by different program inputs. Nevertheless, given the amount and variety of tested code and inputs, we have confidence in our observations.

Almost all applications listed in Table 1 exhibit accesses to Java object fields from inside native code. Some applications (especially the more graphics-intensive ones) have native code access the fields of objects of more than 50 different classes. Throughout all executions of the applications, we observed only two instances of access inside native code to objects of types that were not classified as NUI.

Specifically, in class `sun.awt.font.NativeFontWrapper`, method `populateGlyphVector` (not a user-accessible class) accepts a `java.awt.font.GlyphVector` parameter but implicitly assumes that the actual type of the parameter is `sun.awt.font.StandardGlyphVector` and proceeds to set specific fields of that class. We could not discern a good reason for obscuring this type information from the method's type signatures. (Upon inspection, a couple of more methods in the same class also circumvent the type system for `GlyphVector` arguments.)

The second case was that of the constructor of class `sun.java2d.loops.MaskFill`. The constructor accepts a `java.awt.Composite` parameter but assumes its real type is `java.awt.AlphaComposite`. Although this is again a bad practice of obscuring type information, in this case there is a code economy benefit from doing so: the constructor is only called in native code using dynamic method discovery (i.e., reflection at the native level). Eliding the specific type information allows the constructor to be called by the same code as some other similar constructors.

In summary, our experience confirms that a type-based analysis is quite safe in practice. In the absence of complete information on the behavior of native code, our analysis is a clear win. The only general-purpose alternatives are to either not support indirection for any system classes, or to leave the user with no assistance in determining the correctness of applying indirection.

4. Further Topics

There are several interesting issues that we have not discussed above due to lack of space. These include:

- How the analysis information can be reported to the user in practice. The J-Orchestra GUI environment is an example.
- What features (e.g., reflection, multithreading) typically cannot be handled transparently using a general-purpose technique and require a domain-specific treatment.
- How our work applies to a general aspect language like AspectJ.
- How native code is treated differently in the .NET CLR, compared to the Java VM, which we assumed in our discussion.
- How we can get fully accurate analysis by using a model of the behavior of native code supplied by VM vendors.

The extended version of our paper discusses these topics more.

Acknowledgments

Yannis Smaragdakis performed this work while at the Georgia Institute of Technology. The work is supported by the NSF under Grants No. CCR-0220248 and CCR-0238289.

References

[1] Anasua Bhowmik and William Pugh, "A Secure Implementation of Java Inner Classes", *PLDI 99 poster session.*

[2] Kumar Brahnmath, Nathaniel Nystrom, Antony Hosking and Quintin Cutts, "Swizzle Barrier Optimizations for Orthogonal Persistence in Java", proc. *8th International Workshop on Persistent Object Systems (POS8) and 3rd International Workshop on Persistence and Java (PJW3)*, 1998.

[3] Michael Factor, Assaf Schuster and Konstantin Shagin, "JavaSplit: A Runtime for Execution of Monolithic Java Programs on Heterogeneous Collections of Commodity Workstations", *2003 International Conference on Cluster Computing (CLUSTER'03).*

[4] Michael Factor, Assaf Schuster and Konstantin Shagin, "Instrumentation of Standard Libraries in Object-Oriented Languages: the Twin Class Hierarchy Approach", *Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, 2004.

[5] Bernhard Haumacher, Jürgen Reuter, Michael Philippsen, "JavaParty: A distributed companion to Java", `http://wwwipd.ira.uka.de/JavaParty/`

[6] Jarle Hulaas and Walter Binder, "Program Transformations for Portable CPU Accounting and Control in Java", Partial Evaluation and Semantics-Based Program Manipulation (*PEPM*), 2004.

[7] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin, "Aspect-Oriented Programming", *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold, "An Overview of AspectJ", *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[9] Gordon Landis, Charles Lamb, Tim Blackman, Sam Haradhvala, Mark Noyes, and Dan Weinreb, "ObjectStore/PSE: a Persistent Storage Engine for Java", proc. *2nd International Workshop on Persistence and Java (PJW2)*, p. 129-137, 1997.

[10] Han B. Lee and Benjamin G. Zorn, "Bytecode Instrumentation as an Aid in Understanding the Behavior of Java Persistent Stores", *OOPSLA 1997 Workshop on Garbage Collection and Memory Management.*

[11] ObjectDesign Inc., *ObjectStore PSE/PSE Pro for Java API Guide*, 1999.

[12] Michael Philippsen and Matthias Zenger, "JavaParty - Transparent Remote Objects in Java", *Concurrency: Practice and Experience*, 9(11):1125-1242, 1997.

[13] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst, "Automatic Test Factoring for Java", International Conference on *Automated Software Engineering (ASE)*, 2005.

[14] Andre Spiegel, "Pangaea: An Automatic Distribution Front-End for Java", 4th *IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '99)*, April 1999.

[15] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano, "A Bytecode Translator for Distributed Execution of 'Legacy' Java Software", *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[16] Eli Tilevich and Yannis Smaragdakis, "J-Orchestra: Automatic Java Application Partitioning", *European Conference on Object-Oriented Programming (ECOOP)*, 2002.

[17] Eli Tilevich and Yannis Smaragdakis, "Portable and Efficient Distributed Threads for Java", *5th International Middleware Conference (Middleware'04).*