Obfuscation of Design Intent in Object-Oriented Applications[*]

Mikhail Sosonkin, Gleb Naumovich, and Nasir Memon

Department of Computer and Information Science

Polytechnic University

5 MetroTech Center

Brooklyn, NY 11201

mike@isis.poly.edu, gleb@poly.edu, memon@poly.edu

Abstract

Protection of digital data from unauthorized access is of paramount importance. In the past several years, much research has concentrated on protecting data from the standpoint of confidentiality, integrity and availability. Software is a form of data with unique properties and its protection poses unique challenges. First, software can be reverse engineered, which may result in stolen intellectual property. Second, software can be altered with the intent of performing operations this software must not be allowed to perform.

With commercial software increasingly distributed in forms from which source code can be easily extracted, such as Java bytecodes, reverse engineering has become easier than ever. Obfuscation techniques have been proposed to impede illegal reverse engineers. Obfuscations are program transformations that preserve the program functionality while obscuring the code, thereby protecting the program against reverse engineering. Unfortunately, the existing obfuscation techniques are limited to obscuring variable names, transformations of local control flow, and obscuring expressions using variables of primitive types. In this paper, we propose obfuscations of design of object-oriented programs.

We describe three techniques for obfuscation of program design. The class coalescing obfuscation replaces several classes with a single class. The class splitting obfuscation replaces a single class with multiple classes, each responsible for a part of the functionality of the original class. The type hiding obfuscation uses the mechanism of interfaces in Java to obscure the types of objects manipulated by the program. We show the results of our initial experiments with a prototype implementation of these techniques.

In particular, we show that the runtime overheads of these obfuscations tend to be small.

Categories and Subject Descriptors

D.1.5 [**Object-oriented Programming**]; D.2.2 [**Design Tools and Techniques**]: Object-oriented design methods; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering; D.3.3 [**Language Constructs and Features**]: Classes and objects

General Terms

Design, Security, Languages

Keywords

Software Obfuscation, Refactoring, Code Generation

1. INTRODUCTION

Computer software is a valuable asset as an enormous amount of time, money and intellectual capital is involved in its production. However, once produced, software is vulnerable to theft and misuse. It is estimated that tens of billion dollars of revenue is lost by the software industry due to software piracy alone. With the advent of Internet appliances, mobile code, and untethered and globally pervasive access to the Internet, the problem of software misuse threatens to become a serious menace. Piracy is no longer the only issue. Software tampering with the malicious intent of planting a Trojan horse in the end user's system, for example, suddenly becomes a frightening possibility. Furthermore, since mobile code is often distributed in architecture-independent formats that essentially contains the source code, it is susceptible to decompilation and reverse engineering. Examples of application domains where reverse engineering of software is potentially harmful include:

- *Home entertainment appliances.* Such appliances often contain software that places restrictions on the use of hardware. For example, in a set-top box, software may prevent users from watching a video-on-demand program more than once or from recording it.

- *Grid computing.* Distributed applications, such as SETI-at-home, that utilize unused computing resources on many networked machines are vulnerable to reverse engineering. For example, malicious owners of a machine used in grid computing may alter the part of the application running on their machine with the goal of corrupting or affecting in subtle ways the results computed by the whole application.

- *Copy protection mechanisms.* Various copy protection mechanisms, ranging from license files to dedicated hardware dongles, are in wide use in software industry. Reverse engineering attacks against such mechanisms aim to detect the place in the code of the program where protection mechanisms are used and then either remove this protection code or patch around it.

- *Applications with Proprietary algorithms.* Often, companies that distribute software do not want to make algorithms and designs used in their programs widely known.

There have been different approaches (mostly by industry) employed in the past to prevent reverse engineering and/or tampering of software. One approach that is receiving increasing attention is *software obfuscation* [7]. Obfuscations are program transformations that preserve the functionality of software while changing the structure and look of its source code. There are three types of obfuscation techniques that have been proposed in the literature. First and the most common type is *control flow obfuscation*. Such techniques concentrate on obscuring the flow of control and purpose of variables in individual procedures. For example, an obfuscation may change names of variables in the program or insert *opaque predicates* [7], such as an `if` statement that evaluates to `true` on all program executions. The second type of obfuscation techniques, *data obfuscations*, make understanding of the purpose of data fields in the program difficult. For example, a single integer variable in a program can be replaced with two integer variables, in such a way that the value of the original variable at any point in the program can be obtained by adding the values of the two new variables at this point. Finally, the third type of obfuscation techniques, *layout obfuscations*, obscure the logic inherent in breaking a program into procedures. For example, the code in a procedure can be inlined in all places in the code from which this procedure is called. Similarly, an arbitrary part of code can be out-lined to become a separate procedure. It should be noted that optimizing compilers routinely perform in-lining of small, frequently called procedures, thereby providing a measure of obfuscation.

In this paper we present a new class of obfuscation techniques, which we call *design obfuscations.* We believe that in practice currently available obfuscation techniques are not sufficient, since high-level program constructs can reveal the design of the software and thereby greatly facilitate understanding. Hence, we propose three new obfuscation techniques directed at obscuring the class-level design of object-oriented programs: class coalescing, class splitting, and type hiding obfuscations.

Class coalescing obfuscation is a program transformation that replaces two or more classes in the program with a single class. At its extreme, this obfuscation can replace all classes used in the program by a single class, in a sense, replacing an OO program with a non-OO, procedural program.

Class splitting obfuscation is a program transformation that replaces a class in the program with a number of classes. There are important decisions that have to be made when splitting a class. For each method and field of the original class, one or more of the resulting classes to contain them has to be chosen. We first present a general approach and then describe one specific way in which our prototype obfuscation tool currently splits classes. Used in tandem with the class coalescing obfuscation, this obfuscation can change the program structure very significantly.

Finally, *type hiding* obfuscation uses the concept of Java interfaces to obscure the design intent. In Java, interfaces are used as lightweight type definitions separated from their implementations. We use this facility for obfuscating the type of variables used in the program. We introduce a (potentially large) number of interfaces that are implemented by the existing classes. As a result, for a reverse engineer attempting to understand functionality required from specific objects in the program, this task is made more difficult by the multitude of different types, often with different types representing the same object in different locations in the code.

We implemented all three design obfuscation techniques. In this paper, we show experimental results that indicate that application of these techniques rarely results in large run-time slowdown of the obfuscated program. In particular, the class splitting and type hiding obfuscations seem to scale very well. While coalescing a large number of classes can significantly slow down a program, this slowdown may be acceptable in practice for non-real-time applications.

The rest of this paper is organized as follows: In the next section, we survey related work. Section 3 introduces the technical terminology used in later sections. In Sections 4, 5, and 6 we detail the coalescing, splitting, and type hiding obfuscations respectively. Section 8 describes experimental results of applying our prototype obfuscation tool to several Java programs. In Section 9 we conclude and present directions of future work.

2. RELATED WORK

A comprehensive survey of recent work on software obfuscation appears in [6]. This survey concentrates on low-level obfuscations, such as splitting program variables of primitive types and locally altering control flow of a method by inserting control statements whose predicates always evaluate to the same value or by re-ordering statements to decrease locality. Unlike these obfuscations, techniques introduced in this paper obscure higher-level program structures and therefore can be used more efficiently to hide design of the program.

Barak et al. [3] prove a theoretical result about impossibility of a specific notion of obfuscation. This notion is defined as follows. The obfuscated program should be a "virtual black box", that is, anything that can be computed from its code, can also be computed from its input-output behavior. However, for the applications we consider in this paper, such ideal notion of obfuscation is not necessary. While cryptographic applications require a very strong notion of obfuscation, we are interested only in making the job of a reverse engineer more difficult, although not impossible. The realistic goal of software obfuscation is to make reverse engineering of a program uneconomical.

Many program transformations performed by optimizing compilers are obfuscations. For example, *constant propagation* [18] replaces the use of a variable in an expression with a constant if the value of this variable always equals to this constant in this expression. This operation may hide logic behind complex expressions. By themselves, however, compiler optimizations are not sufficient to protect software against reverse engineering adequately. Similarly, procedure inlining is often used in optimizing compilers to remove overhead of frequently called procedures. Inlining can be viewed as an obfuscation, since it complicates understanding of code by removing a high-level abstraction.

Although little theoretical work on software obfuscation is available, there exist a number of obfuscation tools. Unfortunately, the functionality of these tools is limited to obfuscation of variable and class names and, in some cases, obfuscation of control flow.

The closest work to ours is Snelting and Tip's class hierarchy re-engineering [15]. They use concept analysis [11] to classify the usage of visible methods and fields of a class into usage profiles. Given this classification, a user of a tool based on this analysis can decide to replace the class by several classes. This splitting can be

done automatically. While this transformation bears close resemblance to our class splitting obfuscation, the goals of Snelting and Tip's work and our work are completely opposite. While their goal is to improve the design, ours is to obscure the design of a program. Furthermore, our class splitting is not guided by concept analysis and can split the class almost arbitrarily.

In addition to software obfuscation, several other approaches have been proposed for protection of software against misuse and/or theft of intellectual property. *Tamper-proofing* techniques protect software against tampering. Such techniques disable the software if they determine that this software has been changed. Hardware-based tamper-proofing approaches put the software to be protected on hard-to-crack hardware devices, such as eXecute Only Memory (XOM) [12]. It was demonstrated [2] that tamper-resistant hardware is difficult to construct. In addition, widespread use of such hardware may have prohibitive costs. The Trusted Computing Platform Alliance (TCPA, `www.trustedcomputing.org`) initiative aims to define specifications for hardware assisted, OS based, trusted subsystems that would become an integral part of standard computing platforms including the ubiquitous PC platform. While it enjoys endorsement of a number of large software and hardware vendors, it is not clear if the developed standards will be widely adopted.

Many proposed software based tamper-proofing mechanisms rely on the simple idea of computing checksums for a part of the software. Chang and Atallah [5] argue that tamper-proofing techniques should not rely on a single module whose only purpose is to tamper-proof the entire program. Instead, a number of small program units, called *guards*, working together, should protect the program. The importance of this idea is that guards operate in a network, protecting each other from tampering. Therefore, even if an attacker identifies a number of guards and removes or patches around them, the remaining guards will detect this problem and prevent the program from running. Only by removing all guards can an attacker gain the full functionality of the program.

3. TERMINOLOGY

In this section we introduce the common terminology used in this paper. Although some of this terminology is specific to Java, the class coalescing and splitting obfuscations are applicable to other object-oriented languages, with only small modifications.

Let P be a Java program. Let *Classes*(P) and *Interfaces*(P) denote respectively sets of classes and interfaces defined and/or used in P. For any class or interface c, *Methods*(c) denotes the set of methods defined by c. For any class c, *Fields*(c) denotes the set of fields defined by c[1]. Note that *Methods*(c) and *Fields*(c) do not include any methods and fields inherited by c, but only (1) methods and fields that are introduced by c and (2) methods that override those from its superclasses. For example, class A in Figure 1 does not override method `equals` inherited from class `java.lang.Object` that it implicitly extends. Therefore, this method is not included in *Methods*(A).

For the purposes of our obfuscations it is important to differentiate between static and instance methods. We use set *instanceMethods*(P) to contain all instance methods in all classes in program P. All methods not included in this set are static methods. Similarly, it is important to differentiate between public and non-public methods. We use set *publicMethods*(P) to contain all public methods in all classes in program P. All methods not included in this set are non-public methods.

We use a *dependency mapping* to represent the dependency of methods on other methods and fields of the same class. We write $(m_1, m_2) \in$ *depends* if method m_1 from some class depends on method or field m_2 from the same class. Formally, for any $c \in$ *Classes*(P) and any $m, n \in$ *Methods*(c), $(m, n) \in$ *depends* iff m can call n. Similarly for any field $f \in$ *Fields*(c), $(m, f) \in$ *depends* iff m uses f (either defines it or uses its value). We use this notion of dependency for our splitting obfuscation, which has to separate all methods and fields of a class into two or more parts, taking dependency relationships into account.

An important requirement for all obfuscations is that they preserve functional behaviors of the program. This means that if the original program terminates abnormally on some input, the obfuscated program must also terminate abnormally on this input. If the original program reaches a point at which intermediate or final results are computed, these results should be identical for the obfuscated version at this point. This definition is more complicated for programs with non-deterministic behaviors and real-time programs. All our obfuscations preserve functional program behaviors for deterministic programs that have no dependencies on the clock.
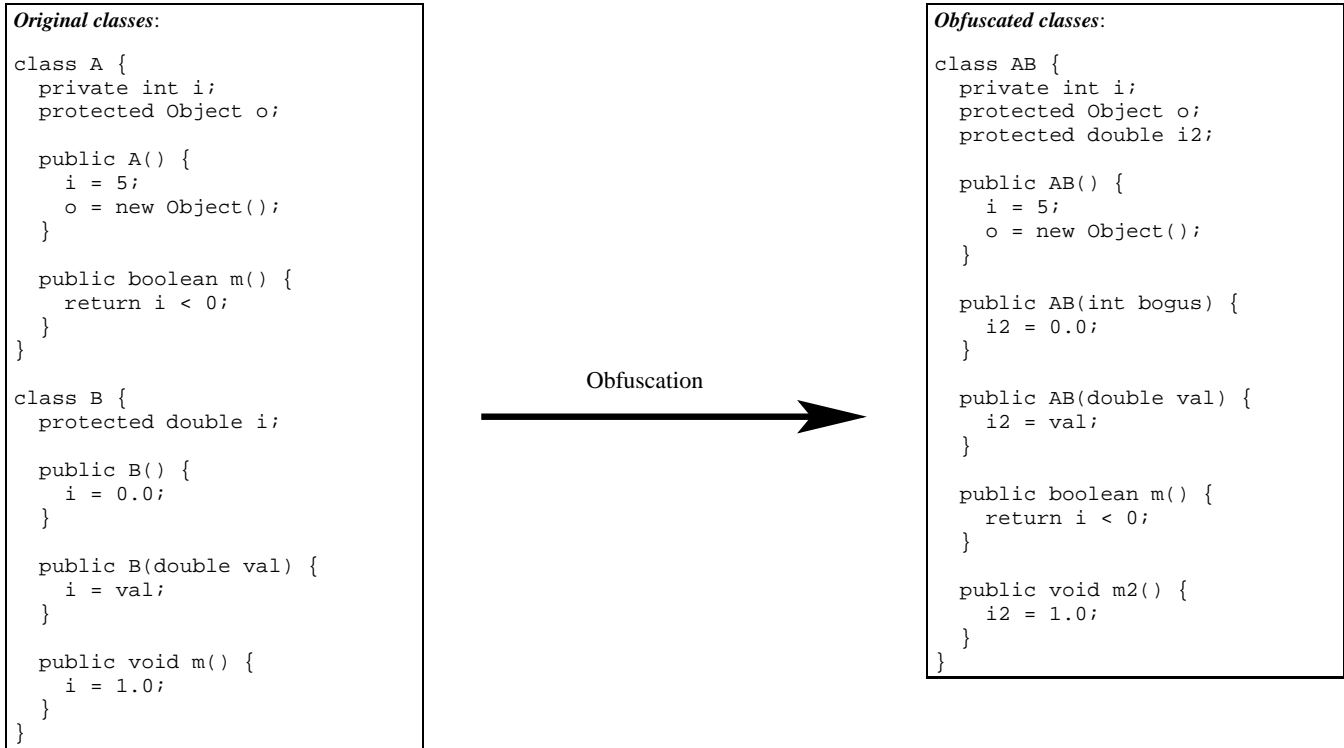
4. CLASS COALESCING

Given two *original* classes c_1 and c_2, the class coalescing obfuscation replaces these classes with a single *target* class c_t. On the high level, we coalesce two classes by combining their fields and methods, renaming these fields and methods as needed. Subsequently, all variable declarations using classes c_1 and c_2 are converted into declarations using c_t and program statements using these variables are modified to ensure that correct fields and methods of c_t are referenced. In the rest of this section we describe these steps in detail. Formally, class coalescing obfuscation can be viewed as building mappings $\mu_f :$ *Fields*$(c_1) \cup$ *Fields*$(c_2) \rightarrow$ *Fields*(c_t) and $\mu_m :$ *Methods*$(c_1) \cup$ *Methods*$(c_2) \rightarrow$ *Methods*(c_t).

Consider first a simple case where c_1 and c_2 are both direct subclasses of class `java.lang.Object`[2], do not implement any interfaces, and do not override any methods from `java.lang.-Object`. In this case, mappings μ_f and μ_m are bijections: we put all fields and methods of c_1 and c_2 into c_t. If c_1 and c_2 have two fields with the same name, we rename one of them uniquely. For example, in Figure 1, classes A and B have identically named fields `i`. In class AB that combines A and B, field `i` coming from B is renamed to `i2`.

Similarly, for non-constructor methods $m_1 \in$ *Methods*(c_1) and $m_2 \in$ *Methods*(c_2) that have the same signature (same name and order and types of arguments), we rename one of m_1 and m_2 in c_t. In Figure 1, method `m` from B became method `m2` in AB. Since constructors cannot be renamed, if c_1 and c_2 have constructors with the same order and types of arguments, we add a bogus argument to one of them in c_t. For example, no-argument constructors of classes A and B in Figure 1 become the no-argument constructor and a constructor with a bogus `int` argument in class AB respectively. Various low-level obfuscation techniques (e.g. opaque predicates [7]) can be used to make these additional arguments appear used by the code. The constructor of B with a `double` argument becomes a constructor of AB without any changes. In future, in order to make this obfuscation more stealthy, we plan to add initialization of all fields of the target class to each constructor of this class; for example, fields `i` and `o` can be initialized to something other than their default initial values in the constructor with a `double` argument in class AB, to prevent a malicious reverse en-

---

[1] Interfaces may only have constant fields, which our obfuscations do not modify.

[2] In Java, all user-defined classes implicitly subclass `java.lang.Object`.

```
Original classes:

class A {
  private int i;
  protected Object o;

  public A() {
    i = 5;
    o = new Object();
  }

  public boolean m() {
    return i < 0;
  }
}

class B {
  protected double i;

  public B() {
    i = 0.0;
  }

  public B(double val) {
    i = val;
  }

  public void m() {
    i = 1.0;
  }
}
```

Obfuscation →

```
Obfuscated classes:

class AB {
  private int i;
  protected Object o;
  protected double i2;

  public AB() {
    i = 5;
    o = new Object();
  }

  public AB(int bogus) {
    i2 = 0.0;
  }

  public AB(double val) {
    i2 = val;
  }

  public boolean m() {
    return i < 0;
  }

  public void m2() {
    i2 = 1.0;
  }
}
```

Figure 1: The basic case of class coalescing

gineer from guessing easily that this constructor was generated by
class coalescing.

Class coalescing becomes more complicated when classes c_1 and
c_2 are involved in inheritance relationships or if these classes im-
plement interfaces. The reason for this is that renaming a method or
adding an extra argument to it is not an option when this method is
defined in a superclass of an original class or interface that an orig-
inal class implements. As an illustration, consider the case where
c_2 extends c_1. In the example in Figure 2, class B is a subclass of
A. The difficulty in coalescing these two classes is combining their
methods m. We cannot rename or add arguments to one of these
methods, because m of B overrides m of A. Our approach is to en-
code the original class of a variable into the state of the new class
of this variable. Figure 2 shows class AB, obtained by coalescing
classes A and B. Note that the AB constructor with no arguments
corresponds to the constructor of A and the AB constructor with
a single int argument corresponds to the constructor of B. Field
isA is introduced to indicate whether an object of type AB is used
in the context where an object of type A was previously used[3]. Note
the use of this field in method m, which combines code from meth-
ods m of both A and B. In this case, mapping μ_m is not bijective,
but only injective.

If original classes are subclasses of the same class or implement
different sets of interfaces, methods from implemented interfaces
and methods that override methods in superclasses cannot be re-
named. In this case, we use a technique similar to that illustrated in
Figure 2.

---

[3]The presence of this field may be a giveaway of the fact that class
coalescing obfuscation was used. Therefore, in practice, local ob-
fuscating transformations can be used to make this less obvious.
Additionally, many different encodings for this information are
possible; varying these encodings will make it more difficult for
attackers to single out such fields.

Under certain conditions imposed by inheritance relationships
among classes, coalescing two classes may require coalescing sev-
eral other classes into the same target class. For example, this may
happen when the two coalesced classes c_1 and c_2 are subclasses of
different classes. Because Java does not support multiple inheri-
tance, we first recursively coalesce superclasses of c_1 and c_2.

After c_t has been generated, we use mappings μ_f and μ_m to
modify the code in all methods in P that use c_1 and c_2 (this may
also include methods in c_t). First, we replace every declaration of
a field, local variable, or method argument of type c_1 or c_2 with
a declaration of type c_t. Second, we replace every reference of a
field $f \in Fields(c_1) \cup Fields(c_2)$ with a reference of field $\mu_f(f)$.
Finally, we replace every call to a method $m \in Methods(c_1) \cup
Methods(c_2)$ with a call to method $\mu_m(m)$. These transformations
are comprised of the well-known object-oriented refactoring tech-
niques *Add Parameter*, *Move Field*, *Move Method*, and *Rename
Method* [9].

If classes c_1 and c_2 extend different classes from Java standard
libraries, we do not coalesce them, since it would require coalesc-
ing library classes and the resulting code would not be portable.
Also, at present we do not coalesce in cases where one of the origi-
nal classes has native methods, since full analysis of such methods
is not possible, and the target class is not guaranteed to have the
same behaviors as the original classes. In Section 8, we show the
percentage of pairs of classes that can be coalesced for each of the
five programs used in our experiments.

5. CLASS SPLITTING

Given an original class c, the class splitting obfuscation replaces
it with two new target classes $c_{t,1}$ and $c_{t,2}$. Since in general a
class can be split into two in a variety of different ways, we use
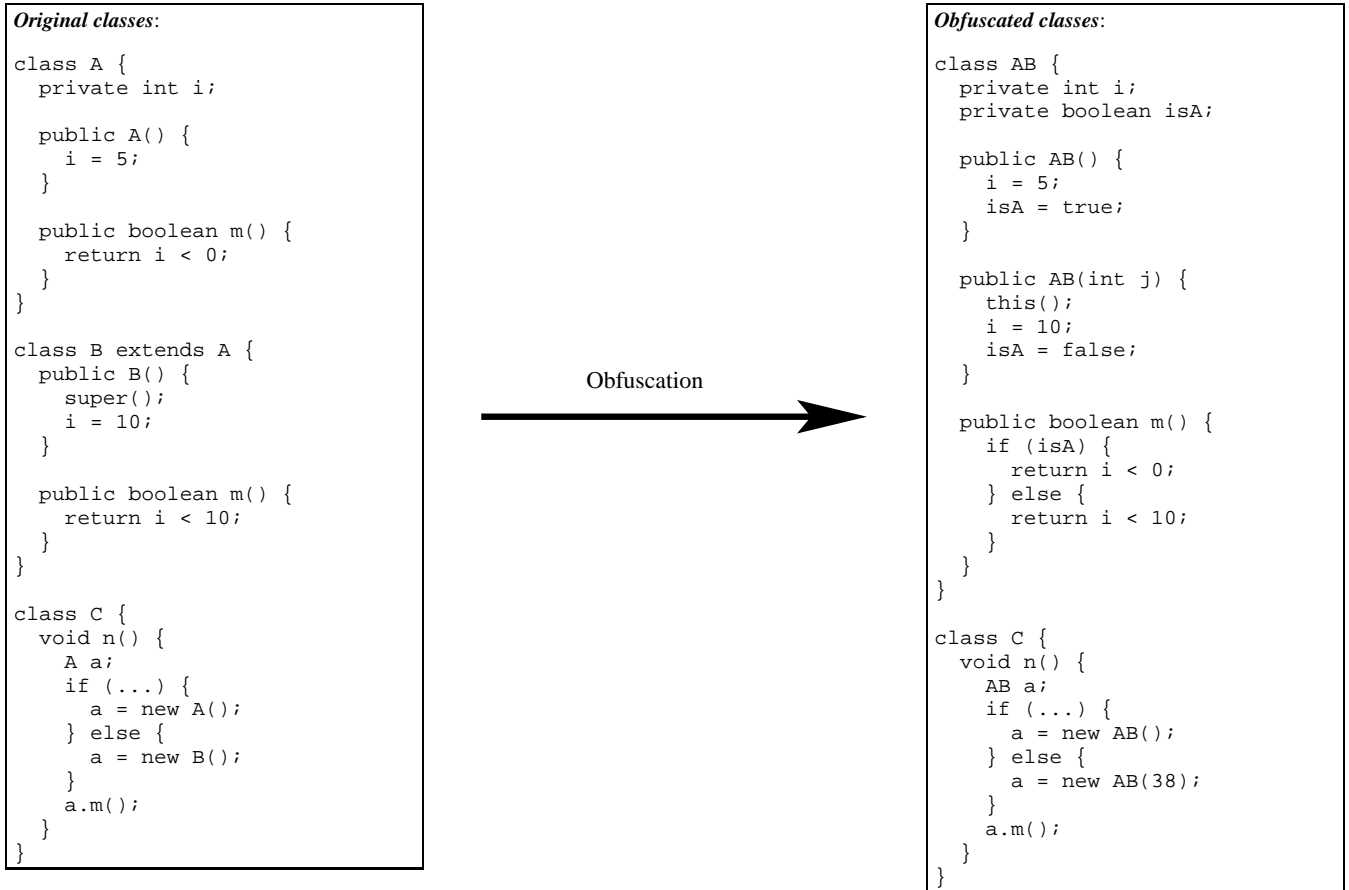a *split function* $\mu_{split} : Members(c_t) \rightarrow 2^{\{c_{t,1}, c_{t,2}\}}$ to designate a

```
Original classes:

class A {
  private int i;

  public A() {
    i = 5;
  }

  public boolean m() {
    return i < 0;
  }
}

class B extends A {
  public B() {
    super();
    i = 10;
  }

  public boolean m() {
    return i < 10;
  }
}

class C {
  void n() {
    A a;
    if (...) {
      a = new A();
    } else {
      a = new B();
    }
    a.m();
  }
}
```

Obfuscation →

```
Obfuscated classes:

class AB {
  private int i;
  private boolean isA;

  public AB() {
    i = 5;
    isA = true;
  }

  public AB(int j) {
    this();
    i = 10;
    isA = false;
  }

  public boolean m() {
    if (isA) {
      return i < 0;
    } else {
      return i < 10;
    }
  }
}

class C {
  void n() {
    AB a;
    if (...) {
      a = new AB();
    } else {
      a = new AB(38);
    }
    a.m();
  }
}
```

Figure 2: Illustration of class coalescing in the presence of inheritance

specific way of splitting a class. In other words, this function can place a member of the original class either in one or both of the target classes. In the general case, splitting a class using an arbitrary split function is not possible because of dependencies among methods and fields of the original class. For example, consider class C in Figure 3. It is not possible to split this class into two classes not related by inheritance where one class contains method m2 and the other contains method m3, because m2 calls m3 in the same class. This example illustrates that dependency relationships of class members have to be taken into account when selecting a split function. We call a split function *valid* if the program resulting from class splitting according to this function is well-formed.

In general, conservative dependency analysis is required when determining validity of a split function. Furthermore, we believe that in practice, splitting a class into two classes not related by inheritance or aggregation is possible only in situations where the original design is flawed and, instead of a single class, there should have been several different classes[15]. Therefore, when splitting class c into $c_{t,1}$ and $c_{t,2}$, we make $c_{t,2}$ a subclass of $c_{t,1}$. Consider the example in Figure 3. In this example, class C is split into classes C1 and C2, where C2 is a subclass of C1. Class D provides an example of how uses of split classes change in the code. In this example, we assume that the split function μ_{split} places fields i and d in C1 and field o in C2. It places both constructors in both C1 and C2, methods m1 and m3 in C1, method m2 in C2, and method m4 in both C1 and C2.

After $c_{t,1}$ and $c_{t,2}$ have been generated, the code using c is modified to use $c_{t,1}$ and $c_{t,2}$ in the following straightforward way. First,

we replace every declaration of a field, local variable, or method argument of type c with a declaration of either type $c_{t,1}$ or $c_{t,2}$. Second, every call to the constructor of c is replaced by the call to the corresponding constructor of $c_{t,2}$. In cases where the field, variable, or argument were declared as $c_{t,1}$ and a call to a method that $c_{t,1}$ does not have is made, we insert a type cast to $c_{t,2}$ prior to this call[4]. No additional manipulations are required since for each method and field of c, class $c_{t,2}$ either contains this method or inherits it from $c_{t,1}$.

Our method of class splitting using inheritance produces a large number of valid split functions, because the only restriction on these split functions is that all methods and fields that a method uses have to be defined in the same class where this method is defined. Formally:

$$\forall m \in Methods(c):$$
$$\quad \text{if } c_{t,1} \in \mu_{split}(m), \text{ then}$$
$$\quad \forall n \in Methods(c): depends(m,n) \Rightarrow c_{t,1} \in \mu_{split}(n) \text{ and}$$
$$\quad \forall f \in Fields(c): depends(m,f) \Rightarrow c_{t,1} \in \mu_{split}(f)$$

Note that method polymorphism can be used to make understanding of code in split classes particularly difficult. First, a method in $c_{t,1}$ may be overridden by $c_{t,2}$ in such a way that the version of this method in $c_{t,1}$ is never actually called. Such method will provide a distraction for a reverse engineer, who is likely to

---

[4]In our current implementation, to avoid run-time overhead associated with type casts, the declared type is always $c_{t,2}$.

```
Original classes:

class C {
  private int i;
  private double d;
  protected Object o;

  public C() {
    i = 5;
    d = 1.0;
    o = new Object();
  }

  public C(int iarg, double darg) {
    i = iarg;
    d = darg;
    o = new Object();
  }

  public boolean m1() {
    return i < 0;
  }

  public void m2() {
    d = 3.0;
    m3(3);
  }

  protected void m3(int iarg) {
    i = iarg;
    m4(new Object());
  }

  public void m4(Object obj) {
    o = obj;
  }
}

class D {
  void n() {
    C c = new C();
    if (c.m1) {...}
    c.m2;
    c.m4;
  }
}
```

```
Obfuscated classes:

class C1 {
  private int i;
  private double d;

  public C1() {
    i = 5;
    d = 1.0;
  }
  public C1(int iarg, double darg) {
    i = iarg;
    d = darg;
  }
  public boolean m1() {
    return i < 0;
  }
  protected void m3(int iarg) {
    i = iarg;
    m4(new Object());
  }
  public void m4(Object obj) {
    o = obj.getClass();
  }
}

class C2 extends C1 {
  protected Object o;

  public C2() {
    super();
    o = new Object();
  }
  public C(int iarg, double darg) {
    super(iarg, darg);
    o = new Object();
  }
  public void m2() {
    d = 3.0;
    m3(3);
  }
  public void m4(Object obj) {
    o = obj;
  }
}

class D {
  void n() {
    C2 c = new C2();
    if (c.m1) {...}
    c.m2;
    c.m4;
  }
}
```

Obfuscation

Figure 3: Illustration of class splitting

assume that they have a purpose. Furthermore, method polymorphism can create the yo-yo effect [16] that makes object-oriented programs difficult to understand. Consider methods m2, m3, and m4 in Figure 3. As defined in class C, m2 calls m3, which in turn calls m4 of the same class. After splitting, method m3 appears only in C1, method m2 appears only in C2, and method m4 appears in both C1 and C2. This arrangement makes it difficult to trace dependencies among these methods: a call to m2 of an object of class C2 generates a call to m3 of C1, which generates a call to m4 of C2 due to polymorphism, even though to a casual code inspector it may appear that m3 of C1 calls m4 of C1.

In our experiments, we were able to produce a variety of valid split functions by placing roughly half of the methods from the original class in $c_{t,1}$ and half in $c_{t,2}$. This is done by processing methods from the original class one by one and choosing one of $c_{t,1}$ and $c_{t,2}$ randomly to place it into. If a method m_1 has been placed in $c_{t,1}$ and it calls another method m_2 that has been placed in $c_{t,2}$, then a method m_2 is also placed in $c_{t,1}$. Note that since this method is overridden by the one in $c_{t,2}$, it will never be called in the program, so arbitrary code can be placed in m_2 in $c_{t,1}$. After all methods have been placed, we place fields. A field f from the original class is placed in $c_{t,1}$ if it is used in a method in $c_{t,1}$. Otherwise, it is placed in $c_{t,2}$. At present, all static methods are placed in $c_{t,2}$.

An alternative to splitting a class c into classes $c_{t,1}$ and $c_{t,2}$, where $c_{t,2}$ is a subclass of $c_{t,1}$ is to add a field of class $c_{t,1}$ to class

$c_{t,2}$. To access the functionality of $c_{t,1}$, an object of class $c_{t,2}$ calls the appropriate methods on its field of class $c_{t,1}$. A drawback of this approach is that class $c_{t,2}$ would have to declare all methods of the original class c, even though some of them only contain delegation calls. We plan to implement and experiment with this splitting technique in the future, but believe that the present, subclassing, solution produces more obfuscated code.

6. TYPE HIDING

In addition to classes, Java also has *interfaces*, a light-weight representation of types. Unlike classes, interfaces do not have instance fields and do not provide implementations for methods that they declare. Interfaces are legitimate types, and as such they can be used in declaration of variables, fields, and methods. Interfaces also can be used in casting operations. All these features make interfaces a convenient design feature that allows decoupling of design from implementation [4]. We use this feature of Java to obscure the nature of objects in the program, by declaring a number n of interfaces $i_1, ..., i_n$ for a given class c and using these interfaces, instead of c, in declarations. Note that class c remains in the program and its members are not changed by this obfuscation.

The key feature of this obfuscation technique is that each interface i_k it creates for a given class c includes only a small random subset of all public methods of c. Class C in Figure 4 is the same as in Figure 3. As our type hiding obfuscation for this example, we introduce two interfaces, I1 and I2 that, combined, declare all public methods of C, m1, m2, and m4. Specifically, I1 declares m1 and m2 and I2 declares m4. Note the use of I1 and I2 in class D: variable c, which prior to this obfuscation had type C, is declared to be of type I1. Since I1 does not declare method m4, a type cast is required when this method is called on c.

Formally, this transformation constructs a mapping μ_{hiding} : $Methods(c) \cap publicMethods(P) \cap instanceMethods(P) \rightarrow \{i_1, ..., i_n\}$ that determines which interface declares which of the private instance methods of c[5]. After $i_1, ..., i_n$ have been generated, we use μ_{hiding} to modify the places in P where fields, local variables, and method arguments of type c are declared and used. First, we modify every declaration using c as type to use an arbitrary interface i from $\{i_1, ..., i_n\}$ as its type. Second, we analyze each instruction that uses the declared field, variable, or argument. If the instruction uses a field of c or calls a static or non-public method of c, we cast the field, variable, or argument to c before this instruction. Alternatively, if the instruction calls a public instance method $m \in Methods(c)$, we use mapping μ_{hiding} to obtain the interface $i = \mu_{hiding}(m)$ that declares the called method and insert a cast to i before the method call instruction.

If used as described here, the type hiding obfuscation can be defeated in an automated way by stripping interfaces, since each interface introduced by this obfuscation is implemented by a single class. For example, tool Jax [17], developed for reducing the size of Java class files, can be adapted for this purpose. To avoid this, method renaming can be used to ensure that a large subset of introduced interfaces is implemented by two or more classes. In addition, polymorphic situations can be created, which would make automated analysis of the obfuscated code even more difficult.

One caveat of this approach is that type casting is expensive in Java. In fact, our early experiments with an implementation of type hiding obfuscation indicated that applying this technique to a large number of classes in a program is likely to slow down this program

---

[5]Note that we could also put a public instance method of c into several different interfaces from $\{i_1, ..., i_n\}$. This feature is planned for a future release of our design obfuscation tool.

significantly. To avoid a large performance impact, our current implementation applies an optimization similar to invariant code motion commonly used in optimizing compilers [1]. Instead of placing type casts inside of loops, we place such type casts before loops.

7. IMPLEMENTATION

We implemented the three design obfuscation techniques described in this paper in a *Design Obfuscator for Java (DOJ)* tool. In this section, we briefly describe the design of DOJ and illustrate its user interface.

7.1 DOJ Design

DOJ works with the bytecodes of the Java application under analysis, relying on the Soot bytecode analysis framework [14]. In a typical obfuscation session, DOJ first uses Soot to analyze the bytecodes in the application that has to be obfuscated. Information about the application is presented to the user, to facilitate the choice of parts of the application to obfuscate and obfuscations to use. The user of DOJ has the choice of selecting classes that have to be obfuscated manually or letting the tool select these classes randomly, according to customizable preference settings.

Once the choice of obfuscations and parts of the application to be obfuscated has been made, DOJ performs obfuscations on the internal representation of Soot. Each obfuscation technique supported by DOJ is implemented as a separate class. Extensibility of DOJ is achieved by using a special class that serves as an adapter between each obfuscation class and the tool interface, implementing the Proxy design pattern[10]. In addition to design obfuscations, DOJ also incorporates several low-level obfuscations, including renaming of classes, methods, and fields.

After the obfuscation modules complete their work, Soot is used to generate bytecodes for the obfuscated application. The user is presented with statistical information about the obfuscations, such as the number of methods in the application that have been modified as the result of obfuscation.

7.2 DOJ User Interface

The main view of the DOJ user interface is split into two panels, as shown in Figure 5. The left panel lists all classes and interfaces of a small online voting application to be obfuscated. The right panel lists classes that are selected for obfuscation. Figure 5 shows classes selected for coalescing: classes ZipCode, Democrat, Republican, and MiddleAge are selected to be coalesced into class named NewClassX and classes JuniorAge, SeniorAge, WashingtonDC, and Main$1 are selected to be coalesced into class named NewClassY. Note that actual obfuscation is not performed until the user clicks the Obfuscate button at the bottom right of the screen. The user has the flexibility of either configuring all obfuscations and then having DOJ to run them all or working in an exploratory mode, performing obfuscations one by one, checking intermediate results after each obfuscation.

performing an obfuscating and viewing the results.

Obfuscating a large application may require information about inner working of this application. DOJ provides a number of tools that aid the user in selecting classes to be obfuscated. Figure 6 illustrates one of these tools, the Hierarchy View. This tool shows dependency relationships among classes in a program. It captures both inheritance relationships and calling dependencies. By clicking on a class in this diagram, the user is shown all classes that depend on this class. This information is helpful when selecting classes to be obfuscated. For example, it is often a good idea to obfuscate a class on which many other classes depend, since this is likely to change the application significantly. In Figure 6, the
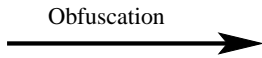
<table>
<tr><td>

Original classes:

```
class C {
  private int i;
  private double d;
  protected Object o;

  public C() {
    i = 5;
    d = 1.0;
    o = new Object();
  }
  public C(int iarg, double darg) {
    i = iarg;
    d = darg;
    o = new Object();
  }
  public boolean m1() {
    return i < 0;
  }
  public void m2() {
    d = 3.0;
    m3(3);
  }
  protected void m3(int iarg) {
    i = iarg;
    m4(new Object());
  }
  public void m4(Object obj) {
    o = obj;
  }
}

class D {
  void n() {
    C c = new C();
    if (c.m1) {...}
    c.m2;
    c.m4;
  }
}
```

</td><td>

Obfuscated classes:

```
class C implements I1, I2 {
  // code in this class is unchanged
}

interface I1 {
  boolean m1();
  public void m2();
}

interface I2 {
  void m4(Object obj);
}

class D {
  void n() {
    I1 c = new C();
    if (c.m1) {...}
    c.m2;
    ((I2) c).m4;
  }
}
```

</td></tr>
</table>

Obfuscation →

Figure 4: Illustration of type hiding
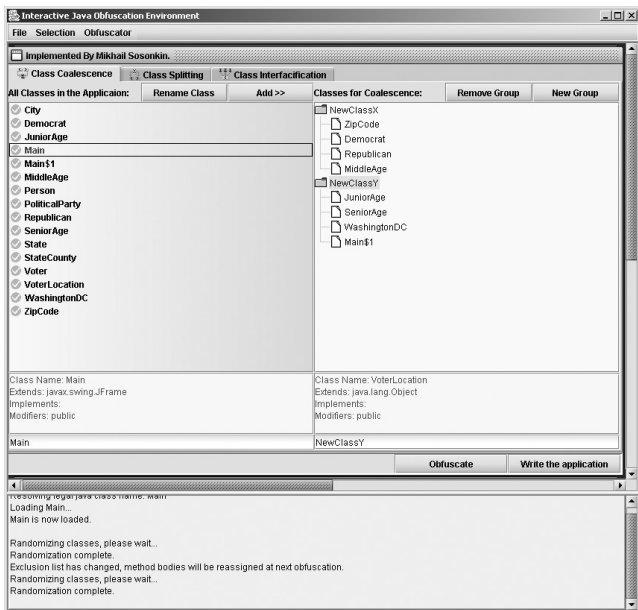
Figure 5: DOJ main view

user clicked on class `PoliticalParty`. DOJ highlights other classes, indicating that classes `Democrat` and `Republican` depend on `PoliticalParty` as its subclasses and classes `Voter`, `MiddleAge`, `JuniorAge`, and `SeniorAge` depend on `PoliticalParty` since they contain calls to its methods.

In addition to Hierarchy View, DOJ provides several other views, including one that provides a number of relevant statistics that attempt to quantify the impact of obfuscations on the program. For example, numbers of classes, methods, and fields that have been changed by the obfuscation are given.

8. EXPERIMENTS

We successfully used DOJ to obfuscate several medium- and large-size programs, including a commercial program with over 2,000 classes[6]. Since in practice run-time performance of software is an important business consideration, it is important that obfuscations do not cripple a software program by making it run too slow. To evaluate the impact of our three obfuscation techniques on program performance, we experimented with five medium-size programs, described in Figure 7.

Each of our obfuscations works with a subset of classes in the

---

[6]Unfortunately, at present we cannot reveal the identity of this program.

Program	Description	Classes	Interfaces	Size (Kb)
javac	Java compiler, included in Sun JDK 1.3.1	298	7	1,083
javap	Java class file disassembler, included in Sun JDK 1.3.1	307	7	1,117
MHP	A tool for computing pairs of statements that may execute in parallel in concurrent programs [13]	94	29	281
HTML Parser	A tool for extracting various features from HTML documents (`http://htmlparser.sourceforge.net`)	81	4	157
DOJ	The obfuscator itself	63	2	249

Figure 7: Programs used in the experiments with DOJ

Figure 6: DOJ Interactive Hierarchy Diagram

Program	Class coalescing, %	Class splitting, %	Type hiding, %
javac	31.34	99.66	99.66
javap	31.22	99.67	99.67
MHP	41.8	100	100
HTML Parser	30.86	100	100
DOJ	27.34	100	100

Figure 8: Applicability of the obfuscations

sign obfuscations to our five test programs. (The actual number of classes that can be obfuscated by DOJ is often less because of imperfections of this tool.) Figure 8 shows the results of the experiment that computes the percentage of classes that can be obfuscated. For class coalescing, this number is the percentage of all pairs of classes that can be coalesced. This number is close to 30% for four of the five test programs. The reason this number is so low is that many classes in these programs subclass library classes. For class splitting and type hiding, the reported numbers are the percentage of classes in each program to which the obfuscation is applicable. Only `javac` and `javap` contain classes (2 in each of them) to which these obfuscations cannot be applied.

In the rest of this section, we report on the experimental results of running DOJ on each test program P. For each test program, we show a graph that plots overhead of running obfuscated versions $P_1, ..., P_k$ compared to running time of P. (Note that version P_i for different obfuscations may be different, depending on the applicability of these obfuscations to classes in P.) All experiments were performed on a RedHat Linux 7.1 (kernel 2.4.2-2) PC workstation with an AMD Athlon 1.3 GHz processor and 1 GB RAM. The obfuscator is implemented in Java and was run on Sun HotSpot JVM 1.3.1, with maximal size of heap set to 1256 MB[8]. We ran each obfuscated version 10 times and averaged running times (we ran each obfuscated version of the MHP test program only 5 times, since each execution takes a substantial amount of time).

8.1 javac

Figure 9 shows the results of the experiment with the Java compiler `javac`. We used the MHP tool (see Figure 7 as input to `javac` in this experiment. Of the 298 classes in this application, we were able to coalesce 54[9], split 290, and type hide 94. The main reason for not being able to split and type hide all classes is immaturity of DOJ. For readability, in Figure 9 we show time comparison only for the first 100 classes in our random order. Performance of those obfuscated versions for class splitting that are not shown differs very little from the performance of the version with 100 split classes (running time of the version with 290 split classes is only

program under analysis. In practice, this subset may be chosen by the user of DOJ. For example, the user may decide to obfuscate only classes that represent crucial intellectual property of the company that developed the program. Since we do not have such information about most of our test programs, we choose subsets of classes to obfuscate randomly (this capability of random selection is a feature of DOJ). For each of our test programs P and each obfuscation technique t, we create a number of obfuscated versions[7] $P_{1,t}, P_{2,t}, ..., P_{k,t}$ in the following manner. We arbitrarily order all classes in P: $c_1, ..., c_{i_t}, ..., c_r$ in such a way that obfuscation t is applicable to subsets $\{c_1\}, \{c_1, c_2\}, ..., \{c_1, c_2, ..., c_{i_t}\}$. For any $i : 1 \leq i \leq i_t$, we obtain $P_{i,t}$ by obfuscating classes in the subset $c_1, ..., c_i$. In some cases, an obfuscation cannot be applied to the whole subset. For example, it is possible that one of the classes in $c_1, ..., c_i$ cannot be coalesced with the other classes in this subset. In such cases, we do not create $P_{i,t}$.

Since one of the goals of this experiment is to compare run-time overheads for different obfuscations, we apply each of our three obfuscations separately. We believe that for maximum protection, in practice, a program should be protected by all three design obfuscation techniques.

First, we report on the theoretical applicability of the three de-

---

[7]We experiment with each obfuscation technique separately (i.e. we do not create obfuscated versions of P by applying both class coalescing and splitting to it).

[8]In all experiments, actual memory usage was under 1GB.

[9]This number could be higher, but since we select classes randomly, the presence of some classes in the set of coalesced classes may prevent adding more classes to this set.
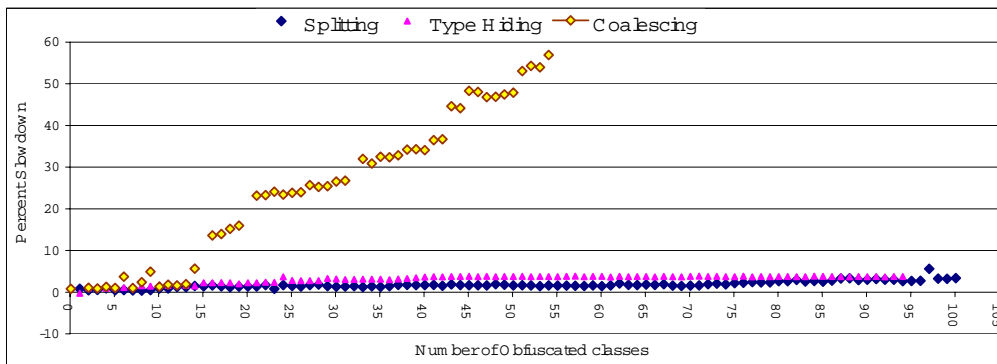
Figure 9: Results of the experiment for javac

0.06 seconds longer than running time of the version with 100 split classes).

From these results, it is obvious that the class splitting and type hiding transformation add very little to the run time of this program. Unfortunately, this is not the case for the class coalescing transformation. After coalescing about 15 classes, the run-time cost of this obfuscation starts rising significantly.

8.2 javap

Figure 10 shows the results for the experiment with the Java decompiler `javap`. We used a component of the Soot framework as input to `javap` in this experiment. DOJ was able to split 305, type hide 189, and coalesce 141 classes. We do not show running times for obfuscated versions with more than 141 classes split and type hidden, since for all these versions, the impact of obfuscations on run-time performance is insignificant. Same as for `javac`, application of class splitting and type hiding obfuscations does not result in large run-time overheads. Also, if a large number of classes are coalesced together, the run-time overheads become large. (The sharp increase in running time after class number 100 was added can be explained by the fact that a large number of objects of this class are created. Since in the obfuscated program each of these objects is significantly larger, more time is spent on their instantiation.) However, where only a small number of classes were coalesced, the obfuscated program was actually faster than the original. We believe that the reason for this is that the Java run-time system loads a small number of large classes more efficiently than a large number of smaller classes. It is a part of our future work to determine if class coalescing can generally be used for optimization purposes.

8.3 MHP

Figure 11 shows the results for the experiment with the MHP (May Happen in Parallel) tool. We used a large concurrent Ada program as input to this tool. While, similar to `javac` and `javap`, application of the class splitting obfuscation does not result in large run-time overheads, application of type hiding to more than 8 classes results in about 25% increase in the running time. Methods of objects of the ninth class are called frequently in the program, often from within nested loops. A type cast before each such invocation is expensive, resulting in the observed loss of performance. Unfortunately, in this case the optimization of the type hiding obfuscation, described in Section 6, is not sufficient to avoid this large overhead. On the other hand, class splitting seems to scale very well; in fact, running time of the version with 2 split classes is exactly the same as running time of the version with 83 split classes — 72.9 sec.

8.4 HTML parser

Figure 12 shows the results for the experiment with the HTML parser. We used an HTML file of size about .5 Mb as input to this tool. For this test program, none of the obfuscations significantly added to the running time of the program. This is not surprising, since this is the smallest program in our tests and so the number of classes used in the obfuscations was small compared to the other test programs.

8.5 DOJ

Figure 13 shows the results for the experiment in which we obfuscated DOJ itself. We used the Concurrent Train Simulation program created by Jim Tsanakaliotis as input to DOJ versions in this experiment. The increase in running time was very insignificant for class coalescing and type hiding. For class splitting, we observed a slight improvement in the running time (0.3 seconds for the version with 59 split classes), but this could be a result of our imprecise time measurement. Overall, good scalability of our obfuscation techniques in this test can be explained by the fact that DOJ does not create a large number of objects. Furthermore, a large portion of running time is spent in calls to the underlying Soot libraries, which were not obfuscated in this experiment.

8.6 Summary

In all of our experiments, the class splitting obfuscation had little or no effect on the program running time. In several cases, application of this obfuscation actually seems to improve program performance somewhat. In our future work, we plan to investigate the possibility of using techniques from our design obfuscations for general optimization of Java programs.

In most cases, the type hiding obfuscation had very little or no effect on the program running time. The only exception was the MHP example. In this example, a number of objects of a class that was type hidden are called intensively from inside nested loops. Since a type cast is inserted before each such call, the cumulative cost of performing the type casts results in a significant run-time penalty. In practice, the user of DOJ would need to decide whether obfuscation of this class with the type hiding obfuscation is necessary for concealing uses of this class in the program. If the class does not represent important intellectual property, a decision can be made not to apply type hiding to it, thereby improving performance of the obfuscated program.

The class coalescing obfuscation appears to be the most expensive of our three design obfuscation techniques. However, in our experiments we used this obfuscation technique in a somewhat extreme way, coalescing a large number of classes into a single class.
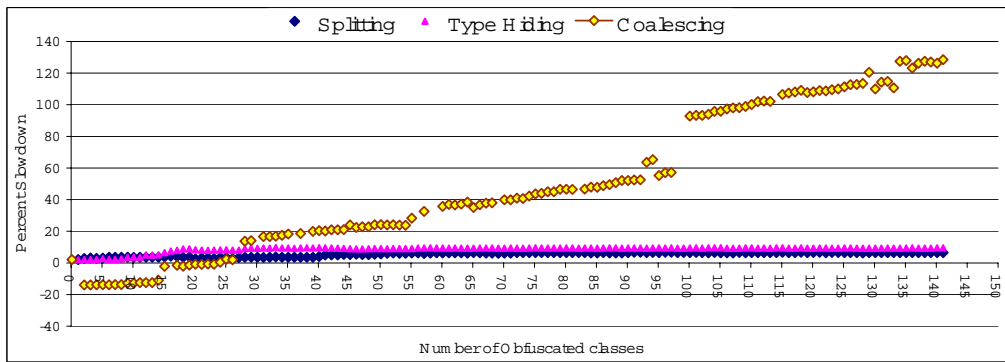
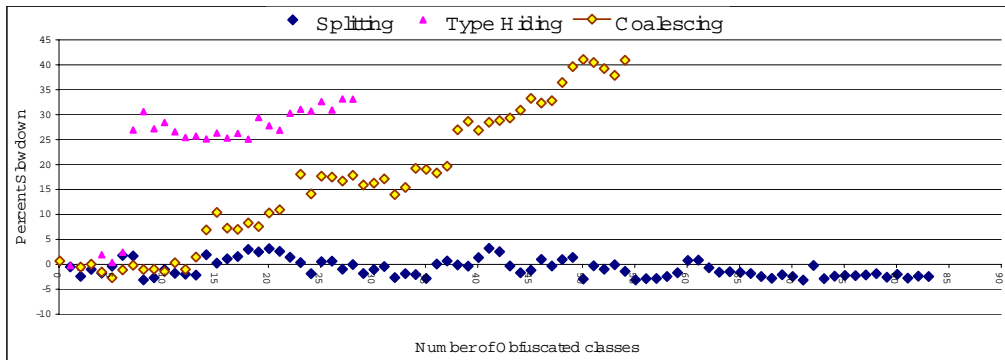**Figure 10: Results of the experiment for javap**

Figure 11: Results of the experiment for MHP

As a result, although the number of objects dynamically created in the program stays constant, many of these objects are much larger, resulting in higher memory consumption and the resulting slowdown. We believe that in practice, it may make more sense to coalesce a number of small groups of classes. In our future work, we plan to investigate a number of patterns of application of this and other obfuscation techniques, evaluating the impact of the choice of a pattern on the running time of the obfuscated program, as well as the strength of obfuscation.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced the notion of design obfuscations for object-oriented programs. These obfuscations alter high-level class structure of a program, concealing the original design. We described three design obfuscations: class coalescing, class splitting, and type hiding. We showed results of an experiment applying these obfuscations to a medium-size Java program. The experiment explored scalability of these obfuscations with respect to run-time overheads caused by them. Based on these preliminary results, it seems that run-time overhead of the class splitting obfuscation tends to be insignificant, under 10% of the total running time of the program, even after most classes in the application have been split. Run-time overhead of type hiding also tends to be insignificant, except in rare cases where an obfuscated class is used very intensively by the program. Finally, run-time overhead of class coalescing seems to be proportional to the number of classes coalesced. Note that in this experiment we coalesced a large number of classes into a single class, instead of coalescing many small groups of classes. We leave experiments with different patterns of coalescing for future work.

More experiments with design obfuscations are needed to evaluate their scalability and run-time overheads. In addition to these experiments, we will evaluate compatibility of design obfuscations with other, low-level types of obfuscations. Finally, it is important to evaluate the degree of protection against reverse engineering provided by our obfuscations. Unfortunately, such protection is a subjective measure and may require experiments with human subjects. Instead, we plan to estimate protection provided by our obfuscations for other protection mechanisms, such as license files and software watermarking [8]. On other area where protection provided by our obfuscation techniques can be objectively measured is that of automated program understanding and refactoring [9] tools. For example, researchers at the University of Passau in Germany are currently implementing a class refactoring technique based on concept analysis[15]. When their tool becomes available, we will be able to determine if our class coalescing obfuscation provides adequate protection against automated improvements of the design by splitting classes.

10. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1988.
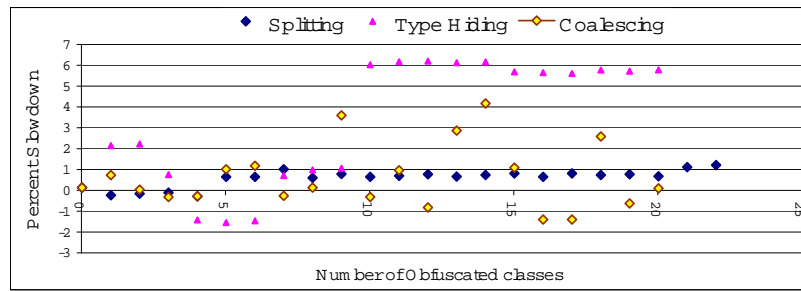
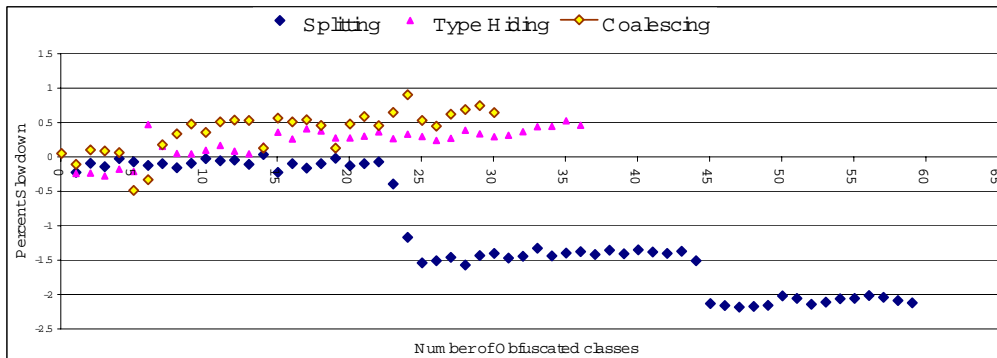**Figure 12: Results of the experiment for HTML Parser**

Figure 13: Results of the experiment for DOJ

[2] R. Anderson and M. Kuhn. Tamper resistance — a cautionary note. In *Proceedings of the second USENIX Workshop on Electronic Commerce*, 1996.

[3] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Proceedings of Advances in Cryptology, 21st Annual International Cryptology Conference (CRYPTO 2001)*, volume 2139, pages 1–18, 2001.

[4] J. Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, 2001.

[5] H. Chang and M. Atallah. Protecting software code by guards. In *Proceedings of the ACM Workshop on Security and Privacy in Digital Rights Management*, pages 160–175, Nov. 2001.

[6] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, July 1997.

[7] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, San Diego, CA, Jan. 1998.

[8] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation — tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, Aug. 2002.

[9] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994. ISBN 0-201-63361-2.

[11] B. Ganter and R. Willie. *Formal Concept Analysis:*

Mathematical Foundations. Springer, 1999.

[12] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.

[13] G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of the 6th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 24–34, Nov. 1998.

[14] P. Pominville, F. Qian, R. Vallee-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing java using attributes. In *Proceedings of CASCON 2000*, pages 152–168, 2000.

[15] G. Snelting and F. Tip. Understanding class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems*, 22(3):540–582, 2000.

[16] D. Taenzer, M. Gandi, and S. Podar. Problems in object-oriented software reuse. In *Proceedings of European Conference on Object-Oriented Programming*, pages 25–38, 1989.

[17] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Practical experience with an application extractor for Java. *ACM SIGPLAN Notices*, 34(10):292–305, 1999.

[18] M. N. Wegman and K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr. 1991.