A Taxonomy of Obfuscating Transformations

Christian Collberg Clark Thomborson Douglas Low

Technical Report #148

Department of Computer Science
The University of Auckland
Private Bag 92019
Auckland, New Zealand.
{*collberg,cthombor,dlow001*}*@cs.auckland.ac.nz*

Abstract

It has become more and more common to distribute software in forms that retain most or all of the information present in the original source code. An important example is Java bytecode. Since such codes are easy to decompile, they increase the risk of malicious reverse engineering attacks.

In this paper we review several techniques for technical protection of software secrets. We will argue that automatic *code obfuscation* is currently the most viable method for preventing reverse engineering. We then describe the design of a *code obfuscator*, a tool which converts a program into an equivalent one that is more difficult to understand and reverse engineer.

The obfuscator is based on the application of code transformations, in many cases similar to those used by compiler optimizers. We describe a large number of such transformations, classify them, and evaluate them with respect to their *potency* (To what degree is a human reader confused?), *resilience* (How well are automatic *deobfuscation* attacks resisted?), and *cost* (How much overhead is added to the application?).

We finally discuss some possible deobfuscation techniques (such as program *slicing*) and possible countermeasures an obfuscator could employ against them.

1 Introduction

Given enough time, effort and determination, a competent programmer will always be able to reverse engineer any application. Having gained physical access to the application, the reverse engineer can decompile it (using disassemblers or decompilers [4]) and then analyze its data structures and control flow. This can either be done manually or with the aid of reverse engineering tools such as program slicers [28].

This is not a new problem. Until recently, however, it is a problem that has received relatively little attention from software developers. The reason is that most programs are large, monolithic, and shipped as stripped, native code, making them difficult (although never impossible) to reverse engineer.

This situation is changing. It is becoming more and more common to distribute software in forms that are easy to decompile and reverse engineer. Important examples include Java bytecode [7] and the *Architecture Neutral Distribution Format* (ANDF) [18]. Java applications in particular pose a problem to software developers. They are distributed over the Internet as Java *class files*, a hardware-independent virtual machine code that retains virtually all the information of the original Java source. Hence, these class files are easy to decompile. Moreover, because much of the computation takes place in standard libraries, Java programs are often small in size and therefore relatively easy to reverse engineer.

The main concern of Java developers is not outright reengineering of entire applications. There is relatively little value in such behavior since it clearly violates copyright law [26], and can be handled through litigation. Rather, developers are mostly frightened by the prospect of a competitor being able to extract proprietary algorithms and data structures from their applications in order to incorporate them into their own programs. Not only does it give the competitor a commercial edge (by cutting development time and cost), but it is also difficult to detect and pursue legally. The last point is particularly valid for small developers who may ill afford lengthy legal battles against powerful corporations [19] with unlimited legal budgets.

The purpose of this paper is to discuss the various forms of technical protection of intellectual property which are available to software developers. We will re-

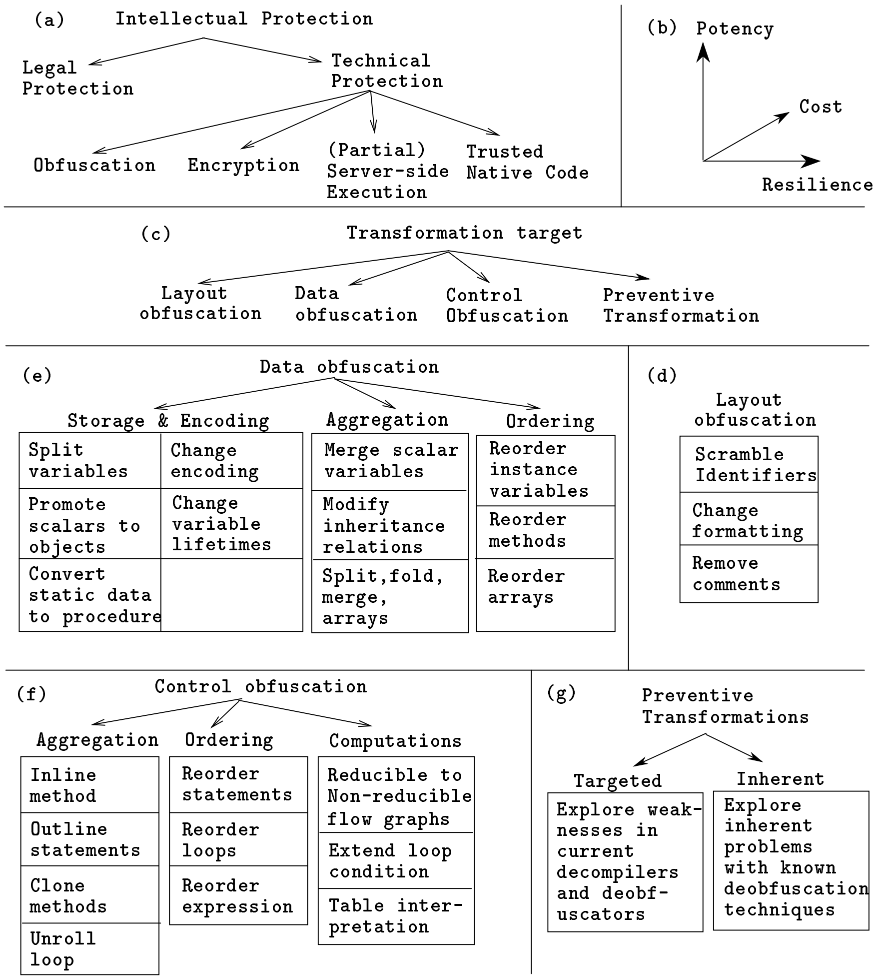


Figure 1: Classification of (a) kinds of protection against malicious reverse engineering, (b) the quality of an obfuscating transformation, (c) information targeted by an obfuscating transformation, (d) layout obfuscations, (e) data obfuscations, (f) control obfuscations, and (g) preventive obfuscations.

strict our discussion to Java programs distributed over the Internet as Java class-files, although most of our results will apply to other languages and architecture-neutral formats as well. We will argue that the only reasonable approach to the protection of mobile code is *code obfuscation*. We will furthermore present a number of *obfuscating transformations*, classify them according to effectiveness and efficiency, and show how they can be put to use in an automatic obfuscation tool.

The remainder of the paper is structured as follows. In Section 2 we give an overview of different forms of technical protection against software theft and argue that code obfuscation currently affords the most economical prevention. In Section 3 we give a brief overview of the design of Kava, a code obfuscator for Java, which is currently under construction. Sections 4 and 5 describe the criteria we use to classify and evaluate different types of obfuscating transformations. The main contributions of the paper are contained in Sections 6, 7, 8, and 9, which present a catalogue of obfuscating transformations. In Section 10 we give more detailed obfuscation algorithms. We conclude with a summary of our results and a discussion of future directions of code obfuscation (Section 11).

2 Protecting Intellectual Property

Consider the following scenario. Alice is a small software developer who wants to make her applications available to users over the Internet, presumably at a charge. Bob is a rival developer who feels that he could gain a commercial edge over Alice if he had access to her application's key algorithms and data structures.

This can be seen as a two-player game between two adversaries: the software developer (Alice) who tries to protect her code from attack, and the reverse engineer (Bob) whose task it is to analyze the application and convert it into a form that is easy to read and understand. Note that it is not necessary for Bob to convert the application back to something close to Alice's original source; all that is necessary is that the reverse engineered code be understandable by Bob and his programmers. Note also that it may not be necessary for Alice to protect her entire application from Bob; it probably consists mostly of "bread-and-butter code" that is of no real interest to a competitor.

Alice can protect her code from Bob's attack using either *legal* or *technical* protection. While copyright law does cover software artifacts, economic realities make it difficult for a small company like Alice's to enforce the law against a larger and more powerful competitor. A more attractive solution is for Alice to protect her code by making reverse engineering so technically difficult that it becomes impossible or at the very least economically inviable. Some early attempts at technical protection are described by Gosler [6].

The most secure approach is for Alice not to sell her application at all, but rather sell its *services*. In other words, users never gain access to the application itself but rather connect to Alice's site to run the program remotely (Figure 2(a)), paying a small amount of electronic money every time. The advantage to Alice is that Bob will never gain physical access to the application and hence will not be able to reverse engineer it. The downside is of course that, due to limits on network bandwidth and latency, the application will perform much worse than if it had run locally on the user's site. A partial solution is to break the application into two parts: a public part that runs locally on the user's site, and a private part (that contains the algorithms that Alice wants to protect) that is run remotely (Figure 2(b)).

Another approach would be for Alice to *encrypt* her code before it is sent off to the users (Figure 3). Unfortunately, this only works if the entire decryption/execution process takes place in hardware. Such systems are described in Herzberg [11] and Wilhelm [31]. If the code is executed in software by a virtual machine interpreter (as is most often the case with Java bytecodes), then it will always be possible for Bob to intercept and decompile the decrypted code.

Java has gained popularity mainly because of its architecture neutral bytecode. While this clearly facilitates mobile code, it does decrease the performance by an order of magnitude in comparison to native code. Predictably, this has lead to the development of *just-in-time compilers* that translate Java bytecodes to native code on-the-fly. Alice could make use of such translators to create native code versions of her application for all popular architectures. When downloading the application, the user's site would have to identify the architecture/operating system combination it is running, and the corresponding version would be transmitted (Figure 4). Only having access to the native code will make Bob's task more difficult, although not impossible.

There is a further complication with transmitting native code. The problem is that — unlike Java bytecodes which are subjected to *bytecode verification* before execution — native codes cannot be run with complete security on the user's machine. If Alice is a trusted member of the community, the user may accept her assurances that the application does not do anything harmful at the user's end. To make sure that no one tries to contaminate the application, Alice would have to digitally sign the codes as they are being transmitted, to prove to the user that the code was the original one written by her.

The final approach we are going to consider is *code obfuscation* (Figure 5). The basic idea is for Alice to run her application through an *obfuscator*, a program that transforms the application into one that is functionally identical to the original but which is much more difficult for Bob to understand. It is our belief that obfuscation is a viable technique for protecting software trade secrets, that has yet to receive the attention that it deserves.

Unlike server-side execution, code obfuscation can never completely protect an application from malicious reverse engineering efforts. Given enough time and determination, Bob will always be able to dissect Alice's application to retrieve its important algorithms and data structures. To aid this effort, Bob may try to run the obfuscated code through an automatic *deobfuscator* that attempts to undo the obfuscating transformations.

Hence, the level of security from reverse engineering that an obfuscator adds to an application depends on (a) the sophistication of the transformations employed by the obfuscator, (b) the power of the available deobfuscation algorithms, and (c) the amount of resources (time and space) available to the deobfuscator. Ideally, we would like to mimic the situation in current public-key cryptosystems, where there is a dramatic difference in the cost of encryption (finding large primes is easy) and decryption (factoring large numbers is difficult). Later on in the paper we will see that there are, in fact, obfuscating transformations that can be applied in polynomial time but which require exponential time to deobfuscate.

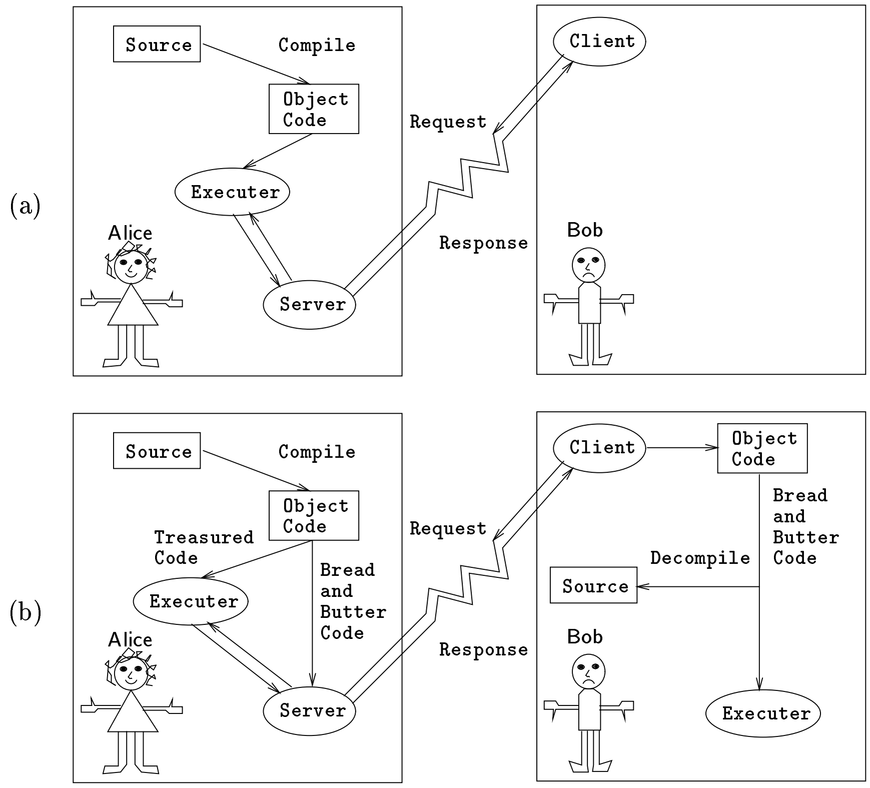


Figure 2: Protection by (a) Server-side and (b) Partial Server-side execution.

3 The Design of a Java Obfuscator

Figure 6 outlines the design of the Java obfuscation tool Kava (K*onfused Java*[1]) which is currently under development. Input to the tool is a Java application, given as a set of Java class files. The user also selects the required level of obfuscation (the *potency*) and the maximum execution time/space penalty that the obfuscator is allowed to add to the application (the *cost*). Kava reads and parses the class files along with any library files referenced directly or indirectly. A complete inheritance tree is constructed, as well as a symbol table giving type information for all symbols, and control flow graphs for all methods.

Kava contains a large pool of code transformations which will be described later in this paper. Before these can be applied, however, a preprocessing pass must collect various types of information about the application. Some kinds of information can be gathered using standard compiler techniques such as inter-procedural data-flow analysis and data dependence analysis, some can be provided by the user, and some are gathered using specialized techniques. *Pragmatic analysis*, for example, analyses the application to see what sort of language constructs and programming idioms it contains.

The information gathered during the preprocessing pass is used to select and apply appropriate code transformations. All types of language constructs in the application can be the subject of obfuscation: classes can be split or merged, methods can be changed or created, new control- and data structures can be created and original ones modified, etc. New constructs added to the application are selected to be as similar as possible to the ones in the source application, based on the pragmatic information gathered during the preprocessing pass.

The transformation process is repeated until the required potency has been achieved or the maximum cost has been exceeded. The output of the tool is a new application – functionally equivalent to the original one – normally given as a set of Java class files. The tool will also be able to produce Java source files annotated with information about which transformations have been applied, and how the obfuscated code relates to the original source. The annotated source will be useful for debugging.

[1]Kava, made from the Kava root (*Piper Methysticum*), is a ceremonial, slightly intoxicating, drink of the south pacific.

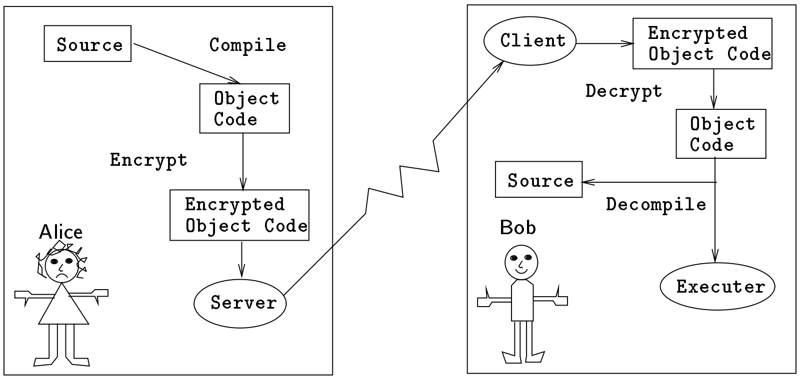


Figure 3: Protection by Encryption.

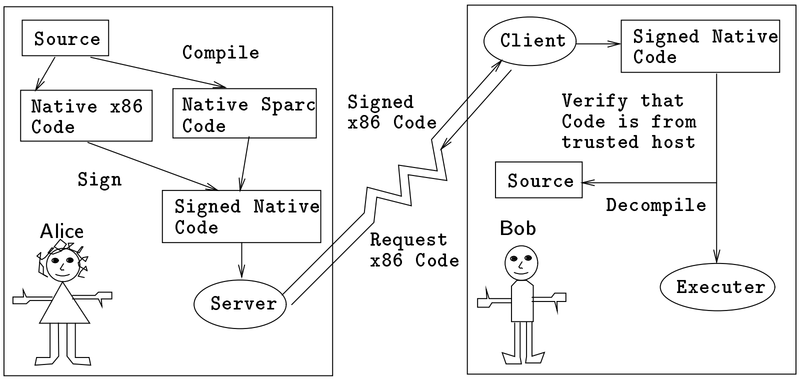


Figure 4: Protection through Signed Native Code.

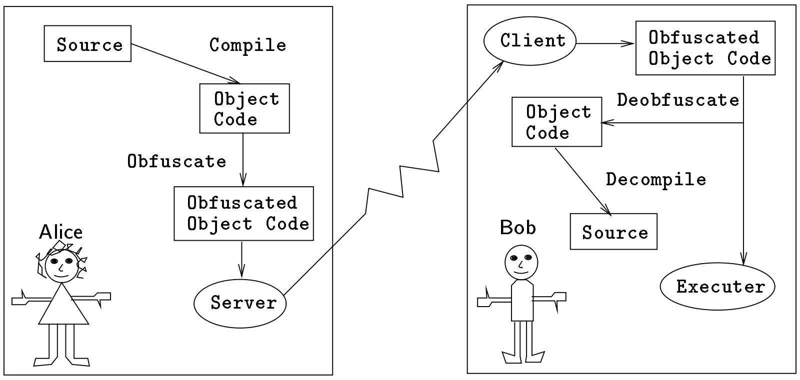


Figure 5: Protection through Obfuscation.

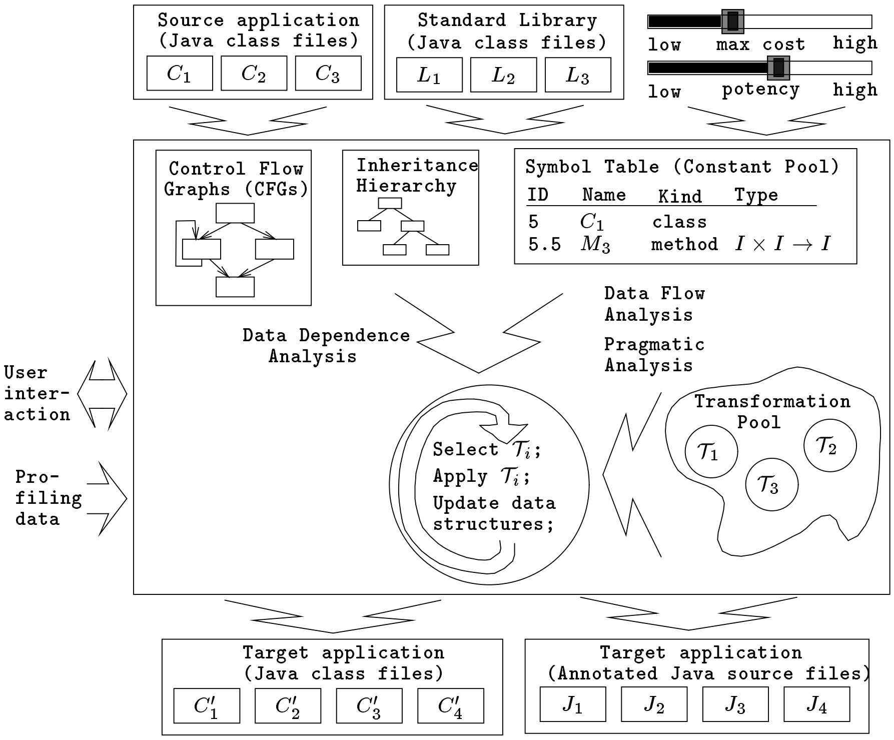


Figure 6: Architecture of Kava, the Java obfuscator. The main input to the tool is a set of Java class files and the obfuscation level required by the user. The user may optionally provide files of *profiling data*, as generated by Java profiling tools. This information can be used to guide the obfuscator to make sure that frequently executed parts of the application are not obfuscated by very expensive transformations.

4 Classifying Obfuscating Transformations

In the remainder of this paper we will describe, classify, and evaluate various obfuscating transformations. We start by formalizing the notion of an *obfuscating transformation*:

Definition 1 (Obfuscating Transformation) Let $P \xrightarrow{\mathcal{T}} P'$ be a transformation of a source program P into a target program P'.

$P \xrightarrow{\mathcal{T}} P'$ is an *obfuscating transformation*, if P and P' have the same *observable behavior*. More precisely, in order for $P \xrightarrow{\mathcal{T}} P'$ to be a legal obfuscating transformation the following conditions must hold:

- If P fails to terminate or terminates with an error condition, then P' may or may not terminate.
- Otherwise, P' must terminate and produce the same output as P.

□

Observable behavior is defined loosely as "behavior as experienced by the user." This means that P' may have side-effects (such as creating files, sending messages over the Internet, etc) that P does not, as long as these side effects are not experienced by the user. Note that we do not require P and P' to equally efficient. In fact, many of our transformations will result in P' being slower or using more memory than P.

The main dividing line between different classes of obfuscation techniques is shown in Figure 1(c). We primarily classify an obfuscating transformation according to the kind of *information* it targets. Some simple transformations target the lexical structure (the *layout*) of the application, such as source code formatting, names of variables, etc. In this paper, the more sophisticated transformations that we are interested target either the data structures used by the application or its flow of control.

Secondly, we classify a transformation according to the kind of operation it performs on the targeted information. As can be seen from Figures 1(d–g), there are several transformations that manipulate the *aggregation* of control or data. Such transformations typically break up abstractions created by the programmer, or construct new bogus abstractions by bundling together unrelated data or control.

Similarly, some transformations affect the *ordering* of data or control. In many cases the order in which two items are declared or two computations are performed has no effect on the observable behavior of the program. There can, however, be much useful information embedded in the chosen order, to the programmer who wrote the program as well as to a reverse engineer. The closer two items or events are in space or time, the higher the likelihood that they are related in one way or another. Ordering transformations try to explore this by randomizing the order of declarations or computations.

5 Evaluating Obfuscating Transformations

Before we can attempt to design any obfuscating transformations, we need to be able to evaluate the quality of such a transformation. In this section we will attempt to classify transformations according to several criteria: how much obscurity they add to the program, how difficult they are to break for a deobfuscator, and how much computational overhead they add to the obfuscated application.

5.1 Measures of Potency

We will first define what it means for a program P' to be more *obscure* (or *complex* or *unreadable*) than a program P. Any such metric will, by definition, be rather vague, since it must be based (in part) on human cognitive abilities.

Fortunately, we can draw upon the vast body of work in the *Software Complexity Metrics* branch of Software Engineering. In this field, metrics are designed with the intent to aid the construction of readable, reliable, and maintainable software. The metrics are frequently based on counting various textual properties of the source code and combining these counts into a measure of complexity. While some of the formulas that have been proposed have been derived from empirical studies of real programs, others have been purely speculative.

The detailed complexity formulas found in the metrics' literature are of little interest to us, but they can be used to derive general statements such as: "if programs P and P' are identical except that P' contains more of property q than P, then P' is more complex than P." Given such a statement, we can attempt to construct a transformation which adds more of the q-property to a program, knowing that this is likely to increase its obscurity.

In Table 1 we paraphrase some of the more popular complexity measures. When used in a software construction project the goal is to *minimize* these measures. In contrast, when obfuscating a program we want to *maximize* the measures.

The complexity metrics allow us to formalize the concept of *potency* which will be used in the remainder of this article as a measure of the usefulness of a transformation. Informally, a transformation is *potent* if it does a good job confusing Bob, by hiding the intent of Alice's original code. In other words, the potency of a transformation measures how much more difficult the obfuscated code is to understand (for a human) than the original code. This is formalized in the following definition:

DEFINITION 2 (TRANSFORMATION POTENCY) Let $\mathcal{T}$ be a behavior-conserving transformation, such that $P \xrightarrow{\mathcal{T}} P'$ transforms a source program P into a target program P'. Let $E(P)$ be the complexity of P, as defined by one of the metrics[2] in Table 1.

$\mathcal{T}_{\text{pot}}(P)$, the *potency* of $\mathcal{T}$ with respect to a program P, is a measure of the extent to which $\mathcal{T}$ changes the complexity of P. It is defined as

$$\mathcal{T}_{\text{pot}}(P) \stackrel{\text{def}}{=} E(P')/E(P) - 1.$$

$\mathcal{T}$ is a *potent obfuscating transformation* if $\mathcal{T}_{\text{pot}}(P) > 0$. □

For the purposes of this paper, we will measure potency on a three-point scale, $\langle$*low*, *medium*, *high*$\rangle$.

The observations in Table 1 make it possible for us to list some desirable properties of a transformation $\mathcal{T}$. In order for $\mathcal{T}$ to be a *potent* obfuscating transformation, it should

- increase overall program size (μ_1) and introduce new classes and methods (μ_7^{a}).
- introduce new predicates (μ_2) and increase the nesting level of conditional and looping constructs (μ_3).
- increase the number of method arguments (μ_5) and inter-class instance variable dependencies (μ_7^{d}).
- increase the height of the inheritance tree ($\mu_7^{\text{b,c}}$).
- increase long-range variable dependencies (μ_4).

5.2 Measures of Resilience

At first glance it would seem that increasing $\mathcal{T}_{\text{pot}}(P)$ would be trivial. To increase the μ_2 metric, for example, all we have to do is to add some arbitrary if-statements to P:

[2] We are deliberately vague as to which particular metric (or combination of metrics) to use since the exact choice is not critical to our application.

Metric	Metric Name	Citation
μ_1	Program Length	Halstead [8]
	$E(P)$ increases with the number of operators and operands in P.	
μ_2	Cyclomatic Complexity	McCabe [20]
	$E(F)$ increases with the number of predicates in F.	
μ_3	Nesting Complexity	Harrison [9]
	$E(F)$ increases with the nesting level of conditionals in F.	
μ_4	Data Flow Complexity	Oviedo [23]
	$E(F)$ increases with the number of inter-basic block variable references in F.	
μ_5	Fan-in/out Complexity	Henry [10]
	$E(F)$ increases with the number of formal parameters to F, and with the number of global data structures read or updated by F.	
μ_6	Data Structure Complexity	Munson [21]
	$E(P)$ increases with the complexity of the static data structures declared in P. The complexity of a scalar variable is constant. The complexity of an array increases with the number of dimensions and with the complexity of the element type. The complexity of a record increases with the number and complexity of its fields.	
μ_7	OO Metric	Chidamber [3]
	$E(C)$ increases with (μ_7^{a}) the number of methods in C, (μ_7^{b}) the depth (distance from the root) of C in the inheritance tree, (μ_7^{c}) the number of direct subclasses of C, (μ_7^{d}) the number of other classes to which C is coupled[a], (μ_7^{e}) the number of methods that can be executed in response to a message sent to an object of C, (μ_7^{f}) the degree to which C's methods do not reference the same set of instance variables. Note: μ_7^{f} measures *cohesion*; i.e. how strongly related the elements of a module are.	

[a]Two classes are coupled if one uses the methods or instance variables of the other.

Table 1: Overview of some popular software complexity measures. $E(X)$ is the complexity of a software component X. F is a function or method, C a class, and P a program.

```
main() {                  main() {
    S1;                       S1;
    S2;        =T=>           if (5==2) S1;
}                             S2;
                              if (1>2) S2;
                          }
```

Unfortunately, such transformations are virtually useless, since they can easily be undone by simple automatic techniques. It is therefore necessary to introduce the concept of *resilience*, which measures how well a transformation holds up under attack from an automatic deobfuscator. The resilience of a transformation $\mathcal{T}$ can be seen as the combination of two measures:

Programmer Effort: the amount of time required to construct an automatic deobfuscator that is able to effectively reduce the potency of $\mathcal{T}$, and

Deobfuscator Effort: the execution time and space required by such an automatic deobfuscator to effectively reduce the potency of $\mathcal{T}$.

It is important to distinguish between resilience and potency. A transformation is *potent* if it manages to confuse a human reader, but it is *resilient* if it confuses an automatic deobfuscator.

We measure resilience on a scale from *trivial* to *one-way*, as shown in Figure 7 (a). One-way transformations are special, in the sense that they can never be undone. This is typically because they *remove* information from the program that was useful to the human programmer, but which is not necessary in order to execute the program correctly. Examples include transformations that remove formatting, scramble variable names, etc. Other transformations typically add useless information to the program that does not change its observable behavior, but which increases the "information load" on a human reader. These transformations can be undone with varying degrees of difficulty.

Figure 7 (b) shows that deobfuscator effort is classified as either *polynomial time* or *exponential time*. Programmer effort, the work required to automate the deobfuscation of a transformation $\mathcal{T}$, is measured as a function of the *scope* of $\mathcal{T}$. This is based on the intuition that it is easier to construct counter-measures against an obfuscating transformation that only affects a small part of a procedure, than against one that may affect an entire program.

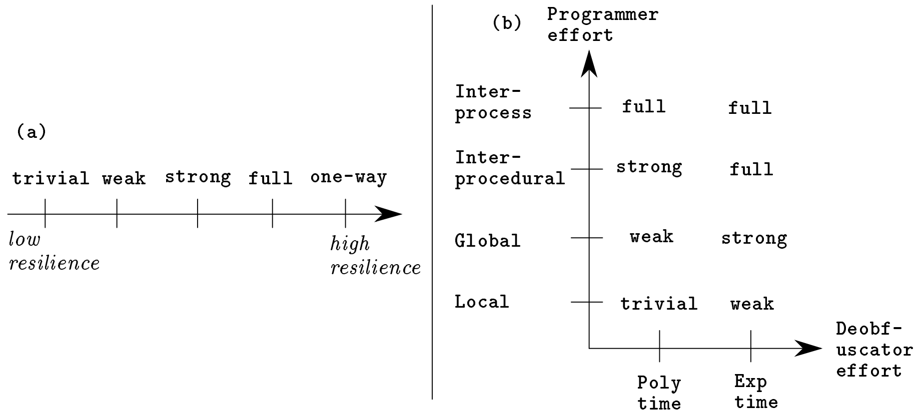


Figure 7: The resilience of an obfuscating transformation.

The scope of a transformation is defined using terminology borrowed from code optimization theory: $\mathcal{T}$ is a *local* transformation if it affects a single basic block of a control flow graph (CFG), it is *global* if it affects an entire CFG, it is *inter-procedural* if it affects the flow of information between procedures, and it is an *inter-process* transformation if it affects the interaction between independently executing threads of control.

Definition 3 (Transformation Resilience) Let $\mathcal{T}$ be a behavior-conserving transformation, such that $P \xrightarrow{\mathcal{T}} P'$ transforms a source program P into a target program P'. $\mathcal{T}_{\text{res}}(P)$ is the *resilience* of $\mathcal{T}$ with respect to a program P.

$\mathcal{T}_{\text{res}}(P)$=*one-way* if information is removed from P such that P cannot be reconstructed from P'. Otherwise,

$$\mathcal{T}_{\text{Res}} \stackrel{\text{def}}{=} \texttt{Resilience}(\mathcal{T}_{\substack{\texttt{Deobfuscator} \\ \texttt{effort}}}, \mathcal{T}_{\substack{\texttt{Programmer} \\ \texttt{effort}}}),$$

where `Resilience` is the function defined in the matrix in Figure 7 (b). □

5.3 Measures of Execution Cost

In Figure 1(b) we see that potency and resilience are two of the three components describing the *quality* of a transformation. The third component, the *cost* of a transformation, is the execution time/space penalty which a transformation incurs on an obfuscated application. We classify the cost on a four-point scale ⟨*free*, *cheap*, *costly*, *dear*⟩, where each point is defined below:

Definition 4 (Transformation Cost) Let $\mathcal{T}$ be a behavior-conserving transformation, such that $P \xrightarrow{\mathcal{T}} P'$ transforms a source program P into a target program P'. $\mathcal{T}_{\text{cost}}(P)$ is the extra execution time/space of P' compared to P.

$$\mathcal{T}_{\text{cost}}(P) \stackrel{\text{def}}{=} \begin{cases} \textit{dear} & \text{if executing } P' \text{ requires } \textit{exponentially} \text{ more resources than } P. \\ \textit{costly} & \text{if executing } P' \text{ requires } \mathcal{O}(n^p),\ p > 1, \text{ more resources than } P. \\ \textit{cheap} & \text{if executing } P' \text{ requires } \mathcal{O}(n) \text{ more resources than } P. \\ \textit{free} & \text{if executing } P' \text{ requires } \mathcal{O}(1) \text{ more resources than } P. \end{cases}$$

□

It should be noted that the actual cost associated with a transformation depends on the environment in which it is applied. For example, a simple assignment statement ⌜`a=5`⌝ inserted at the topmost level of a program will only incur a constant overhead. The same statement inserted inside an inner loop will have a substantially higher cost. Unless noted otherwise, we always give the cost of a transformation as if it had been applied at the outermost nesting level of the source program.

5.4 Measures of Quality

We can now give a formal definition of the *quality* of an obfuscating transformation:

Definition 5 (Transformation Quality) $\mathcal{T}_{\text{qual}}(P)$, the quality of a transformation $\mathcal{T}$, is defined as the combination of the potency, resilience, and cost of $\mathcal{T}$:

$$\mathcal{T}_{\text{qual}}(P) = (\mathcal{T}_{\text{pot}}(P), \mathcal{T}_{\text{res}}(P), \mathcal{T}_{\text{cost}}(P)).$$

□

5.5 Layout Transformations

Before we explore novel transformations, we will briefly consider the trivial layout transformations which are typical of current Java obfuscators such as Crema [29]. The first transformation removes the source code formatting information sometimes available in Java class files. This is a *one-way* transformation because once the original formatting is gone it cannot be recovered; it is a transformation with *low* potency, because there is very little semantic content in formatting, and no great confusion is introduced when that information is removed; finally, this is a *free* transformation since the space and time complexity of the application is not affected.

Scrambling identifier names is also a *one-way* and *free* transformation. However, it has a much higher potency than formatting removal since identifiers contain a great deal of pragmatic information.

6 Control Transformations

In this and the next few sections we will present a catalogue of obfuscating transformations. Some have been derived from well-known transformations used in other areas such as compiler optimization and software reengineering, others have been developed for the sole purpose of obfuscation and are presented here for the first time.

In this section we will discuss transformations that attempt to obscure the control-flow of the source application. As indicated in Figure 1(f), we classify these transformations as affecting the *aggregation*, *ordering*, or *computations* of the flow of control. Control aggregation transformations break up computations that logically belong together or merge computations that do not. Control ordering transformations randomize the order in which computations are carried out. Computation transformations, finally, insert new (redundant or dead) code, or make algorithmic changes to the source application.

For transformations that alter the flow of control, a certain amount of computational overhead will be unavoidable. For Alice this means that she may have to choose between a highly efficient program, and one that is highly obfuscated. Typically, an obfuscator will assist her in this trade-off by allowing her to choose between cheap and expensive transformations.

6.1 Opaque Predicates

The real challenge when designing control-altering transformations is to make them not only cheap, but also resistant to attack from deobfuscators. To achieve this, many transformations rely on the existence of *opaque variables* and *opaque predicates*. Informally, a variable V is opaque if it has some property q which is known *a priori* to the obfuscator, but which is difficult for the deobfuscator to deduce. Similarly, a predicate P (a boolean expression) is opaque if a deobfuscator can deduce its outcome only with great difficulty, while this outcome is well known to the obfuscator.

Being able to create opaque variables and predicates which are difficult for an obfuscator to crack is a major challenge to a creator of obfuscation tools, and the key to highly resilient control transformations. We measure the resilience of an opaque variable or predicate (i.e. its resistance to deobfuscation attacks) on the same scale as transformation resilience, i.e. $\langle$*trivial*, *weak*, *strong*, *full*, *one-way*$\rangle$. Similarly, we measure the added cost of an opaque construct on the same scale as transformation cost, i.e. $\langle$*free*, *cheap*, *costly*, *dear*$\rangle$.

Definition 6 (Opaque Constructs) A variable V is *opaque* at a point p in a program, if V has a property q at p which is known at obfuscation time. We write this as V_p^q or V^q if p is clear from context.

A predicate P is opaque at p if its outcome is known at obfuscation time. We write P_p^F (P_p^T) if P always evaluates to `False` (`True`) at p, and $P_p^?$ if P sometimes evaluates to `True` and sometimes to `False`. See Figure 8. Again, p will be omitted if clear from context. □

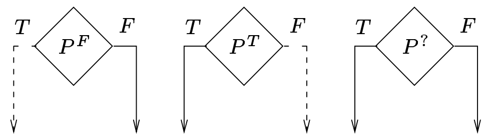


Figure 8: Different types of opaque predicates. Solid lines indicate paths that may sometimes be taken, dashed lines paths that will never be taken.

Below we give some examples of simple opaque constructs. These are easy to construct for the obfuscator, and equally easy to crack for the deobfuscator. Section 8 give examples of opaque constructs with much higher resilience.

6.1.1 Trivial and Weak Opaque Constructs

An opaque construct is *trivial* if a deobfuscator can *crack* it (deduce its value) by a static local analysis. An analysis is local if it is restricted to a single basic block of a control flow graph. Figure 9(a) gives some examples.

We also consider an opaque variable to be trivial if it is computed from calls to library functions with simple, well-understood semantics. For a language like Java which requires all implementations to support a standard set of library classes, such opaque variables are easy to construct. A simple example is $\ulcorner$`int v`$^{\in[1,5]}$`=random(1,5)`$\urcorner$, where `random`(a, b) is a library function that returns an integer in the range $a \cdots b$. Unfortunately, such opaque variables are equally easy to deobfuscate. All that is required is for the deobfuscator-designer to tabulate the semantics of all simple library functions, and then pattern-match on the function calls in the obfuscated code.

An opaque construct is *weak* if a deobfuscator can crack it by a static global analysis. An analysis is global if it is restricted to a single control flow graph. Figure 9(b) gives some examples.

6.2 Computation Transformations

Computation Transformations fall into three categories: hide the real control-flow behind irrelevant statements that do not contribute to the actual computations, introduce code sequences at the object code level for which there exist no corresponding high-level language constructs, or remove real control-flow abstractions or introduce spurious ones.

6.2.1 Insert Dead or Irrelevant Code

The μ_2 and μ_3 metrics suggest that there is a strong correlation between the perceived complexity of a piece of code and the number of predicates it contains. Fortunately, the existence of opaque predicates makes it easy for us to devise transformations that introduce new predicates in a program.

Consider the basic block $S = S_1 \cdots S_n$ in Figure 10. In Figure 10(a) we insert an opaque predicate P^T into S, essentially splitting it in half. The P^T predicate is *irrelevant* code since it will always evaluate to True.

In Figure 10(b) we again break S into two halves, and then proceed to create two *different* obfuscated versions S^a and S^b of the second half. S^a and S^b will be created by applying different sets of obfuscating transformations to the second half of S. Hence, it will not be directly obvious to a reverse engineer that S^a and S^b in fact perform the same function. We use a predicate $P^?$ to select between S^a and S^b at runtime.

Figure 10(c) is similar to Figure 10(b), but this time we introduce a bug into S^b. The P^T predicate always selects the correct version of the code, S^a.

6.2.2 Extend Loop Conditions

Figure 11 shows how we can obfuscate a loop by making the termination condition more complex. The basic idea is to extend the loop condition with a P^T or P^F predicate which will not affect the number of times the loop will execute. The predicate we have added in Figure 11(d), for example, will always evaluate to `True` since $x^2(x+1)^2 \equiv 0 \pmod 4$.

6.2.3 Convert a Reducible to a Non-Reducible Flow Graph

Often, a programming language is compiled to a native or virtual machine code which is more expressive than the language itself. When this is the case, it allows us to device *language-breaking* transformations. A transformation is language-breaking if it introduces virtual machine (or native code) instruction sequences which have no direct correspondence with any source language construct. When faced with such instruction sequences a deobfuscator will either have to try to synthesize an equivalent (but convoluted) source language program, or give up altogether.

For example, the Java *bytecode* has a `goto` instruction while the Java *language* has no corresponding `goto`-statement. This means that the Java bytecode can express *arbitrary* control flow, whereas the Java language can only (easily) express *structured* control flow. Technically [1], we say that the control flow graphs produced from Java programs will always be *reducible*, but the Java bytecode can express *non-reducible* flow graphs.

Since expressing non-reducible flow graphs becomes very awkward in languages without `goto`s, we construct a transformation which converts a reducible flow graph to a non-reducible one. This can be done by turning a structured loop into a loop with multiple headers. In Figure 12(a) we add an opaque predicate P^F to a while loop, to make it appear that there is a jump into the middle of the loop. In fact, this branch will never be taken.

A Java decompiler would have to turn a non-reducible flow graph into one which either duplicates code or which contains extraneous boolean variables. Alternatively, a deobfuscator could guess that all non-reducible flow graphs have been produced by an obfuscator, and simply remove the opaque predicate. To counter this we can sometimes use the alternative transformation shown in Figure 12(b). If a deobfuscator blindly removes P^F, the resulting code will be incorrect.

6.2.4 Remove Library Calls and Programming Idioms

Most programs written in Java rely heavily on calls to the standard libraries. Since the semantics of the library functions are well known, such calls can provide useful clues to a reverse engineer. The problem is exacerbated by the fact that references to Java library classes are always by *name*, and these names cannot be obfuscated.

```
{  int v, a=5; b=6;
   v^{=11} = a + b;
   if (b > 5)^T ...
   if (random(1,5) < 0)^F ...
}
```

(a)

```
{  int v, a=5; b=6;
   if (...) ...
         ⋮  (b is unchanged)
   if (b < 7)^T  a++;
   v^{=36} = (a > 5)?v=b*b:v=b
}
```

(b)

Figure 9: Examples of trivial (a) and weak (b) opaque constructs.

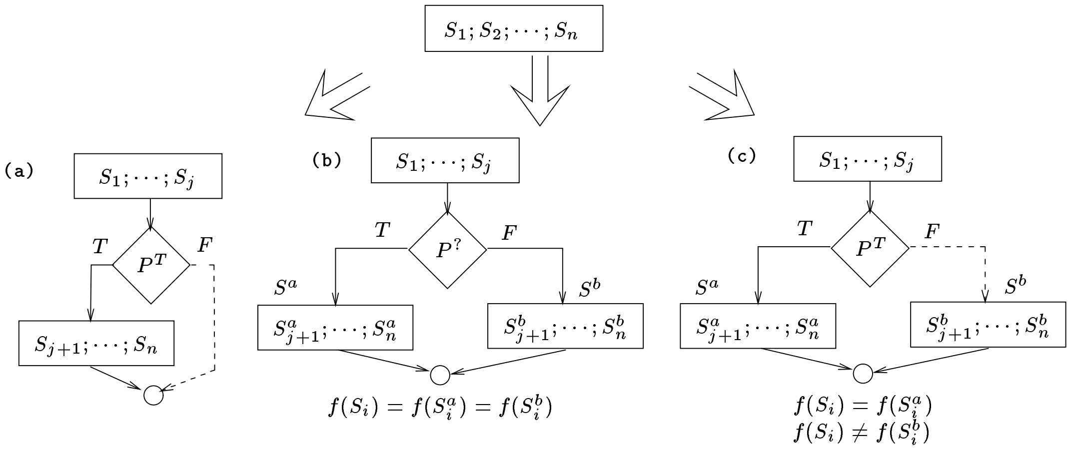
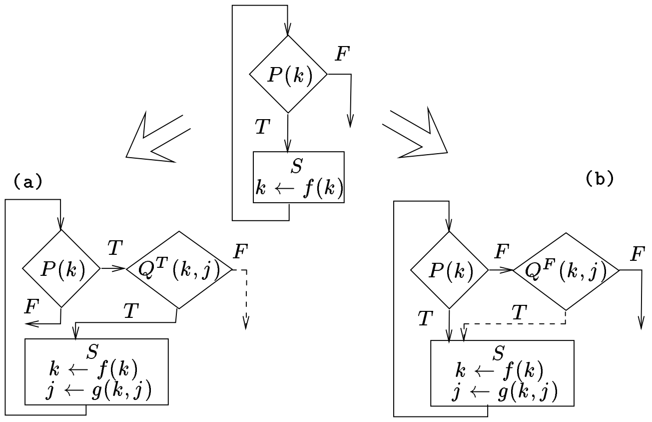


Figure 10: The Branch Insertion transformation.




```
i=1;
while (i<100) {
          ...
   i++;
}
```

(c)

$\overset{\mathcal{T}}{\Longrightarrow}$

```
i=1; j=100;
while ((i<100) && (j*j*(j+1)*(j+1)%4 == 0)^T) {
          ...
   i++;
   j=j*i+3;
}
```

(d)

Figure 11: The Loop Condition Insertion transformation.

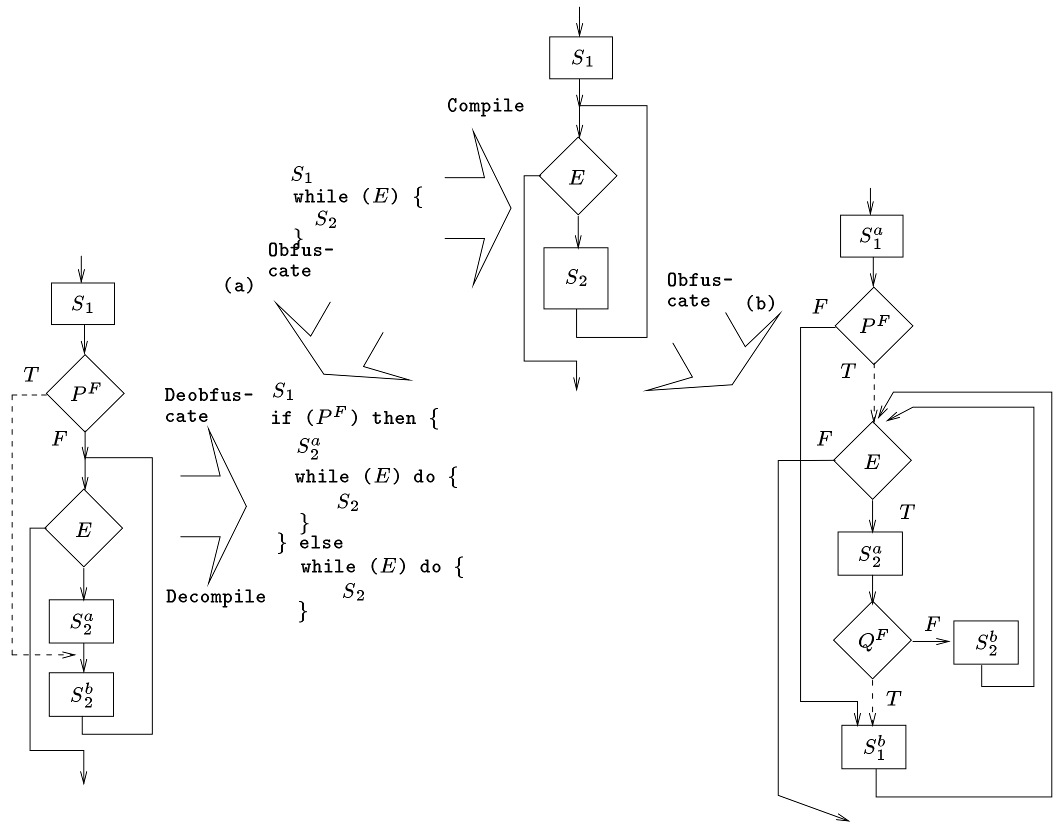


Figure 12: Reducible to Non-Reducible Flow graphs. In (a) we split the loop body S_2 into two parts (S_2^a and S_2^b), and insert a bogus jump to the beginning of S_2^b. In (b) we also break S_1 into two parts, S_1^a and S_1^b. S_1^b is moved into the loop and an opaque predicate P^T ensures that S_1^b is always executed before the loop body. A second predicate Q^F ensures that S_1^b is only executed once.

In many cases the obfuscator will be be able to counter this by simply providing its own versions of the standard libraries. For example, calls to the Java `Dictionary` class (which uses a hash table implementation) could be turned into calls to a class with identical behavior, but implemented as (say) a red-black tree. The cost of this transformation is not so much in execution time, but in the size of the program.

A similar problem occurs with *clichés* (or *patterns*), common programming idioms that occur frequently in many applications. An experienced reverse engineer will search for such patterns to jump-start his understanding of an unfamiliar program. As an example, consider linked lists in Java. The Java library has no standard class that provides common list operations such as `insert`, `delete`, `enumerate`, etc. Instead, most Java programmers will construct lists of objects in an ad hoc fashion by linking them together on a `next` field. Iterating through such lists is a very common pattern in Java programs. Techniques invented in the field *automatic program recognition* [32] can be used to identify common patterns and replace them with less obvious ones. In the linked list case, for example, we might represent the standard list data structure with a less common one, such as cursors into an array of elements.

6.2.5 Table Interpretation

One of the most effective (and expensive) transformations is *table interpretation*.[3] The idea is to convert a section of code (Java bytecode in our case) into a *different* virtual machine code. This new code is then executed by a virtual machine interpreter included with the obfuscated application. Obviously, a particular application can contain several interpreters, each accepting a different language and executing a different section of the obfuscated application.

Since there is usually an order of magnitude slowdown for each level of interpretation, this transformation should be reserved for sections of code that make up a small part of the total runtime or which need a

[3] Thanks to Buz Uzgalis for pointing this out.

very high level of protection.

6.2.6 Add Redundant Operands

Once we have constructed some opaque variables we can use algebraic laws to add redundant operands to arithmetic expressions. This will increase the μ_1 metric. Obviously, this technique works best with integer expressions where numerical accuracy is not an issue. In the obfuscated statement `(1')` below we make use of an opaque variable `P` whose value is 1. In statement `(2')` we construct an *opaque subexpression* `P/Q` whose value is 2. Obviously, we can let `P` and `Q` take on different values during the execution of the program, as long as their quotient is 2 whenever statement `(2')` is reached.

$$\begin{array}{l}\texttt{(1) X=X+V;}\\ \texttt{(2) Z=L+1;}\end{array} \overset{\mathcal{T}}{\Longrightarrow} \begin{array}{l}\texttt{(1') X=X+V*P}^{=1}\texttt{;}\\ \texttt{(2') Z=L+(P}^{=2\texttt{Q}}\texttt{/Q}^{=\texttt{P}/2}\texttt{)/2;}\end{array}$$

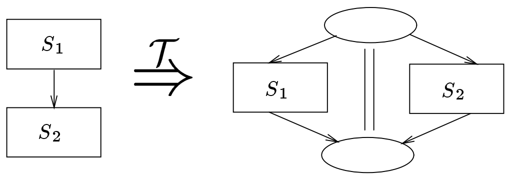
6.2.7 Parallelize Code

Automatic parallelization is an important compiler optimization used to increase the performance of applications running on multi-processor machines. Our reasons for wanting to parallelize a program, of course, are different. We want to increase parallelism not to increase performance, but to obscure the actual flow of control. There are two possible operations available to us:

1. We can create dummy processes that perform no useful task, and
2. we can split a sequential section of the application code into multiple sections executing in parallel.

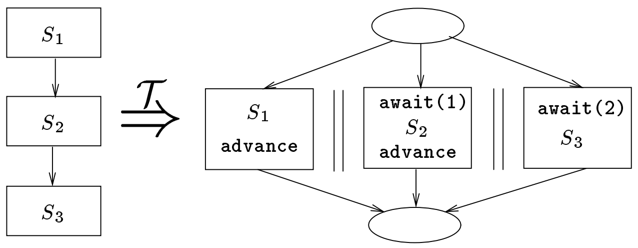
If the application is running on a single-processor machine, we can expect these transformations to have a significant execution time penalty. This may be acceptable in many situations, since the resilience of these transformations is high: static analysis of parallel programs is very difficult since the number of possible execution paths through a program grows exponentially with the number of executing processes. Parallelization also yields high levels of potency: a reverse engineer will find a parallel program much more difficult to understand than a sequential one.

A section of code can be easily parallelized if it contains no data dependencies (Wolfe [33]). For example, if S_1 and S_2 are two data-independent statements they can be run in parallel:



In a programming language like Java that has no explicit parallel constructs, programs will have to be parallelized using calls to thread (lightweight process) libraries.

A section of code that contains data dependencies can be split into concurrent threads by inserting appropriate synchronization primitives such as `await` and `advance` [33]. Such a program will essentially be running sequentially, but the flow of control will be shifting from one thread to the next:



6.3 Aggregation Transformations

Programmers overcome the inherent complexity of programming by introducing abstractions. There is abstraction on many levels of a program, but the *procedural* abstraction is the most important one. For this reason, obscuring procedure and method calls is of the utmost importance to the obfuscator. Below, we will consider several ways in which methods and method invocations can be obscured: inlining, outlining, interleaving, and cloning. The basic idea behind all of these is the same: (1) code which the programmer aggregated into a method (presumably because it logically belonged together) should be broken up and scattered over the program and (2) code which seems *not* to belong together should be aggregated into one method.

6.3.1 Inline and Outline Methods

Inlining is, of course, a important compiler optimization. It is also an extremely useful obfuscation transformation since it removes procedural abstractions from the program. Inlining is a highly resilient transformation (it is essentially *one-way*), since once a procedure call has been replaced with the body of the called procedure and the procedure itself has been removed, there is no trace of the abstraction left in the code.

Outlining (turning a sequence of statements into a subroutine) is a very useful companion transformation to inlining. Figure 13 shows how procedures P and Q are inlined at their call-sites, and then removed from the code. Subsequently, we create a bogus procedural abstraction by extracting the beginning of Q's code and the end of P's code into a new procedure R.

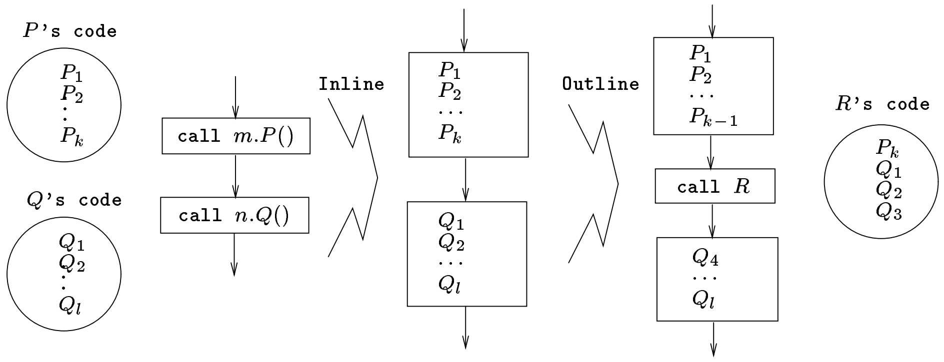


Figure 13: Inlining and outlining transformations.

In object-oriented languages such as Java, inlining may, in fact, not always be a fully one-way transformation. Consider a method invocation $m.P()$. The actual procedure called will depend on the run-time type of m. In cases when more than one method can be invoked at a particular call site, we have to inline all possible methods [5] and select the appropriate code by branching on the type of m (see Figure 14). Hence, even after inlining and removal of methods, the obfuscated code may still contain some traces of the original abstractions.

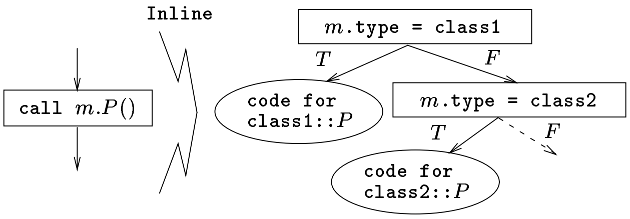


Figure 14: Inlining method calls. Unless we can statically determine the type of `m`, all possible methods to which ⌜`m.P()`⌝ could be bound must be inlined at the call site.

6.3.2 Interleave Methods

The detection of *interleaved code* is an important and difficult reverse engineering task. Rugaber [25] writes:

> *One of the factors that can make a program difficult to understand is that code responsible for accomplishing more than one purpose may be woven together in a single section. We call this* interleaving *[· · ·]*

Figure 15 shows how we can easily interleave two methods declared in the same class. The idea is to merge the bodies and parameter lists of the methods and add an extra parameter (or global variable) to discriminate between calls to the individual methods. Ideally, the methods should be similar in nature to allow merging of common code and parameters. This is the case in Figure 15, where the first parameter of `M1` and `M2` have the same type.

6.3.3 Clone Methods

When trying to understand the purpose of a subroutine a reverse engineer will of course examine its signature and body. However, equally important to understanding the behavior of the routine are the different environments in which it is being called. We can make this process more difficult by obfuscating a method's call sites to make it appear that different routines are being called, when, in fact, this is not the case.

Figure 16 shows how we can create several different versions of a method by applying different sets of obfuscating transformations to the original code. We use method dispatch to select between the different versions at runtime.

Method cloning is similar to the predicate insertion transformations in Figure 10, except that here we are using method dispatch rather than opaque predicates to select between different versions of the code.

6.3.4 Loop Transformations

A large number of loop transformations have been designed with the intent to improve the performance of (in particular) numerical applications. See Bacon [2] for a comprehensive survey. Some of these transformations are useful to us since they also increase the complexity metrics of Table 1. *Loop Blocking* (Figure 17(a)) is used to improve the cache behavior of a loop by breaking up the iteration space so that the inner loop fits in the cache. *Loop unrolling* (Figure 17(b)) replicates the body of a loop one or more times. If the loop bounds are known at compile time the loop can be unrolled in its entirety. *Loop fission* (Figure 17(c)) turns a loop

```
class C {
   method M1 (T1 a) {
      S_1^{M1}; ··· S_k^{M1};
   }
   method M2 (T1 b; T2 c) {
      S_1^{M2}; ··· S_m^{M2};
   }
}

{  C x=new C;
   x.M1(a); x.M2(b, c); }
```

$\stackrel{\mathcal{T}}{\Longrightarrow}$

```
class C' {
   method M (T1 a; T2 c; int V) {
      if (V == p) {S_1^{M1}; ··· S_k^{M1};}
      else        {S_1^{M2}; ··· S_m^{M2};}
   }
}

{  C' x=new C';
   x.M(a, c, V^{=p});
   x.M(b, c, V^{=q}); }
```

Figure 15: Interleaving methods. An opaque variable V is passed to the interleaved method to discriminate between calls to `M1` and `M2`. There is, of course, nothing stopping us from merging the bodies in less obvious ways, possibly using several opaque predicates $P_i(V)$: ⌜`if` $(P_1(V))$ $S_1^{\mathtt{M1}}$ `else` $S_1^{\mathtt{M2}}$`;` `if` $(P_2(V))$ $S_2^{\mathtt{M1}}$ `else` $S_2^{\mathtt{M2}}$`;` $\cdots$⌝

```
class C {
   method m (int x)
      { S_1 ··· S_k }
}

{  C x = new C;
   x.m(5); ··· x.m(7);
}
```

$\stackrel{\mathcal{T}}{\Longrightarrow}$

```
class C1 {
   method m (int x)
      { S_1^a ··· S_k^a }
   method m1 (int x)
      { S_1^c ··· S_k^c }
}

class C2 inherits C1 {
   method m (int x)
      { S_1^b ··· S_k^b }
}

{  C1 x ;
   if (P^F) x=new C1 else x=new C2;
   x.m(5); ···; x.m1(7);
}
```

Figure 16: Cloning methods. `C2::m` and `C1::m1` have been generated by applying different obfuscating transformations to the body of `C::m`. The calls ⌜`x.m(5)`⌝ and ⌜`x.m1(7)`⌝ look as if they were made to two different methods, while in fact they go to different-looking methods with identical behavior. `C1::m` is a buggy version of `C::m` that is never called.

with a compound body into several loops with the same iteration space.

All three transformations increase the μ_1 and μ_2 metrics, since they increase the source application's total code size and number of conditions. The loop blocking transformation also introduces extra nesting, and hence also increases the μ_3 metric.

Applied in isolation, the resilience of these transformations is quite low. It does not require much static analysis for a deobfuscator to *reroll* an unrolled loop. However, when the transformations are combined, the resilience rises dramatically. For example, given the simple loop in Figure 17(b), we could first apply unrolling, then fission, and finally blocking. Returning the resulting loop to its original form would require a fair amount of analysis for the deobfuscator.

6.4 Ordering Transformations

Programmers tend to organize their source code to maximize its *locality*. The idea is that a program is easier to read and understand if two items that are logically related are also physically close in the source text. This kind of locality works on every level of the source: there is locality among terms within expressions, statements within basic blocks, basic blocks within methods, methods within classes, classes within files, etc. All kinds of spatial locality can provide useful clues to a reverse engineer. Therefore, whenever possible, we randomize the placement of any item in the source application. For some types of items (methods within classes, for example) this is trivial. In other cases (such as statements within basic blocks) a data dependency analysis (see

(a)

```
for(i=1,i<=n,i++)
  for(j=1,j<=n,j++)
    a[i,j]=b[j,i]
```

$\stackrel{\mathcal{T}}{\Longrightarrow}$

```
for(I=1,I<=n,I+=64)
  for(J=1,J<=n,J+=64)
    for(i=I,i<=min(I+63,n),i++)
      for(j=J,j<=min(J+63,n),j++)
        a[i,j]=b[j,i]
```

(b)

```
for(i=2,i<(n-1),i++)
  a[i] += a[i-1]*a[i+1]
```

$\stackrel{\mathcal{T}}{\Longrightarrow}$

```
for(i=2,i<(n-2),i+=2) {
  a[i] += a[i-1]*a[i+1];
  a[i+1] += a[i]*a[i+2];
};
if (((n-2) % 2) == 1)
  a[n-1] += a[n-2]*a[n]
```

(c)

```
for(i=1,i<n,i++) {
  a[i] += c;
  x[i+i]=d+x[i+1]*a[i]
}
```

$\stackrel{\mathcal{T}}{\Longrightarrow}$

```
for(i=1,i<n,i++)
  a[i] += c;
for(i=1,i<n,i++)
  x[i+i]=d+x[i+1]*a[i]
```

Figure 17: Loop transformation examples. Loop blocking (a), loop unrolling (b), and loop fission (c). These examples were adapted from [2].

[2, 33]) will have to be performed to determine which reorderings are legal.

These transformations have low potency (they do not add much obscurity to the program) but their resilience is high, in many cases *one-way*. For example, when the placement of statements within a basic block has been randomized, there will be no traces of the original order left in the resulting code.

Ordering transformations can be particularly useful companions to the "Inline-Outline" transformation of Section 6.3.1. The potency of that transformation can be enhanced by (1) inlining several procedure calls in a procedure P, (2) randomizing the order of the statements in P, and (3) outlining contiguous sections of P's statements. This way, unrelated statements that were previously part of several different procedures are brought together into bogus procedural abstractions.

In certain cases it is also possible to reorder loops, for example by running them backwards. Such *loop reversal* transformations are common in high-performance compilers [2].

7 Data Transformations

In this section we will discuss transformations that obscure the data structures used in the source application. As indicated in Figure 1(e), we classify these transformations as affecting the *storage*, *encoding*, *aggregation*, or *ordering* of the data.

7.1 Storage and Encoding Transformations

In many cases there is a "natural" way to store a particular data item in a program. For example, to iterate through the elements of an array we probably would choose to allocate a local integer variable of the appropriate size as the iteration variable. Other variable types might be possible, but they would be less natural and probably less efficient.

Furthermore, there is also often a "natural" interpretation of the bit-patterns that a particular variable can hold which is based on the type of the variable. For example, we would normally assume that a 16-bit integer variable storing the bit-pattern ⌜0000000000001100⌝ would represent the integer value 12. Of course, these are mere conventions and other interpretations are possible.

Obfuscating *storage* transformations attempt to choose unnatural storage classes for dynamic as well as static data. Similarly, *encoding* transformations attempt to choose unnatural encodings for common data types. Storage and encoding transformations often go hand-in-hand, but they can sometimes be used in isolation.

7.1.1 Change Encoding

As a simple example of an encoding transformation we will replace an integer variable `i` by $\texttt{i}' = c_1 \cdot \texttt{i} + c_2$, where c_1 and c_2 are constants. For efficiency, we could choose c_1 to be a power of two. In the example below, we let $c_1 = 8$ and $c_2 = 3$:

```
int i=1;
while (i < 1000) {
  ··· A[i] ···;
  i++;
}
```

$\stackrel{\mathcal{T}}{\Longrightarrow}$

```
int i=11;
while (i<8003) {
  ··· A[(i-3)/8] ···;
  i+=8;
}
```

Obviously, overflow (and, in case of floating point variables, accuracy) issues need to be addressed. We could either determine that because of the range of the variable[4] in question no overflow will occur, or we could change to a larger variable type.

There will be a trade-off between resilience and potency on one hand, and cost on the other. A simple encoding function such as $\texttt{i}' = c_1 \cdot \texttt{i} + c_2$ in the example above, will add little extra execution time but can be deobfuscated using common compiler analysis techniques [33, 2].

7.1.2 Promote Variables

There are a number of simple storage transformations that *promote* variables from a specialized storage class to a more general class. Their potency and resilience are generally low, but used in conjunction with other transformations they can be quite effective.

For example, in Java, an integer variable can be promoted to an integer *object*. The same is true of the other scalar types which all have corresponding "packaged" classes. Since Java supports garbage collection, the objects will be automatically removed when they are no longer referenced. Here is an example:

```
int i=1;
while (i < 9) {
   ··· A[i] ···;
   i++;
}
```

$\xrightarrow{\mathcal{T}}$

```
Int i = new Int(1);
while (i.value < 9) {
   ··· A[i.value] ···;
   i.value++;
}
```

It is also possible to change the *lifetime* of a variable. The simplest such transform turns a local variable into a global one which is then shared between independent procedure invocations. For example, if procedures P and Q both reference a local integer variable, and P and Q cannot both be active at the same time[5] then the variable can be made global and shared between them:

```
void P() {
   int i; ··· i ···
}
void Q() {
   int k; ··· k ···
}
```

$\xrightarrow{\mathcal{T}}$

```
int C;
void P() {
   ··· C ···
}
void Q() {
   ··· C ···
}
```

This transformation increases the μ_5 metric since the number of global data structures referenced by P and Q is increased.

7.1.3 Split Variables

Boolean variables and other variables of restricted range can be split into two or more variables. We will write a variable V split into k variables $p_1, \cdots, p_k$ as $V = [p_1, \cdots, p_k]$. Typically, the potency of this transformation will grow with k. Unfortunately, so will the cost of the transformation, so we usually restrict k to 2 or 3.

To allow a variable V of type T to be split into two variables p and q of type U requires us to provide three pieces of information: (1) a function $f(p, q)$ that maps the values of p and q into the corresponding value of V, (2) a function $g(V)$ that maps the value of V into the corresponding values of p and q, and (3) new operations (corresponding to the primitive operations on values of type T) cast in terms of operations on p and q. In the remainder of this section we will assume that V is of type boolean, and p and q are small integer variables.

Figure 18(a) shows a possible choice of representation for split boolean variables. The table indicates that if V has been split into p and q, and if, at some point in the program, $p = q = 0$ or $p = q = 1$, then that corresponds to V being `False`. Similarly, $p = 0, q = 1$ or $p = 1, q = 0$ corresponds to `True`.

Given this new representation, we have to device substitutions for the built-in boolean operations (`&`, `|`, `~`, `^`). The easiest way is simply to provide a run-time lookup table for each operator. Tables for `&` and `|` are shown in Figure 18(c) and (d), respectively. Given two boolean variables $V_1 = [p, q]$ and $V_2 = [r, s]$, $V_1 \& V_2$ is computed as $\texttt{AND}[2p + q, 2r + s]$.

In Figure 18(e) we show the result of splitting three boolean variables `A=[a1,a2]`, `B=[b1,b2]`, and `C=[c1,c2]`. An interesting aspect of our chosen representation is that there are several possible ways to compute the same boolean expression. Statements (3') and (4') in Figure 18(e), for example, look different, although they both assign `False` to a variable. Similarly, while statements (5') and (6') are completely different, they both compute `A & B`.

The potency, resilience, and cost of this transformation all grow with the number of variables into which the original variable is split. The resilience can be further enhanced by selecting the encoding *at run-time*. In other words, the run-time look-up tables of Figure 18(b-d) are not constructed at compile-time (which would make them susceptible to static analyses) but by algorithms included in the obfuscated application. This, of course, would prevent us from using in-line code to compute primitive operations, as done in statement (6') in Figure 18(e).

7.1.4 Convert Static to Procedural Data

Static data, particularly character strings, contain much useful pragmatic information to a reverse engineer. A simple way of obfuscating a static string is to convert it into *a program* that produces the string. The program

[4] The range can be determined using static analysis techniques or by querying the user.

[5] Unless the program contains threads this can be determined by examining the static call graph.

$g(V)$		$f(p,q)$	
p	q	V	$2p+q$
0	0	False	0
0	1	True	1
1	0	True	2
1	1	False	3

(a)

VAL[p,q]		p = 0	p = 1
q	0	0	1
	1	1	0

(b)

AND[A,B]		A = 0	A = 1	A = 2	A = 3
B	0	3	0	0	0
	1	3	1	2	3
	2	0	2	1	3
	3	3	0	0	3

(c)

OR[A,B]		A = 0	A = 1	A = 2	A = 3
B	0	3	1	2	3
	1	1	1	2	2
	2	2	2	1	1
	3	0	1	2	0

(d)

(e)

```
(1)  bool A,B,C;                 (1')  short a1,a2,b1,b2,c1,c2;
(2)  A = True;                   (2')  a1=0; a2=1;
(3)  B = False;                  (3')  b1=0; b2=0;
(4)  C = False;          T       (4')  c1=1; c2=1;
(5)  C = A & B;         ==>      (5')  x=AND[2*a1+a2,2*b1+b2]; c1=x/2; c2=x%2;
(6)  C = A & B;                  (6')  c1=(a1 ^ a2) & (b1 ^ b2); c2=0;
(7)  C = A | B;                  (7')  x=OR[2*a1+a2,2*b1+b2]; c1=x/2; c2=x%2;
(8)  if (A) ···;                 (8')  x=2*a1+a2; if ((x==1) || (x==2)) ···;
(9)  if (B) ···;                 (9')  if (b1 ^ b2) ···;
(10) if (C) ···;                 (10') if (VAL[c1,c2]) ···;
```

Figure 18: Variable splitting example. Tables (b-d) are used to compute boolean operations. They are either constructed by the obfuscator and stored in the as static data in the obfuscated application, or generated at run-time by the obfuscated application itself.

– which could be a DFA, a Trie traversal, etc. – could possibly produce other strings as well.

As an example, consider the function `G` in Figure 19. This function was constructed to obfuscate the strings `"AAA"`, `"BAAAA"`, and `"CCB"`. The values produced by `G` are `G(1)="AAA"`, `G(2)="BAAAA"`, `G(3)=G(5)="CCB"`, and `G(4)="XCB"` (which is not actually used in the program). For other argument values, `G` may or may not terminate.

Aggregating the computation of all static string data into just one function is, of course, highly undesirable. Much higher potency and resilience is achieved if the `G`-function was broken up into smaller components that were embedded into the "normal" control flow of the source program.

It is interesting to note that we can combine this technique with the *table interpretation* transformation of Section 6.2.5. The intent of that obfuscation is to convert a section of Java bytecode into code for another virtual machine. The new code will typically be stored as static string data in the obfuscated program. For even higher levels of potency and resilience, however, the strings could be converted to programs that produce them, as explained above.

7.2 Aggregation Transformations

In contrast to imperative and functional languages, object-oriented languages are more data-oriented than control-oriented. In other words, in an object-oriented program, the control is organized around the data structures, rather than the other way around. This means that an important part of reverse-engineering an object-oriented application is trying to restore the program's data structures. Conversely, it is important for an obfuscator to try to hide these data structures.

In most object-oriented languages, there are just two ways to aggregate data: in arrays and in objects. In the next three sections we will examine ways in which these data structures can be obfuscated.

7.2.1 Merge Scalar Variables

Two or more scalar variables $V_1 \cdots V_k$ can be merged into one variable V_M, provided the combined ranges of $V_1 \cdots V_k$ will fit within the precision of V_M. For example, two 32-bit integer variables could be merged into one 64-bit variable. Arithmetic on the individual variables would be transformed into arithmetic on V_M. As a simple example, consider merging two 32-bit integer variables `X` and `Y` into a 64-bit variable `Z`. Using the merging formula

$$\mathtt{Z(X,Y)} = 2^{32} \cdot \mathtt{Y} + \mathtt{X}$$

we get the arithmetic identities in Figure 20(a). Some simple examples are given in Figure 20(b).

The resilience of variable merging is quite low. A deobfuscator only needs to examine the set of arithmetic operations being applied to a particular variable in order to guess that it actually consists of two merged

```
String G (int n) {
   int i=0,k;
   String S;
   while (1) {
      L1:  if (n==1) {S[i++]="A";k=0;goto L6};
      L2:  if (n==2) {S[i++]="B";k=-2;goto L6};
      L3:  if (n==3) {S[i++]="C";goto L9};
      L4:  if (n==4) {S[i++]="X";goto L9};
      L5:  if (n==5) {S[i++]="C";goto L11};
            if (n>12) goto L1;
      L6:  if (k++<=2) {S[i++]="A";goto L6} else goto L8;
      L8:  return S;
      L9:  S[i++]="C"; goto L10;
      L10:  S[i++]="B"; goto L8;
      L11:  S[i++]="C"; goto L12;
      L12:  goto L10;
   }
}
```

Figure 19: A function producing the the strings "AAA", "BAAAA", and "CCB".

(a)

$$\begin{aligned}
\mathtt{Z}(\mathtt{X}+r,\mathtt{Y}) &= 2^{32}\cdot\mathtt{Y}+(r+\mathtt{X}) &&= \mathtt{Z}(\mathtt{X},\mathtt{Y})+r\\
\mathtt{Z}(\mathtt{X},\mathtt{Y}+r) &= 2^{32}\cdot(\mathtt{Y}+r)+\mathtt{X} &&= \mathtt{Z}(\mathtt{X},\mathtt{Y})+r\cdot 2^{32}\\
\mathtt{Z}(\mathtt{X}\cdot r,\mathtt{Y}) &= 2^{32}\cdot\mathtt{Y}+\mathtt{X}\cdot r &&= \mathtt{Z}(\mathtt{X},\mathtt{Y})+(r-1)\cdot\mathtt{X}\\
\mathtt{Z}(\mathtt{X},\mathtt{Y}\cdot r) &= 2^{32}\cdot\mathtt{Y}\cdot r+\mathtt{X} &&= \mathtt{Z}(\mathtt{X},\mathtt{Y})+(r-1)\cdot 2^{32}\cdot\mathtt{Y}
\end{aligned}$$

(b)

```
(1) int X=45,Y=95;
(2) X += 5;
(3) Y += 11;
(4) X *= c;
(5) Y *= d;
```

$\stackrel{\mathcal{T}}{\Longrightarrow}$

```
(1') long Z=167759086119551045;
(2') Z += 5;
(3') Z += 47244640256;
(4') Z += (c-1)*(Z & 4294967295);
(5') Z += (d-1)*(Z & 18446744069414584320);
```

Figure 20: Merging two 32-bit variables X and Y into one 64-bit variable Z. Y occupies the top 32 bits of Z, X the bottom 32 bits. If the actual range of either X or Y can be deduced from the program, less intuitive merges could be used. (a) gives rules for addition and multiplication with X and Y. (b) shows some simple examples. The example could be further obfuscated, for example by merging (2') and (3') into ⌜Z+=47244640261⌝.

variables. We can increase the resilience by introducing bogus operations that could not correspond to any reasonable operations on the individual variables. In the example in Figure 20(b) we could insert operations that appear to merge Z's two halves, for example by rotation: ⌜if (P^F) Z = rotate(Z,5)⌝.

A variant of this transformation is to merge $V_1 \cdots V_k$ into an array $V_A = \begin{array}{ccc} \scriptstyle 1 & \scriptstyle\cdots & \scriptstyle k \\ \hline \multicolumn{1}{|c|}{V_1} & \multicolumn{1}{c|}{\cdots} & \multicolumn{1}{c|}{V_k} \\ \hline \end{array}$ of the appropriate type. If $V_1 \cdots V_k$ are object reference variables, for example, then the element type of V_A can be any class that is higher in the inheritance hierarchy than any of the types of $V_1 \cdots V_k$.

7.2.2 Restructure Arrays

A number of transformations can be devised for obscuring operations performed on arrays: we can *split* an array into several sub-arrays, *merge* two or more arrays into one array, *fold* an array (increasing the number of dimensions), or *flatten* an array (decreasing the number of dimensions).

Figure 21 shows some examples of array restructuring. In statements (1-2) an array A is split up into two sub-arrays A1 and A2. A1 holds the elements of A that have *even* indices, and A2 holds the elements with *odd* indices.

Statements (3-4) of Figure 21 show how two integer arrays B and C can be interleaved into a resulting array BC. The elements from B and C are evenly spread over the resulting array.

Statements (6-7) demonstrate how a one-dimensional array `D` can be folded into a two-dimensional array `D1`. Statements (8-9), finally, demonstrate the reverse transformation: a two-dimensional array `E` is flattened into a one-dimensional array `E1`.

Array splitting and folding increase the μ_6 data complexity metric. Array merging and flattening, on the other hand, seem to *decrease* this measure. While this may seem to indicate that these transformations have only marginal or even negative potency, this, in fact, is deceptive. The problem is that the complexity metrics of Table 1 fail to capture an important aspect of some data structure transformations: they introduce structure where there was originally none or they remove structure from the original program. This can greatly increase the obscurity of the program. For example, a programmer who declares a two-dimensional array does so for a purpose: the chosen structure somehow maps cleanly to the data that is being manipulated. If that array is folded into a one-dimensional structure, a reverse engineer will have been deprived of much valuable pragmatic information.

7.2.3 Modify Inheritance Relations

In current object-oriented language such as Java, the main modularization and abstraction concept is the *class*. Classes are essentially abstract data types that encapsulate data (instance variables) and control (methods). We write a class as $C = (V, M)$, where V is the set of C's instance variables and M its methods.

In contrast to the traditional notion of abstract data types, two classes C_1 and C_2 can be composed by *aggregation* (C_2 has an instance variable of type C_1) as well as by *inheritance* (C_2 extends C_1 by adding new methods and instance variables). Borrowing the notation used in [27], we write inheritance as $C_2 = C_1 \oplus \Delta C_2$. C_2 is said to inherit from C_1, its super- or parent class. The $\oplus$ operator is the function that combines the parent class with the new properties defined in ΔC_2. The exact semantics of $\oplus$ depends on the particular programming language. In languages such as Java, $\oplus$ is usually interpreted as *union* when applied to the instance variables and as *overriding* when applied to methods.

According to metric μ_7, the complexity of a class C_1 grows with its *depth* (distance from the root) in the inheritance hierarchy, and the number of its direct descendants. There are two basic ways in which we can increase this complexity: we can split up (factor) a class (Figure 22(a)) or insert a new, bogus, class (Figure 22(b)).

A problem with class factoring is its low resilience; there is nothing stopping a deobfuscator from simply merging the factored classes. To prevent this, factoring and insertion are normally combined as shown in Figure 22(d). Another way of increasing the resilience of these types of transformations is to make sure that new objects are created of all introduced classes.

Figure 22(c) shows a variant of class insertion, called *false refactoring*. Refactoring is a (sometimes automatic) technique for restructuring object-oriented programs whose structure has deteriorated [22]. Refactoring is a two-step process. First, it is detected that two, apparently independent classes, in fact implement similar behavior. Secondly, features common to both classes are moved into a new (possibly abstract) parent class. False refactoring is a similar operation, only it is performed on two classes C_1 and C_2 *that have no common behavior*. If both classes have instance variables of the same type, these can be moved into the new parent class C_3. C_3's methods can be buggy versions of some of the methods from C_1 and C_2.

7.3 Ordering Transformations

In Section 6.4 we argued that (when possible) randomizing the order in which computations are performed is a useful obfuscation. Similarly, it is useful to randomize the order of declarations in the source application. Particularly, we randomize the order of methods and instance variables within classes and formal parameters within methods. In the latter case, the corresponding actuals will of course have to be reordered as well. The potency of these transformations is low and the resilience is one-way.

In many cases it will also be possible to reorder the elements within an array. Simply put, we provide an opaque encoding function $f(i)$ which maps the i:th element in the original array into its new position of the reordered array:

`int i=1, A[1000];`
`while (i < 1000) {`
`  ··· A[i] ···;`
`  i++;`
`}`
	$\stackrel{\mathcal{T}}{\Longrightarrow}$	`int i=1, A[1000];`
		`while (i < 1000) {`
		`  ··· A[`$f(i)$`] ···;`
		`  i++;`
		`}`

8 Opaque Values and Predicates

As we have seen, opaque predicates are the major building block in the design of transformations that obfuscate control flow. In fact, the quality of most control transformations is directly dependent on the quality of such predicates.

In Section 6.1 we gave examples of simple opaque predicates with *trivial* and *weak* resilience. This means that the opaque predicates can be broken (an automatic

```
(1) int A[9];
(2) A[i] = ···;
      ...
      ...
(3) int B[9],C[19];
(4) B[i] = ···;
(5) C[i] = ···;
      ...
(6) int D[9];
(7) for(i=0;i<=8;i++)
      D[i]=2*D[i+1];
      ...
      ...
      ...
(8) int E[2,2];
(9) for(i=0;i<=2;i++)
      for(j=0;i<=2;i++)
        swap(E[i,j], E[j,i]);
```

$\stackrel{\mathcal{T}}{\Longrightarrow}$

```
(1') int A1[4],A2[4];
(2') if ((i%2)==0) A1[i/2]=···
     else A2[i/2]=···;
       ...
(3') int BC[29];
(4') BC[3*i] = ···;
(5') BC[i/2*3+1+i%2] = ···;
       ...
(6') int D1[1,4];
(7') for(j=0;j<=1;j++)
       for(k=0;k<=4;k++)
         if (k==4)
           D1[j,k]=2*D1[j+1,0];
         else
           D1[j,k]=2*D1[j,k+1];
       ...
(8') int E1[8];
(9') for(i=0;i<=8;i++)
       swap(E[i], E[3*(i%3)+i/3]);
```

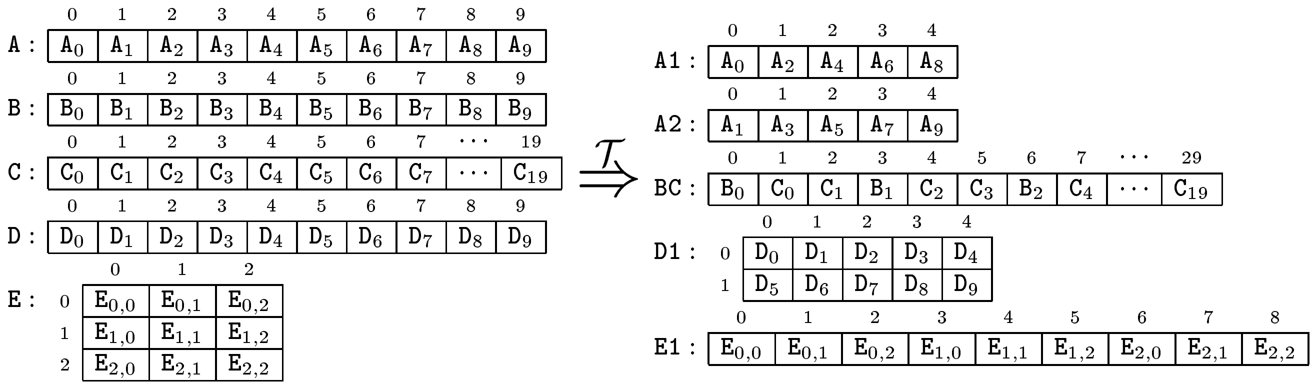


Figure 21: Array Restructuring. Array splitting (statements (1-2)), array merging (statements (3-5)), array folding (statements (6-7)), and array flattening (statements (8-9)).

deobfuscator could determine their value) using local or global static analysis. Obviously, we generally require a much higher resistance to attack. Ideally, we would like to be able to construct opaque predicates that require worst case exponential time (in the size of the program) to break but only polynomial time to construct. In this section we will present two such techniques. The first one is based on aliasing, the second on lightweight processes.

8.1 Opaque Constructs Using Objects and Aliases

Inter-procedural static analysis is significantly complicated whenever there is a possibility of aliasing. In fact, different versions of precise static alias analysis have been shown to be NP-hard [12] or even undecidable [24]. We can exploit this fact to construct opaque predicates which are difficult to break. It should be noted that there are many fast but imprecise alias analysis algorithms that will detect some aliases some of the time, but not all aliases all of the time.

The basic idea is to construct a complex dynamic structure and maintain a set of pointers into this structure. Opaque predicates can then be designed which ask questions that can only be answered if an inter-procedural aliasing analysis has been performed.

Consider the obfuscated method `P` in Figure 23. Interspersed with `P`'s original code are bogus method calls and redundant computations guarded by opaque predicates. The method calls manipulate two global pointers `g` and `h` which point into different connected components (`G` and `H`) of a dynamic structure. The statement ⌜`g=g.Move()`⌝ will non-deterministically update `g` to point somewhere else within `G`. The statement ⌜`h = h.Insert(new Node)`⌝ inserts a new node into `H` and updates `h` to point to some node within `H`. `P` (and other methods that P calls) is given an extra pointer argument f which also refers to objects within G.

This set-up allows us to construct opaque predicates like those of statements 4 and 5 of Figure 23. The predicate f==g may be either true or false since f and g move around within the same component. Conversely, g==h must be false since g and h refer to nodes within different components.

Statements 6–9 in Figure 23 exploit aliasing. The predicate in statement 7 will be true or false depending on whether f and g point to the same or different objects. The predicate in statement 8 must evaluate to true since f and h cannot alias the same object.

8.2 Opaque Constructs Using Threads

Parallel programs are more difficult to analyze statically than their sequential counterparts. The reason is their *interleaving* semantics: n statements in a parallel region ⌜**PAR** S_1 ; S_2 ; $\cdots$; S_n ; **ENDPAR**⌝ can be executed in $n!$ different ways. In spite of this, some static analyses over parallel programs can be performed in polynomial time [15], while others require all $n!$ interleavings to be considered.

In Java, parallel regions are constructed using lightweight processes known as *threads*. Java threads have (from our point of view) two very useful properties: (1) their scheduling policy is not specified strictly by the language specification and will hence depend on the implementation, and (2) the actual scheduling of a thread will depend on asynchronous events generated by user interaction, network trafic, etc. Combined with the inherent interleaving semantics of parallel regions, this means that threads are very difficult to analyze statically.

We will use these observations to create opaque predicates that will require worst-case exponential time to break. The basic idea is very similar to the one used in Section 8.1: a global data structure V is created and occasionally updated, but kept in a state such that opaque queries can be made. The difference is that V is updated by concurrently executing threads.

Obviously, V can be a dynamic data structure such as the one created in Figure 23. The threads would randomly move the global pointers g and h around in their respective components, by asynchronously executing calls to move and insert. This has the advantage of combining data races with interleaving and aliasing effects, for very high degrees of resilience.

In Figure 24 we illustrate these ideas with a much simpler example where V is a pair of global integer variables X and Y. It is based on the well-known fact from elementary number theory that, for any integers x and y, $7y^2 - 1 \neq x^2$.

```
thread S {                    thread T {
  int R;                        int R;
  while (1) {                   while (1) {
    R = random(1,C);              R = random(1,C);
    X = R*R;                      Y = 7*R*R;
    sleep(3);                     sleep(2);
  }                               X *= X;
}                                 sleep(5);
                                }
                              }
int X, Y;
const C = sqrt(maxint)/10;
main () {
  S.run(); T.run();
     ...
  if ((Y - 1) == X)^F <= [p]
     ...
}
```

Figure 24: In this example, the predicate at point $\boxed{\texttt{p}}$ will always evaluate to False. Two threads S and T occasionally wake up to update global variables X and Y with new random values. Notice that S and T are involved in a data-race on X, but that this does not matter as long as assignments are atomic. Regardless of whether S or T wins the race, X will hold the square of a number.

9 Deobfuscation and Preventive Transformations

Many of our obfuscating transformations (particularly the control transformations of Section 6.2) can be said to embed a bogus program within a real program. In other words, an obfuscated application really consists of two programs merged into one: a real program which performs a useful task and a bogus program which computes useless information. The sole purpose of the bogus program is to confuse potential reverse engineers by hiding the real program behind irrelevant code.

The opaque predicate is the main device the obfuscator has at its disposal to prevent the bogus inner program from being easily identified and removed. For example, in Figure 25(a), an obfuscator embeds bogus code protected by opaque predicates within three statements of a real program. A deobfuscator's task is to examine the obfuscated application and automatically identify and remove the inner bogus program. To accomplish this, the deobfuscator must first *identify* and then *evaluate* opaque constructs. This process is illustrated in Figure 25(b-d).

Figure 26 shows the anatomy of a semi-automatic deobfuscation tool. It incorporates a number of techniques that are well known in the reverse engineering community. In the remainder of this section we will briefly review some of these techniques and discuss various counter-measures (so called *preventive transformations*) that an obfuscator can employ to make deobfuscation more difficult.

9.1 Preventive Transformations

Preventive transformations (Figure 1(g)) are quite different in flavor from control or data transformations. In contrast to these, their main goal is not to obscure the program to a human reader. Rather, they are designed to make known automatic deobfuscation techniques more difficult (*inherent* preventive transformations), or to explore known problems in current deobfuscators or decompilers (*targeted* preventive transformations).

9.1.1 Inherent Preventive Transformations

Inherent preventive transformations will generally have low potency and high resilience. Most importantly, they will have the ability to *boost* the resilience of other transformations. As an example, assume that we have reordered a for-loop to run backwards, as suggested in section 6.4. We were able to apply this transformation only because we could determine that the loop had no loop-carried data dependencies. Naturally, there is nothing stopping a deobfuscator from performing the same analysis and then returning the loop to forward execution. To prevent this, we can add a bogus data dependency to the reversed loop:

```
                           int B[50];
for(i=1;i<=10;i++)   T     for(i=10;i>=1;i--){
  A[i]=i            ==>      A[i]=i;
                             B[i]+=B[i*i/2]
                           }
```

The resilience this *inherent preventive* transformation adds to the loop reordering transformation depends on the complexity of the bogus dependency and the state-of-the-art in dependency analysis [33].

9.1.2 Targeted Preventive Transformations

As an example of a targeted preventive transformation, consider the `HoseMocha` [16] program. It was designed specifically to explore a weakness in the `Mocha` [30] decompiler. `HoseMocha` inserts extra instructions after every `return`-statement in every method in the source program. This transformation has no effect on the behavior of the application, but it is enough to make `Mocha` crash.

9.2 Identifying and Evaluating Opaque Constructs

The most difficult part of deobfuscation is *identifying* and *evaluating* opaque constructs. Note that identification and evaluation are distinct activities. An opaque construct can be *local* (contained within a single basic block), *global* (contained within a single procedure), or *inter-procedural* (distributed throughout the entire program). For example, $\ulcorner$`if` $(\mathtt{x}*\mathtt{x} == (7*\mathtt{y}*\mathtt{y}-1)^F)\ \cdots\urcorner$ is a local opaque predicate, whereas $\ulcorner$`R=x*x;`$\cdots$`;S=7*y*y-1;`$\cdots$`;if` $(\mathtt{R} == \mathtt{S}^F)\cdots\urcorner$ is global. If the computation of `R` and `S` were performed in different procedures, the construct would be inter-procedural. Obviously, identification of a local opaque predicate is easier than identification of an inter-procedural one.

9.3 Identification by Pattern Matching

A deobfuscator can use knowledge of the strategies employed by known obfuscators to identify opaque predicates. A designer of a deobfuscator could examine an obfuscator (either by decompiling it or simply by examining the obfuscated code it generates) and construct pattern-matching rules that can identify commonly used opaque predicates. This method will work best for simple local predicates, such as $\ulcorner\mathtt{x}*\mathtt{x} == (7*\mathtt{y}*\mathtt{y}-1)^F\urcorner$ or $\ulcorner\mathtt{random}(1,5) < 0^F\urcorner$.

To thwart attempts at pattern matching, the obfuscator should avoid using canned opaque constructs. It is also important to choose opaque constructs that are syntactically similar to the constructs used in the real application.

9.4 Identification by Program Slicing

The basic premise of this paper is that a programmer will find the obfuscated version of a program more difficult to understand and reverse engineer than the original one. The main reasons are that in the obfuscated program (a) live "real" code will be interspersed with dead bogus code and (b) logically related pieces of code will have been broken up and dispersed over the program. Program slicing tools can be used by a reverse engineer to counter these obfuscations. Such tools can interactively aid the engineer to decompose a program into manageable chunks called *slices*. A slice of a program P with respect to a point p and a variable v consists of all the statements of P that could have contributed to v's value at p. Hence, a program slicer would be able to extract from the obfuscated program the statements of the algorithm that computes an opaque variable v, even if the obfuscator has dispersed these statements over the entire program.

There are several strategies available to an obfuscator to make slicing a less useful identification tool:

Add parameter aliases A parameter alias is two formal parameters (or a formal parameter and a global variable) that refer to the same memory location. The cost of precise inter-procedural slicing grows with the number of potential aliases in a program, which in turn grows *exponentially* with the number of formal parameters [13]. Hence, if the obfuscator adds aliased dummy parameters to a program it will either substantially slow down the slicer (if precise slices are required), or force the slicer to produce imprecise slices (if fast slicing is required).

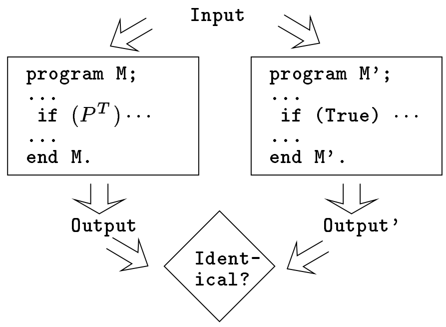
Add variable dependencies Popular slicing tools such as `Unravel` [17] work well for small slices, but will sometimes require excessive time to compute larger ones. For example, when working on a 4000 line C program `Unravel` in some cases required over 30 minutes to compute a slice. To force this behavior, the obfuscator should attempt to increase slice sizes, by adding bogus variable dependencies. In the example below, we have increased the size of the slice computing `x` by adding two statements which apparently contribute to `x`'s value, but which, in fact, do not.

```
main() {                      main() {
   int x=1;          T           int x=1;
   x = x * 3;       ==>          if (P^F) x++;
}                                x = x + V^0;
                                 x = x * 3;
                              }
```

9.5 Statistical Analysis

A deobfuscator can instrument an obfuscated program to analyze the outcome of all predicates. We will call any deobfuscation method that examines the run-time characteristics of an obfuscated application in this way *Statistical Analysis*. The deobfuscator would alert the reverse engineer to any predicate that always returns the same truth value over a large number of test runs, since they may turn out to be an opaque P^T (P^F) predicate. The deobfuscator could not blindly replace such predicates with `True` (`False`), since this would be too dangerous. Many applications will contain "real" predicates that check for conditions that only happen under exceptional circumstances, and to the deobfuscator they will appear to behave identically to an opaque predicate. As an example, consider ⌜`if (`*Leap Year*`) ···`⌝.

Statistical analysis can also be used for evaluation. When a potential opaque predicate (P^T, say) in a program `M` has been identified, we guess its value (`True`), and make a version `M'` of the obfuscated program where the opaque predicate has been replaced by the guessed value. We then run `M` and `M'` in parallel on the same input, and compare to see that they produce identical output. If the outputs are the same, we can conclude that the predicate was part of the bogus, not the real, application:



Obviously, we have to make sure that our chosen inputs adequately cover all paths in the program. Again, if the program contains paths that are rarely taken (⌜`if (`*Leap Year*`) ···`⌝) this will be difficult. Furthermore, generating large numbers of correct input/output data is very difficult, particularly when internal structure of the application is unknown, or the input is entered (as is often the case with Java programs) through a complex graphical user interface.

To prevent identification by statistical analysis, the obfuscator may choose to favor transformations that insert $P^?$ predicates (such as the one shown in Figure 10(b)) over those that insert P^T or P^F predicates.

Another possible counter-measure against statistical analysis is to design opaque predicates in such a way that several predicates have to be cracked at the same time. One way of doing this is to let the opaque predicates have side-effects. In the example below the obfuscator has determined (through some sort of static flow analysis) that statements S_1 and S_2 must always execute the same number of times. The statements are obfuscated by introducing opaque predicates which are calls to functions $\mathtt{Q}_1$ and $\mathtt{Q}_2$. $\mathtt{Q}_1$ and $\mathtt{Q}_2$ increment and decrement a global variable `k`:

```
                       int k=0;
                       bool Q1(x) {
                          k+=2^31; return (P1^T)}
{                      bool Q2(x) {
   S1;          T         k-=2^31; return (P2^T)}
      ...      ==>
   S2;                 {
}                      if (Q1(j)^T) S1;
                             ...
                       if (Q2(k)^T) S2;
                       }
```

If the deobfuscator tries to replace one (but not both) predicates with `True`, `k` will overflow. As a result, the deobfuscated program will terminate with an error condition.

9.6 Evaluation by Data-Flow Analysis

Deobfuscation is similar to many types of code optimization. Removing ⌜`if (False) ···`⌝ is *dead code elimination* and moving identical code from if-statement branches (e.g. S_1 and S_1' in Figure 25) is *code hoisting*, both common code optimization techniques.

When an opaque construct has been identified we can attempt to evaluate it. In simple cases constant propagation using a reaching definition data-flow analysis can be sufficient: ⌜`x=5;···;y=7;···;if (x*x==(7*y*y-1)) ···`⌝.

9.7 Evaluation by Theorem Proving

If data-flow analysis is not powerful enough to break the opaque predicate, a deobfuscator can attempt to use a theorem prover. Whether this is doable or not depends on the power of state-of-the-art theorem provers (which is difficult to ascertain) and the complexity of the theorem that needs to be proven. Certainly, theorems that can be proved by induction (such as $x^2(x+1)^2 \equiv 0 \pmod 4$), are well within reach of current theorem provers.

To make things more difficult, we can use theorems which are know to be difficult to prove, or for which no known proof exists. In the example below the deobfuscator will have to prove that the bogus loop always terminates in order to determine that S_2 is live code:

```
                 {
                 S1;
{        T       n = random(1,2^32);
S1;     ==>      do
S2;                 n = ((n%2)!=0)?3*n+1:n/2
}                while (n>1);
                 S2;
                 }
```

This is known as the Collatz problem. A conjecture says that the loop will always terminate. Although there is no known proof of this conjecture, the code is known to terminate for all numbers up to $7 \cdot 10^{11}$. Thus this obfuscation is safe (the original and obfuscated code behave identically), but difficult to deobfuscate.

9.8 Deobfuscation and Partial Evaluation

Deobfuscation also bears a striking resemblance to *partial evaluation* [14]. A partial evaluator splits a program into two parts: the *static* part which can be precomputed by the partial evaluator, and the *dynamic* part which is executed at runtime. The dynamic part would correspond to our original, unobfuscated, program. The static part would correspond to our bogus inner program, which, if it were identified, could be evaluated and removed at deobfuscation time.

Like all other static inter-procedural analysis methods, partial evaluation is sensitive to aliasing. Hence, the same preventive transformations that were discussed in relation to slicing also applies to partial evaluation.

10 Obfuscation Algorithms

Given the obfuscator architecture of Section 3, the definition of obfuscation quality in Section 5, and the discussion of various obfuscating transformations in Sections 6 to 9, we are now in a position to present more detailed algorithms.

The top-level loop of an obfuscation tool will have this general structure:

```
WHILE NOT Done(A) DO
   S := SelectCode(A);
   T := SelectTransform(S);
   A := Apply(T,S);
END;
```

`SelectCode` returns the next *source code object*[6] to be obfuscated. `SelectTransform` returns the transformation which should be used to obfuscate the particular source code object. `Apply` applies the transformation to the source code object and updates the application accordingly. `Done` determines when the required level of obfuscation has been attained. The complexity of these functions will depend on the sophistication of the obfuscation tool. At the simplistic end of the scale, `SelectCode` and `SelectTransform` could simply return random source code object/transformations, and `Done` could terminate the loop when the size of the application exceeds a certain limit. Normally, such behavior is insufficient.

Algorithm 1 gives a description of a code obfuscation tool with a much more sophisticated selection and termination behavior. The algorithm makes use of several data structures which are constructed by Algorithms 5, 6, and 7:

P_s For each source code object S, $P_s(S)$ is the set of language constructs the programmer used in S. $P_s(S)$ is used to find appropriate obfuscating transformations for S.

A For each source code object S, $A(S) = \{\mathcal{T}_i \mapsto V_1, \cdots, \mathcal{T}_n \mapsto V_n\}$ is a mapping from transformations $\mathcal{T}_i$ to values V_i, describing how appropriate

[6]In the following, the term *source code object* will refer to the classes, methods, basic blocks, etc. that make up an application, as well as the application itself.

it would be to apply $\mathcal{T}_i$ to S. The idea is that certain transformations may be inappropriate for a particular source code object S, because they introduce new code which is "unnatural" to S. The new code would look out of place in S and hence would be easy to spot for a reverse engineer. The higher the appropriateness value V_i the better the code introduced by transformation $\mathcal{T}_i$ will fit in.

I For each source code object S, $I(S)$ is the *obfuscation priority* of S. $I(S)$ describes how *important* it is to obfuscate the contents of S. If S contains an important trade secret then $I(S)$ will be high, if it contains mainly "bread-and-butter" code $I(S)$ will be low.

R For each routine M, $R(M)$ is the execution time *rank* of M. $R(M) = 1$ if more time is spent executing M than any other routine.

The primary input to Algorithm 1 is an application $\mathcal{A}$ and a set of obfuscating transformations $\{\mathcal{T}_1, \mathcal{T}_2, \cdots\}$. The algorithm also requires information regarding each transformation, particularly three quality functions $\mathcal{T}_{\text{res}}(S)$, $\mathcal{T}_{\text{pot}}(S)$, and $\mathcal{T}_{\text{cost}}(S)$ (similar to their namesakes in Section 5, but returning numerical values) and a function P_t:

$\mathcal{T}_{\text{res}}(S)$ returns a measure of the resilience of transformation $\mathcal{T}$ when applied to source code object S, i.e. how well $\mathcal{T}$ will withstand an attack from an automatic deobfuscator.

$\mathcal{T}_{\text{pot}}(S)$ returns a measure of the potency of transformation $\mathcal{T}$ when applied to source code object S, i.e. how much more difficult S will be for a human to understand after having been obfuscated by $\mathcal{T}$.

$\mathcal{T}_{\text{cost}}(S)$ returns a measure of the execution time and space penalty added by $\mathcal{T}$ to S.

P_t maps each transformation $\mathcal{T}$ to the set of language constructs that $\mathcal{T}$ will add to the application.

Points 1 to 3 of Algorithm 1 load the application to be obfuscated, and builds appropriate internal data structures. Point 4 builds $P_s(S)$, $A(S)$, $I(S)$, and $R(M)$. Point 5 applies obfuscating transformations until the required obfuscation level has been attained or until the maximum execution time penalty is exceeded. Point 6, finally, rewrites the new application $\mathcal{A}'$.

ALGORITHM 1 (CODE OBFUSCATION)

input:
- a) An application $\mathcal{A}$ made up of source code or object code files $C_1, C_2, \cdots$.
- b) The standard libraries $L_1, L_2, \cdots$ defined by the language.
- c) A set of obfuscating transformations $\{\mathcal{T}_1, \mathcal{T}_2, \cdots\}$.
- d) A mapping P_t which, for each transformation $\mathcal{T}$ gives the set of language constructs that $\mathcal{T}$ will add to the application.
- e) Three functions $\mathcal{T}_{\text{res}}(S)$, $\mathcal{T}_{\text{pot}}(S)$, and $\mathcal{T}_{\text{cost}}(S)$ expressing the quality of a transformation $\mathcal{T}$ with respect to a source code object S.
- f) A set of input data $I = \{I_1, I_2, \cdots\}$ to $\mathcal{A}$.
- g) Two numeric values `AcceptCost>0` and `ReqObf>0`. `AcceptCost` is a measure of the maximum extra execution time/space penalty the user will accept. `ReqObf` is a measure of the amount of obfuscation required by the user.

output: An obfuscated application $\mathcal{A}'$ made up of source code or object code files.

1. Load the application $C_1, C_2, \cdots$ to be obfuscated. The obfuscator could either
   (a) load source code files, in which case the obfuscator would have to contain a complete compiler front-end performing lexical, syntactic, and semantic analysis,[7] or
   (b) load object code files. If the object code retains most or all of the information in the source code (as is the case with Java class files), this method is preferable.
2. Load library code files $L_1, L_2, \cdots$ referenced directly or indirectly by the application.
3. Build an internal representation of the application. The choice of internal representation depends on the structure of the source language and the complexity of the transformations the obfuscator implements. A typical set of data structures might include:
   (a) A control-flow graph for each routine in $\mathcal{A}$.
   (b) A call-graph for the routines in $\mathcal{A}$.
   (c) An inheritance graph for the classes in $\mathcal{A}$.

[7] A less powerful obfuscator that restricts itself to purely syntactic transformations could manage without semantic analysis.

4. Construct mappings $R(M)$ and $P_s(S)$ (using Algorithm 5), $I(S)$ (using Algorithm 6), and $A(S)$ (using Algorithm 7).

5. Apply the obfuscating transformations to the application. At each step we select a source code object S to be obfuscated and a suitable transformation $\mathcal{T}$ to apply to S. The process terminates when the required obfuscation level has been reached or the acceptable execution time cost has been exceeded.
   REPEAT
   S := SelectCode(I);
   $\mathcal{T}$:= SelectTransform(S,A);
   Apply $\mathcal{T}$ to S and update relevant data structures from point 3;
   UNTIL Done(ReqObf, AcceptCost, S, $\mathcal{T}$, I)

6. Reconstitute the obfuscated source code objects into a new obfuscated application, $\mathcal{A}'$.

□

Algorithm 2 (SelectCode)

input: The obfuscation priority mapping I as computed by Algorithm 6.

output: A source code object S.

I maps each source code object S to $I(S)$, which is a measure of how important it is to obfuscate S. To select the next source code object to obfuscate, we can treat I as a priority queue. In other words, we select S so that $I(S)$ is maximized. □

Algorithm 3 (SelectTransform)

input: a) A source code object S.
b) The appropriateness mapping A as computed by Algorithm 7.

output: A transformation $\mathcal{T}$.

Any number of heuristics can be used to select the most suitable transformation to apply to a particular source code object S. However, there are two important issues to consider. Firstly, the chosen transformation must blend in naturally with the rest of the code in S. This can be handled by favoring transformations with a high appropriateness value in $A(S)$. Secondly, we want to favor transformations which yield a high 'bang-for-the-buck', i.e. high levels of obfuscation with low execution time penalty. This is accomplished by selecting transformations that maximize potency and resilience, and minimize cost. These heuristics are captured by the following code, where $\omega_1, \omega_2, \omega_3$ are implementation-defined constants:

Return a transform $\mathcal{T}$, such that
$\mathcal{T} \mapsto V \in A(S)$, and
$\frac{\omega_1 \mathcal{T}_{\text{pot}}(S)+\omega_2 \mathcal{T}_{\text{res}}(S)+\omega_3 V}{\mathcal{T}_{\text{cost}}(S)}$ is maximized;

□

Algorithm 4 (Done)

input: a) ReqObf, the remaining level of obfuscation.
b) AcceptCost, the remaining acceptable execution time penalty.
c) A source code object S.
d) A transformation $\mathcal{T}$.
e) The obfuscation priority mapping I.

output: a) An updated ReqObf.
b) An updated AcceptCost.
c) An updated obfuscation priority mapping I.
d) A boolean return value which is TRUE if the termination condition has been reached.

The Done function serves two purposes. It updates the priority queue I to reflect the fact that the source code object S has been obfuscated, and should receive a reduced priority value. The reduction is based on a combination of the resilience and potency of the transformation. Done also updates ReqObf and AcceptCost, and determines whether the termination condition has been reached. $\omega_1, \omega_2, \omega_3, \omega_4$ are implementation-defined constants:

$I(S)$:= $I(S) - (\omega_1 \mathcal{T}_{\text{pot}}(S) + \omega_2 \mathcal{T}_{\text{res}}(S))$;
ReqObf := ReqObf - $(\omega_3 \mathcal{T}_{\text{pot}}(S) + \omega_4 \mathcal{T}_{\text{res}}(S))$;
AcceptCost := AcceptCost - $\mathcal{T}_{\text{cost}}$(S);
RETURN AcceptCost$\leq$0 **OR** ReqObf$\leq$0;

□

Algorithm 5 (Pragmatic Information)

input: a) An application $\mathcal{A}$.
b) A set of input data $I = \{I_1, I_2, \cdots\}$ to $\mathcal{A}$.

output: a) A mapping $R(M)$ which, for every routine M in $\mathcal{A}$, gives the execution time *rank* of M.
b) A mapping $P_s(S)$, which, for every source code object S in $\mathcal{A}$, gives the set of language constructs used in S.

Compute pragmatic information. This information will be used to choose the right type of transformation for each particular source code object.

1. Compute dynamic pragmatic information. I.e. run the application under a profiler on the input data set I provided by the user. Compute $R(M)$ (the execution time *rank* of M) for each routine/basic block, indicating where the application spends most of its time.

2. Compute static pragmatic information $P_s(S)$. $P_s(S)$ provides statistics on the kinds of lan-

guage constructs the programmer used in S.

FOR S := each source code object in $\mathcal{A}$ **DO**
  O := The set of operators that S uses;
  C := The set of high-level language constructs (**WHILE** statements, exceptions, threads, etc.) that S uses;
  L := The set of library classes/routines that S references;
  $P_s(S) := O \cup C \cup L$;
END FOR

□

ALGORITHM 6 (OBFUSCATION PRIORITY)

input: a) An application $\mathcal{A}$.
b) $R(M)$, the rank of M.

output: A mapping $I(S)$ which, for each source code object S in $\mathcal{A}$, gives the *obfuscation priority* of S.

$I(S)$ can be provided explicitly by the user, or it can be computed using a heuristic based on the statistical data gathered in Algorithm 5. Possible heuristics might be:

1. For any routine M in $\mathcal{A}$, let $I(M)$ be inversely proportional to the rank of M, $R(M)$. I.e. the idea is that "if much time is spent executing a routine M, then M is probably an important procedure that should be heavily obfuscated."

2. Let $I(S)$ be the complexity of S, as defined by one of the software complexity metrics in Table 1. Again, the (possibly flawed) intuition is that complex code is more likely to contain important trade secrets than simple code.

□

ALGORITHM 7 (OBFUSCATION APPROPRIATENESS)

input: a) An application $\mathcal{A}$.
b) A mapping P_t which, for each transformation $\mathcal{T}$, gives the set of language constructs $\mathcal{T}$ will add to the application.
c) A mapping $P_s(S)$ which, for each source code object S in $\mathcal{A}$, gives the set of language constructs used in S.

output: A mapping $A(S)$ which, for each source code object S in $\mathcal{A}$ and each transformation $\mathcal{T}$, gives the *appropriateness* of $\mathcal{T}$ with respect to S.

Compute the *appropriateness* set $A(S)$ for each source code object S. The mapping is based primarily on the static pragmatic information computed in Algorithm 5.

FOR S := each source code object in $\mathcal{A}$ **DO**
  FOR $\mathcal{T}$:= each transformation **DO**
    V := degree of similarity between $P_t(\mathcal{T})$ and $P_s(S)$;
    $A(S) := A(S) \cup \{\mathcal{T} \mapsto V\}$;
  END FOR
END FOR

□

11 Summary and Discussion

The main contribution of this paper is the insight that it may under many circumstances be acceptable for an obfuscated program to behave differently than the original one. In particular, most of our obfuscating transformations make the target program slower or larger than the original. In special cases we even allow the target program to have different side-effects than the original, or not to terminate when the original program terminates with an error condition. Our only requirement is that the *observable behavior* (the behavior as experienced by a user) of the two programs should be identical.

Allowing such weak equivalence between original and obfuscated program is a novel and very exciting idea. It is our belief that the current paper only identifies some of the more obvious transformations, and that there is great potential for much future research. In particular, we would like to see the following areas investigated:

1. New obfuscating transformations should be identified.

2. The interaction and ordering between different transformations should be studied. This is similar to work in code optimization, where the ordering of a sequence of optimizing transformations has always been a difficult problem.

3. The relationship between potency and cost should be studied. For a particular kind of code we would like to know which transformations would give the best "bang-for-the-buck", i.e. the highest potency at the lowest execution overhead.

For an overview of all the transformations that have been discussed in the paper, see Tables 2 and 3. For an overview of the opaque constructs that have been suggested, see Table 4.

11.1 The Power of Obfuscation

Encryption and program obfuscation bear a striking resemblance to each other. Not only do both try to hide information from prying eyes, they also purport to do so for a limited time only. An encrypted document has a limited shelf-life: it is safe only for as long as the

Target	Obfuscation Operation	Transformation	Quality Potency	Resilience	Cost	Metrics	Section
Layout		Scramble Identifiers	medium	one-way	free		5.5
		Change Formatting	low	one-way	free		5.5
		Remove Comments	high	one-way	free		5.5
Control	Computations	Insert Dead or Irrelevant Code	Depends on the quality of the opaque predicate and the nesting depth at which the construct is inserted.			μ_1,μ_2,μ_3	6.2.1
		Extend Loop Condition				μ_1,μ_2,μ_3	6.2.2
		Reducible to Non-Reducible				μ_1,μ_2,μ_3	6.2.3
		Add Redundant Operands				μ_1	6.2.6
		Remove Programming Idioms	medium	strong	†	μ_1	6.2.4
		Table Interpretation	high	strong	costly	μ_1	6.2.5
		Parallelize Code	high	strong	costly	μ_1,μ_2	6.2.7
	Aggregation	Inline Method	medium	one-way	free	μ_1	6.3.1
		Outline Statements	medium	strong	free	μ_1	6.3.1
		Interleave Methods	Depends on the quality of the opaque predicate.			μ_1,μ_2,μ_5	6.3.2
		Clone Methods				$\mu_1,\mu_7^{\text{a-c,e}}$	6.3.3
		Block loop	low	weak	free	μ_1,μ_2	6.3.4
		Unroll Loop	low	weak	cheap	μ_1	6.3.4
		Loop Fission	low	weak	free	μ_1,μ_2	6.3.4
	Ordering	Reorder Statements	low	one-way	free		6.4
		Reorder Loops	low	one-way	free		6.4
		Reorder Expression	low	one-way	free		6.4
Data	Storage & Encoding	Change Encoding	Depends on the complexity of the encoding function.			μ_1	7.1.1
		Promote Scalar to Object	low	strong	free		7.1.2
		Change Variable Lifetime	low	strong	free	μ_4	7.1.2
		Split Variable	Depends on the number of variables into which the original variable is split.			μ_1	7.1.3
		Convert Static to Procedural Data	Depends on the complexity of the generated function.			μ_1,μ_2	7.1.4
	Aggregation	Merge Scalar Variables	low	weak	free	μ_1	7.2.1
		Factor Class	medium	†	free	$\mu_1,\mu_7^{\text{b,c,e}}$	7.2.3
		Insert Bogus Class	medium	†	free	$\mu_1,\mu_7^{\text{b,c}}$	7.2.3
		Refactor Class	medium	†	free	$\mu_1,\mu_7^{\text{b,c,e}}$	7.2.3
		Split Array	†	weak	free	μ_1,μ_2,μ_6	7.2.2
		Merge Arrays	†	weak	free	μ_1,μ_2	7.2.2
		Fold Array	†	weak	cheap	μ_1,μ_2,μ_6,μ_3	7.2.2
		Flatten Array	†	weak	free		7.2.2
	Ordering	Reorder Methods & Instance Variables	low	one-way	free		7.3
		Reorder Arrays	low	weak	free		7.3

Table 2: Table of Transformations (Part A). A † in any of the quality columns indicates that the measure is dependent on circumstances which are discussed indepth in the corresponding section. The Metrics column lists the complexity measures affected by each transformation. See Table 1 for descriptions of the measures.

Target	Obfuscation Operation	Transformation	Quality Potency	Resilience	Cost	Metrics	Section
Preventive	Targeted	`HoseMocha`	low	trivial	free	μ_1	9
	Inherent	Add Aliased Formals to Prevent Slicing	medium	strong	free	$\mu_1\mu_5$	9.4
		Add Variable Dependencies to Prevent Slicing	Depends on the quality of the opaque predicate.			μ_1	9.4
		Add Bogus Data Dependencies	medium	weak	cheap	μ_1	9.1.1
		Use Opaque Predicates with Side-Effects	medium	weak	free	μ_1	9.5
		Make Opaque Predicates using Difficult Theorems	†	†	†	μ_1	9.5

Table 3: Table of Transformations (Part B).

Opaque Construct	Quality Resilience	Cost	Section
Created from calls to library functions.	trivial	Depends on the cost of the library function.	6.1.1
Created from local (intra-basic block) information.	trivial	free ... cheap	6.1.1
Created from global (inter-basic block) information.	weak	free ... cheap	6.1.1
Created from inter-procedural and aliasing information	full	cheap ... costly	8.1
Created from process interaction and scheduling	full	cheap ... costly	8.2

Table 4: Table of opaque constructs.

encryption algorithm itself withstands attack, and for as long as advances in hardware speed do not allow messages for the chosen key-length to be routinely decrypted. The same is true for an obfuscated application; it remains secret only for as long as sufficiently powerful deobfuscators have yet to be built.

For evolving applications this will not be a problem, as long as the time between releases is *shorter* than the time it takes for the deobfuscator to catch up with the obfuscator. If this is the case, then by the time an application can be automatically deobfuscated it is already outdated and of no interest to a competitor.

However, if an application contains trade secrets that can be assumed to survive several releases, then these should be protected by means other than obfuscation. Partial server-side execution (Figure 2(b)) seems the obvious choice, but has the drawback that the application will execute slowly or (when the network connection is down) not at all.

11.2 Other Uses of Obfuscation

It is interesting to note that there may be potential applications of obfuscation other than the obvious one we have been discussing. One possibility is to use obfuscation in order to trace software pirates. The idea is simple: A vendor creates a new obfuscated version of his application for every new customer[8] and keeps a record of to whom each version was sold. This is probably only reasonable if the application is being sold and distributed over the net. If the vendor finds out that his application is being pirated, all he needs to do is to get a copy of the pirated version, compare it against the data base, and see who bought the original application.[9]

[8] We can generate different obfuscated versions of the same application by introducing an element of randomness into the `SelectTransform` algorithm (Algorithm 3). Different seeds to the random number generator will produce different versions.

[9] It is, in fact, not necessary to store a copy of every obfuscated version sold. It suffices to keep the random number seed that was

Software pirates could themselves make (illicit) use of obfuscation. Since the Java obfuscator we outlined in Figure 6 works at the bytecode level, there is nothing stopping a pirate from obfuscating a legally bought Java application. The obfuscated version could then be resold. When faced with litigation the pirate could argue that he is, in fact, not reselling the application that he originally bought (after all, the code is completely different!), but rather a legally reengineered version.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986. ISBN 0-201-10088-6.

[2] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994. `http://www.acm.org/pubs/toc/Abstracts/0360-0300/197406.html`.

[3] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[4] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Software – Practice & Experience*, 25(7):811–829, July 1995.

[5] Jeffrey Dean. *Whole-Program Optimization of Object-Oriented Languages*. PhD thesis, University of Washington, 1996.

[6] James R. Gosler. Software protection: Myth or reality? In *CRYPTO'85 — Advances in Cryptology*, pages 140–157, August 1985.

[7] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996. ISBN 0-201-63451-1.

[8] M. H. Halstead. *Elements of Software Science*. Elsevier North-Holland, 1977.

[9] Warren A. Harrison and Kenneth I. Magel. A complexity measure based on nesting level. *SIGPLAN Notices*, 16(3):63–74, 1981.

[10] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, September 1981.

used and a hash-value computed from the obfuscated code.

[11] Amir Herzberg and Shlomit S. Pinter. Public protection of software. *ACM Transactions on Computer Systems*, 5(4):371–393, November 1987.

[12] Susan Horwitz. Precise flow-insensitive May-Alias analysis is NP-hard. *TOPLAS*, 19(1):1–6, January 1997.

[13] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *TOPLAS*, 12(1):26–60, January 1990.

[14] Neil D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, September 1996.

[15] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *TOPLAS*, 18(3):268–299, May 1996.

[16] Mark D. LaDue. HoseMocha. `http://www.xynyx.demon.nl/java/HoseMocha.java`, January 1997.

[17] James R. Lyle, Dolorres R. Wallace, James R. Graham, Keith B. Gallagher, Joseph P. Poole, and David W Binkley. Unravel: A CASE tool to assist evaluation of high integrity software. Volume 1: Requirements and design. Technical Report NISTIR 5691, U.S. Department of Commerce, August 1995.

[18] Stavros Macrakis. Protecting source code with ANDF. `ftp://riftp.osf.org/pub/andf/andf_coll_papers/ProtectingSourceCode%.ps`, January 1993.

[19] Apple's QuickTime lawsuit. `http://www.macworld.com/pages/june.95/News.848.html` and `may.95/News.705.html`, May–June 1995.

[20] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[21] John C. Munson and Taghi M. Kohshgoftaar. Measurement of data structure complexity. *Journal of Systems Software*, 20:217–225, 1993.

[22] William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring. In Stan C. Kwasny and John F. Buck, editors, *Proceedings of the 21st Annual Conference on Computer Science*, pages 66–73, New York, NY, USA, February 1993. ACM Press. `ftp://st.cs.uiuc.edu/pub/papers/refactoring/refactoring-superclass%es.ps`.

[23] E. I. Oviedo. Control flow, data flow, and program complexity. In *Proceedings of IEEE COMPSAC*, pages 146–152, November 1980.

[24] G. Ramalingam. The undecidability of aliasing. *TOPLAS*, 16(5):1467–1471, September 1997.

[25] Spencer Rugaber, Kurt Stirewalt, and Linda M. Wills. The interleaving problem in program understanding. In *2nd Working Conference on Reverse Engineering*, Toronto, Ontario, Canada, July 1995. `ftp.cc.gatech.edu//pub/groups/reverse/repository/interleaving.ps`.

[26] Pamela Samuelson. Reverse-engineering someone else's software: Is it legal? *IEEE Software*, pages 90–96, January 1990.

[27] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.

[28] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[29] Hans Peter Van Vliet. Crema — The Java obfuscator. `http://web.inter.nl.net/users/H.P.van.Vliet/crema.html`, January 1996.

[30] Hans Peter Van Vliet. Mocha — The Java decompiler. `http://web.inter.nl.net/users/H.P.van.Vliet/mocha.html`, January 1996.

[31] Uwe G. Wilhelm. Cryptographically protected objects. `http://lsewww.epfl.ch/~wilhelm/CryPO.html`, 1997.

[32] Linda Mary Wills. Automated program recognition: a feasibility demonstration. *Artificial Intelligence*, 45(1–2):113–172, 1990.

[33] Michael Wolfe. *High Performance Compilers For Parallel Computing*. Addison-Wesley, 1996. ISBN 0-8053-2730-4.

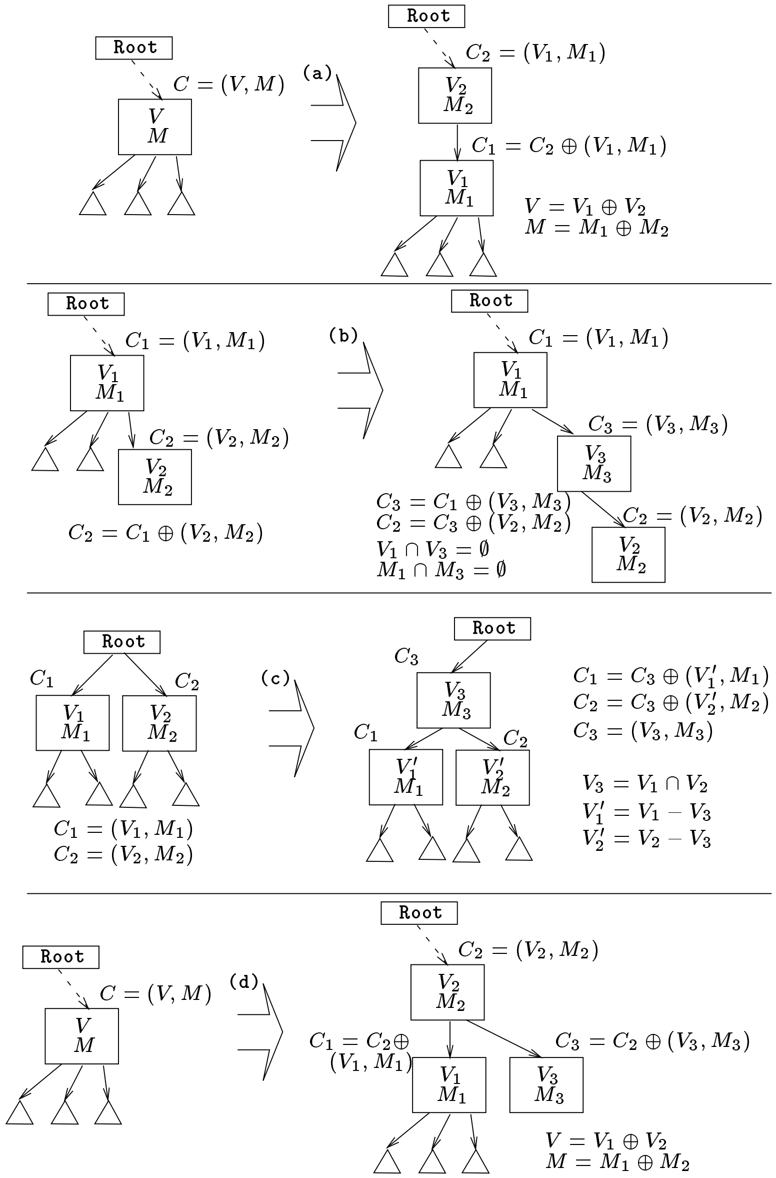
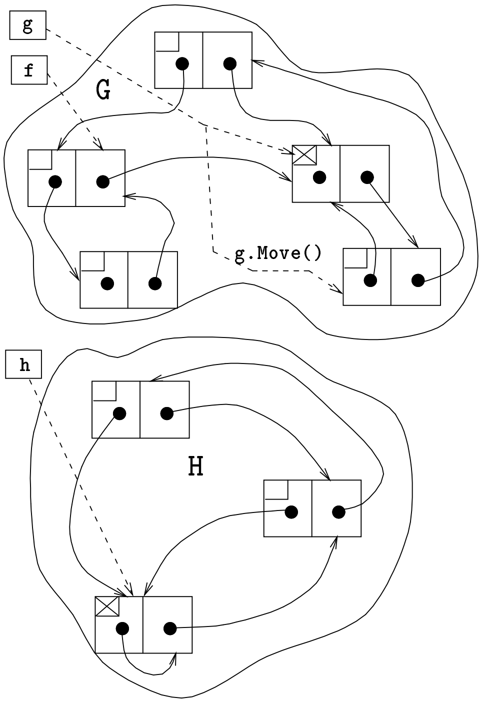


Figure 22: Modifications of the inheritance hierarchy. `Root` is the root of the inheritance tree (`Object` in Java). Triangles represent subtrees. There is an arrow from class C_1 to C_2 if C_2 inherits from C_1. The two basic operations, class factoring and class insertion, are shown in (a) and (b), respectively. After factoring class C, all references to C in the program should be replaced by C_1. Factoring and insertion are normally combined. This is done in (d), where the original class C is first split into C_1 and C_2, and then an extra child is created for C_1.




```
Node g, h;
method P(···,Node f) {
   /* 1 */  g = g.Move();
            h = h.Move();
   /* 2 */  h = h.Insert(new Node);
                     ⋮
   /* 3 */  x.R(···, f.Move());
                     ⋮
   /* 4 */  if (f == g)^? ···
   /* 5 */  if (g == h)^F ···
                     ⋮
   /* 6 */  f.Token=False;
            g.Token=True;
   /* 7 */  if (f.Token)^? ···
                     ⋮
   /* 8 */  f.Token=True;
            h.Token=False;
   /* 9 */  if (f.Token)^T ···
}
```

Figure 23: Opaque predicates constructed from objects and aliases. We construct a dynamic structure made from `Nodes`. Each `Node` has a boolean field `Token` and two pointer fields (represented by black dots) which can point to other nodes. The structure is designed to consist of two connected components, `G` and `H`. There are two global pointers, `g` and `h`, pointing into `G` and `H`, respectively.

```
{
   [S1];
   [S2];
   [S3];
}
```

$\overset{(a)}{\Longrightarrow}$

```
{
   if (P^?)
      [S1]
   else
      S1';
   if (Q^T)
      [S2]
   else
      S2^bug;
   if (R^F)
      S3^bug;
   [S3];
}
```

$\overset{(b)}{\Longrightarrow}$

```
{
   if (P^?)
      [S1]
   else
      S1';
   if (True)
      [S2]
   else
      S2^bug;
   if (False)
      S3^bug;
   [S3];
}
```

$\overset{(c)}{\Longrightarrow}$

```
{
   [S1];
   if (P^?)
      ;
   else
      ;
   if (True)
      [S2]
   else
      S2^bug;
   if (False)
      S3^bug;
   [S3];
}
```

$\overset{(d)}{\Longrightarrow}$

```
{
   [S1];
   [S2];
   [S3];
}
```

Figure 25: Obfuscation vs. deobfuscation. (a) shows an original program consisting of three statements S_{1-3} being obfuscated. The real program statements have been boxed for clarity. The unboxed code represents the bogus "program-within-the-program". In (b) a deobfuscator identifies "constant" opaque predicates (i.e. predicates that always evaluate to the same result) and replaces them with their computed value. In (c) the obfuscator determines that statements S_1 and S_1' in fact are identical, and hoists the common code from the conditional. In (d) the deobfuscator applies some final simplifications, and returns the program to its original form.

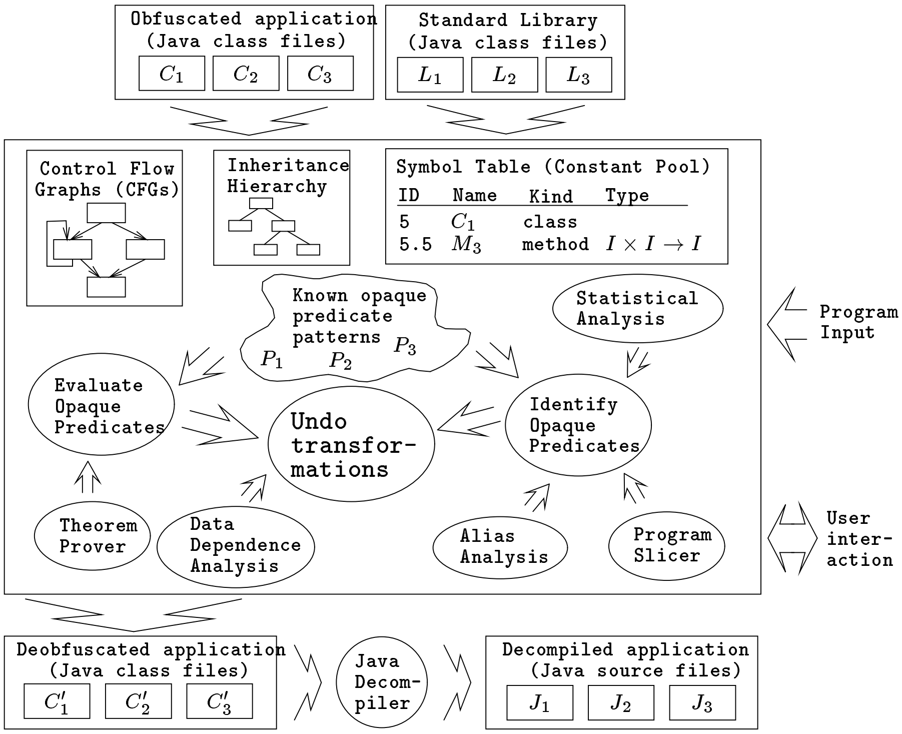


Figure 26: Architecture of a Java deobfuscation tool. The main input to the tool is an application made up of a set of obfuscated Java class files. The reverse engineer may also provide files of input data to allow statistical execution information to be gathered. The tool is likely to require extensive user interaction. Most theorem provers, for example, need guidance to find profitable proof strategies. The output of the tool is a set of deobfuscated class files which can be converted to Java source by a decompiler.