Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs

Christian Collberg   Clark Thomborson   Douglas Low

Department of Computer Science
The University of Auckland
Private Bag 92019
Auckland, New Zealand.
Phone: +64-9-373-7599
{*collberg,cthombor,dlow001*}*@cs.auckland.ac.nz*

Abstract

It has become common to distribute software in forms that are isomorphic to 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 describe the design of a Java *code obfuscator*, a tool which – through the application of code transformations – converts a Java program into an equivalent one that is more difficult to reverse engineer.

We describe a number of transformations which obfuscate control-flow. Transformations are evaluated with respect to *potency* (To what degree is a human reader confused?), *resilience* (How well are automatic *deobfuscation* attacks resisted?), *cost* (How much time/space overhead is added?), and *stealth* (How well does obfuscated code blend in with the original code?).

The resilience of many control-altering transformations rely on the resilience of *opaque predicates*. These are boolean valued expressions whose values are known to the obfuscator but difficult to determine for an automatic deobfuscator. We show how to construct resilient, cheap, and stealthy opaque predicates based on the intractability of certain static analysis problems such as alias analysis.

1 Introduction

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

This can be seen as a 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 gain access to the application, analyze it, and convert it into a form that is easy to read and understand.

This is a problem that has recently received renewed attention. The reason is that it is becoming more common to distribute software in architecture-neutral formats, (such as Java bytecode [10] and ANDF [16]), and because of the emergence of reverse engineering tools such as decompilers [5, 21] and program slicers [24].

1.1 Means of Software Protection

Alice can protect her code from Bob's attack using either *legal* [23] or *technical* [9] protection. Economic realities often make it difficult for a small company like Alice's to enforce the law against a larger and more powerful competitor [17]. A more attractive solution is for Alice to protect her code by making reverse engineering so technically difficult that it becomes at the very least economically inviable.

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, paying a small amount of electronic money every time. Bob will never gain physical access to the application and will be unable to reverse engineer it. Because of limits on network capacity the application will perform much worse than if it had run locally.

Alternatively, Alice could protect her code through *encryption* [14, 27]. This only works if the entire decryption/execution process takes place in hardware. 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.

Alice could forgo architecture neutral formats altogether. When downloading the application, the user's site would identify its architecture, and the corresponding native code version of the application (perhaps digitally signed by Alice to assure authenticity and harmlessness) would be transmitted. Only having access to the native code will make Bob's task more difficult, although not impossible [5].

1.2 Code Obfuscation

The final approach, and the one we will advocate in this paper, is *code obfuscation* (Figure 1). 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.

Unlike server-side execution, code obfuscation can never completely protect an application from malicious reverse en-

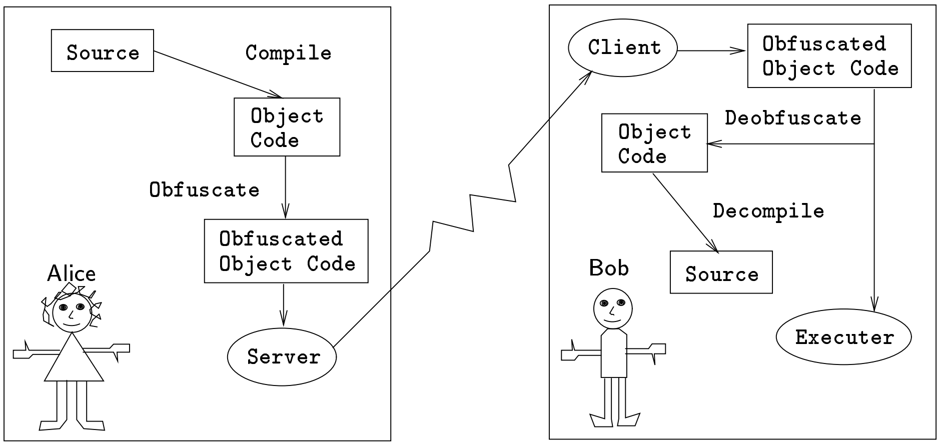


Figure 1: Software protection through obfuscation.

gineering 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, (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 and decryption.

The remainder of the paper is structured as follows. In Section 2 we give a brief overview of the design of a code obfuscator for Java, which is currently under construction. Section 3 describes the criteria used to evaluate different types of obfuscating transformations. Sections 4 and 5 present a catalogue of obfuscating transformations. Section 6 discusses deobfuscation. Section 7 gives implementation details and considers the cost of obfuscation. Finally, Section 8 summarizes our results.

2 The Design of a Java Obfuscator

Figure 2 outlines the design of a Java obfuscation tool currently under development. The input to the tool is

1. a Java application,
2. the required level of obfuscation (the *potency*),
3. the maximum execution time/space penalty that the obfuscator is allowed to add to the application (the *cost*), and
4. profiling data, as generated by Java profiling tools.

The obfuscator reads and parses the Java class files along with any referenced library classes. Symbol tables and inheritance graphs are built from the class files' constant pools, and control-flow graphs are constructed from method bodies.

The obfuscator contains a large pool of code transformations which are applied repeatedly to the application until the required obfuscation potency has been achieved or the maximum cost has been exceeded. 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. The output of the tool is a new application which is functionally equivalent to the original one.

3 Obfuscating Transformations

Existing obfuscation tools (such as Crema [26]) are based on the assumption that the original and obfuscated program must have identical behavior. In the present paper we assume that under certain circumstances it will be possible to relax this constraint. In particular, we allow most of our obfuscating transformations to make the target program slower or larger than the original. In special cases we may 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* of the two programs should be identical. Formally:

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.

□

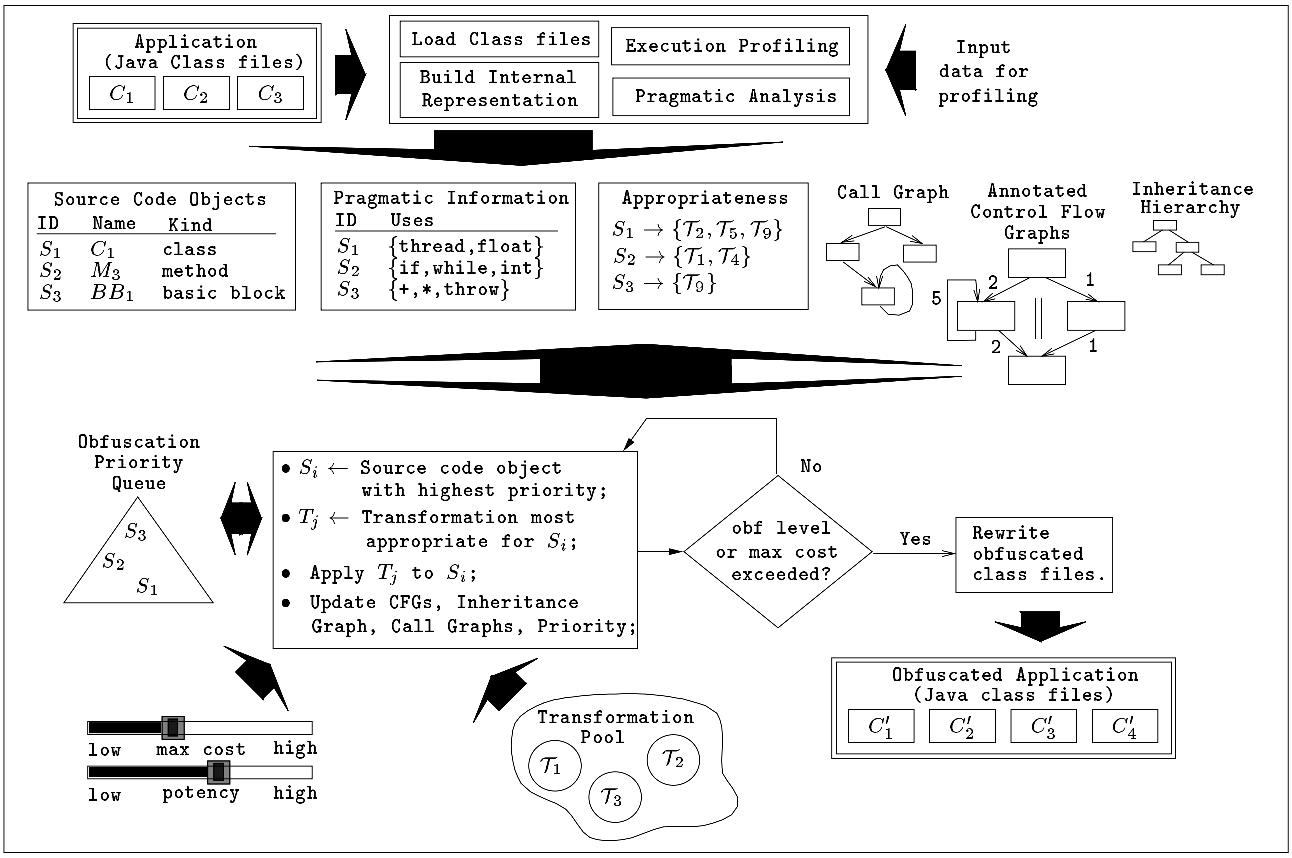


Figure 2: Architecture of a Java obfuscator. The main input to the tool is a set of Java class files and the obfuscation level required by the user. The user also provides files of *profiling data*. The obfuscator reads all referenced class files (including library files) and builds various internal data structures. Symbol tables and inheritance graphs store information on classes and methods, methods are decompiled into control flow graphs, etc. The control flow graphs are annotated with execution counts. *Pagmatic information* expresses the kinds of language constructs a class/method contains. See Section 7.3 for further details.

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.

Obfuscating transformations that cannot be deobfuscated using static analysis techniques may also prevent some code optimizations from being applied to a program. An example is the introduction of spurious aliases (Section 5.1). Thus code optimization would normally be applied before obfuscations.

3.1 Classifying Transformations

The main dividing line between different classes of obfuscation techniques is the kind of *information* it targets. Some simple transformations – typical of current Java obfuscators such as Crema [26] – target the lexical structure of the application, such as source code formatting, names of variables, etc. The more sophisticated transformations that we are interested in target either the data structures used by the application or its flow of control.

3.2 Obfuscation Quality

The quality of an obfuscating transformation is evaluated according to four criteria: how much obscurity it adds to the program (the *potency*), how difficult it is to break for an automatic deobfuscator (the *resilience*), how well the obfuscated code blends in with the rest of the program (the *stealth*), and how much computational overhead it adds to the obfuscated application (the *cost*).

3.2.1 Measures of Potency

What does it mean for a program P' to be more *obscure* (or *complex* or *unreadable*) than a program P? To answer this question we can examine the complexity formulas found in the *Software Complexity Metrics* literature.

Of particular interest to us are the McCabe [18] and Harrison [11] metrics. McCabe states that the complexity of a program grows with the number of predicates it contains. According to Harrison, the complexity is also proportional to the nesting level of conditional and looping constructs.

Other metrics express that the complexity of a program increases with the the complexity of its data structures [19], the number of inter-basic block variable dependencies [20], the number of formal parameters [13], and the depth of its inheritance tree [4].

We say that a transformation which increases any of these metrics is a *highly potent* obfuscating transformation.

3.2.2 Measures of Resilience

At first glance it would seem to be trivial to construct potent obfuscating transformations. To increase the McCabe metric, for example, we simply add some arbitrary if-statements to P:

```
main() {
    S1;
    S2;
}
```

$$\xRightarrow{\mathcal{T}}$$

```
main() {
    if (5==2) S1;
    S1;
    if (2>1) 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 effort 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 $\mathcal{T}$.

Some highly resilient transformations are *one-way*, in the sense that they can never be undone. This is typically because they *remove* information (such as formatting, variable names) from the program. Other transformations *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.

3.2.3 Measures of Stealth

While a resilient transformation may not be susceptible to attacks by automatic deobfuscators, it may still be susceptible to attacks by humans. Particularly, if a transformation introduces new code that differs wildly from what is in the original program it will be easy to spot for a reverse engineer. A predicate such as the one below may be very resilient to automatic attacks, but will stick out "like a sore thumb" in most programs:

$$\texttt{if IsPrime}(\overbrace{837523474\ldots3853845347527}^{\text{512-bit integer}})\ \texttt{then}\ \cdots$$

In other words, it is essential that obfuscated code resemble the original code as much as possible. Such transformations are *stealthy*.

Obviously, stealth is a highly context-sensitive metric. A transformation may introduce code which is stealthy in one program but extremely unstealthy in another one.

3.2.4 Measures of Execution Cost

The *cost* of a transformation is the execution time/space penalty which a transformation incurs on an obfuscated application.

Some trivial transformations (scrambling of variable names, removal of formatting) are *free*, i.e. they incur no run-time cost. Many of the transformations presented in this paper will incur a varying amount of overhead.

Like *stealth*, *cost* is a context-sensitive metric. For example, a 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.

4 Control Transformations

In this section we will present a few obfuscating control-flow transformations. For such transformations, 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.

Obfuscating control-flow transformations fall into three categories: (1) hide the real control-flow behind irrelevant statements that do not contribute to the actual computations, (2) introduce code sequences at the object code level for which there exist no corresponding high-level language constructs, or (3) remove real control-flow abstractions or introduce spurious ones.

4.1 Opaque Predicates

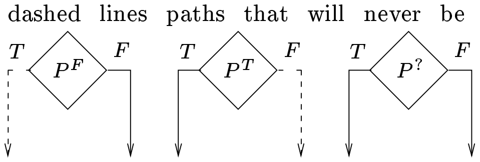
The real challenge when designing control-altering transformations is to make them not only cheap, but also resistant to attack from deobfuscators. To achieve this, many transformations rely on the existence of *opaque variables* and *opaque predicates*. Informally, a variable V (or predicate P) is opaque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

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.

DEFINITION 2 (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 may sometimes evaluate to `True` and sometimes to `False`. □

The different types of opaque predicates are illustrated here (solid lines indicate paths that may sometimes be taken, dashed lines paths that will never be taken):



See Figure 3 for some simple examples and Section 5 for a thorough discussion.

```
{  int v, a=5; b=6;
   v^{=11} = a + b; /* v is 11 here. */
   if (b > 5)^T  ···
   if (random(1,5) < 0)^F  ···
   if (···)  ···
          ⋮ (a and b are unchanged)
   if (b < 7)^T  a++;
   v^{=36} = (a > 5)?v=b*b:v=b /* v is 36 here. */
}
```

Figure 3: Examples of trivial opaque constructs of low resilience. We assume `random`(a, b) is a standard library function (whose semantics is known to the obfuscator as well as deobfuscator) that returns an integer in the range $a \cdots b$. A deobfuscator can crack these and similar opaque constructs using simple intra-procedural static analyses.

4.2 Insert Dead or Irrelevant Code

The McCabe and Harrison software 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 4. In Figure 4(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 4(b) we again break S into two. We then proceed to create two *different* obfuscated versions S^a and S^b of the second half by applying different sets of obfuscating transformations to the second half of S. 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 4(c) is similar to Figure 4(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.

4.3 Extend Loop Conditions

Figure 5 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 5(d), for example, will always evaluate to `True` since $x^2(x+1)^2 \equiv 0 \pmod 4$.

4.4 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. For example, while the Java *virtual machine code* can express arbitrary flow graphs, the Java language can only express *reducible* flow graphs. *Language-breaking* transformations take advantage of this to introduce virtual machine instruction sequences which have no direct correspondence with any source language construct.

Figure 6(a) illustrates a transformation which converts a reducible flow graph to a non-reducible one, by turning a structured loop into a loop with multiple headers [1]. A bogus jump (protected by an opaque predicate P^F) is added to the code to make it appear that there is a jump into the middle of a loop.

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 6(b). If a deobfuscator blindly removes P^F, the resulting code will be incorrect.

5 Manufacturing Opaque Constructs

Opaque predicates are the major building blocks in the design of obfuscating control transformations. In fact, the quality of most control transformations is directly dependent on the quality of such predicates.

In Section 4.1 we gave examples of simple opaque predicates with low resilience. These opaque predicates could be broken (an automatic deobfuscator could determine their value) using simple global static analysis. Obviously, we generally require a much higher resistance to attack.

Equally important is the *cost* and *stealth* of opaque predicates. An introduced predicate that differs wildly from what is in the original program will be unacceptable, since it will be easy to detect for a reverse engineer. Similarly, a predicate is unacceptable if it introduces excessive computational overhead.

A study of some random Java programs reveal that most predicates are extremely simple. Common patterns include ⌜`p==null`⌝, ⌜`p==q`⌝, ⌜`n <=` *IntLit*⌝, where `p,q` are pointers and `n` is an integer. We must be able to generate cheap and inconspicuous opaque predicates that resemble these patterns.

Since we expect most deobfuscators to employ various static analysis techniques (such as data-flow analysis [1] and slicing [24]) it seems natural to base the construction of opaque predicates on problems which these techniques cannot handle well. In particular, precise static analysis of pointer-based structures and parallel regions is known to be intractable. Next, we will show how to construct opaque predicates based on this insight.

5.1 Opaque Constructs Using Objects and Aliases

Inter-procedural static analysis is significantly complicated whenever there is a possibility of aliasing. In fact, precise, flow-sensitive alias analysis is undecidable in languages with dynamic allocation, loops, and if-statements [22].

In this section we will exploit the difficulty of alias analysis to construct opaque predicates which are cheap, stealthy (in pointer-rich languages like Java), and resilient to automatic deobfuscation attacks.

5.1.1 Alias and Shape Analysis

While in the general case alias analysis may be undecidable, there exist many conservative algorithms that perform well for actual programs written by humans.

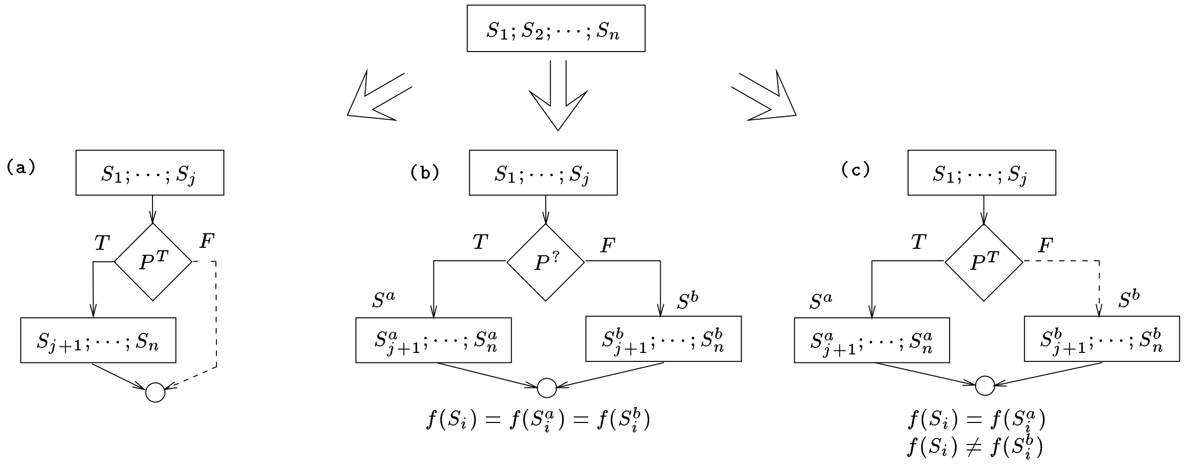


Figure 4: The Branch Insertion transformation.

Of particular interest to us are techniques developed for *shape/heap analysis*. The goal of these analyses is to determine what kind of structure a pointer p points to (a tree, a DAG, or a cyclic graph), and if two pointers must/may refer to the same heap object at some particular program location.

All practical heap analysis algorithms are by necessity imprecise, but different algorithms will perform more or less well for particular types of dynamic structures. Ghiya's [8] algorithm provides accurate information for programs that build simple data structures (trees and arrays of trees), but isn't powerful enough to handle programs that make major structural changes to the structure. Chase's [3] algorithm also has problems with destructive updates. In particular, while it handles *list append*, it fails to analyze an in-place list reversal program. Hendren [12] cannot handle cyclic structures, and many other algorithms only handle recursive structures that are no more than k levels deep.

The most powerful algorithm to date seems to be Deutsch [7], but the implementation is complex (8000 lines of ML) and slow even for small programs (30 seconds to analyze a 50 line program).

Our goal will be to attempt to exploit the general difficulty of the alias analysis problem and the shortcomings of current conservative algorithms to manufacture cheap and resilient opaque predicates. The basic technique we will use is this:

1. Add to the obfuscated application code which builds a set of complex dynamic structures $S_1, S_2, \cdots$.
2. Keep a set of pointers $p_1, p_2, \cdots$ into these structures.
3. The introduced code should occasionally update the structures (modifying pointers, adding nodes, splitting and merging structures, etc), but must maintain certain invariants, such as "p_1 will never refer to the same heap location as p_3", "there may be a path from p_1 to p_2", etc.
4. Use these invariants to manufacture opaque predicates when needed.

This method is very attractive for three reasons:

1. the introduced code will closely resemble the code found in many real, pointer-rich, Java applications (i.e. the bogus code will be *stealthy*),
2. it is easy to construct 'destructive update' operations which current heap analysis algorithms will fail to analyze (i.e. the bogus code will be *resilient*), and
3. it is easy to construct invariants which can be tested for in constant time. (i.e. the bogus code will be *cheap*).

5.1.2 A Simple Example

Consider the obfuscated method `P` in Figure 7. 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 = Move`$_2$`(g)`⌝ will nondeterministically update `g` to point somewhere else within `G`. The statement ⌜`h = Insert`$_{2,1}$`(h)`⌝ 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`.

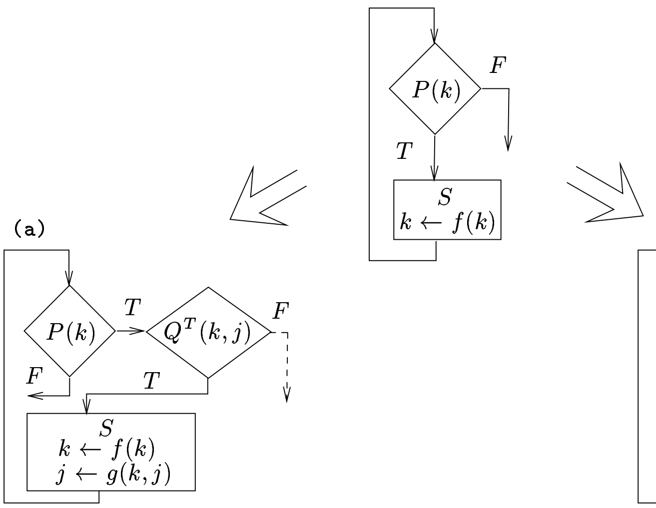
This set-up allows us to construct opaque predicates like those of statements 4 and 5 of Figure 7. The predicate `f==g` may be either `True` or `False` since `f` and `g` move around within the same component. Conversely, `g==h` must be false since `g` and `h` refer to nodes within different components.

Statements 6–9 in Figure 7 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.

Statement 10 splits `G` into two components `G'` and `G''`, with `f` and `g` pointing into different structures. As a result, `(f==g)` must be false in statement 11.

5.1.3 A Graph ADT

To make this more concrete, we will design a Java abstract data type `Graph` that can be included with an obfuscated application. We will use calls to the `Graph` operations to




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

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

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

(c) (d)

Figure 5: The Loop Condition Insertion transformation.

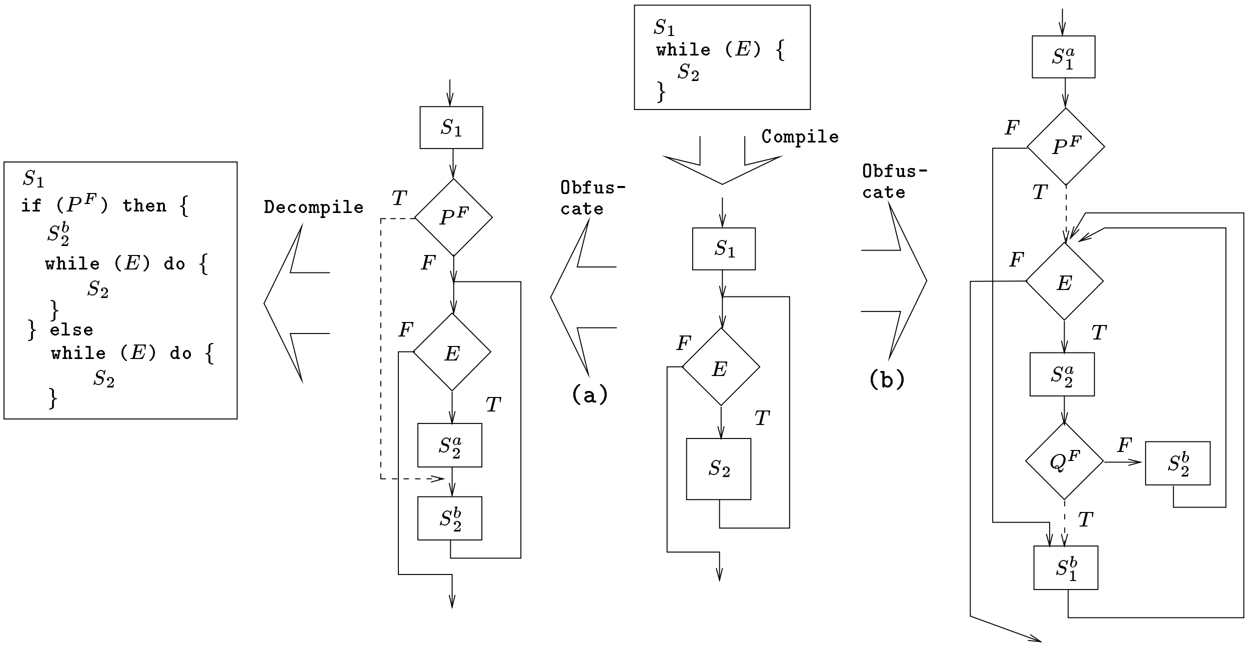
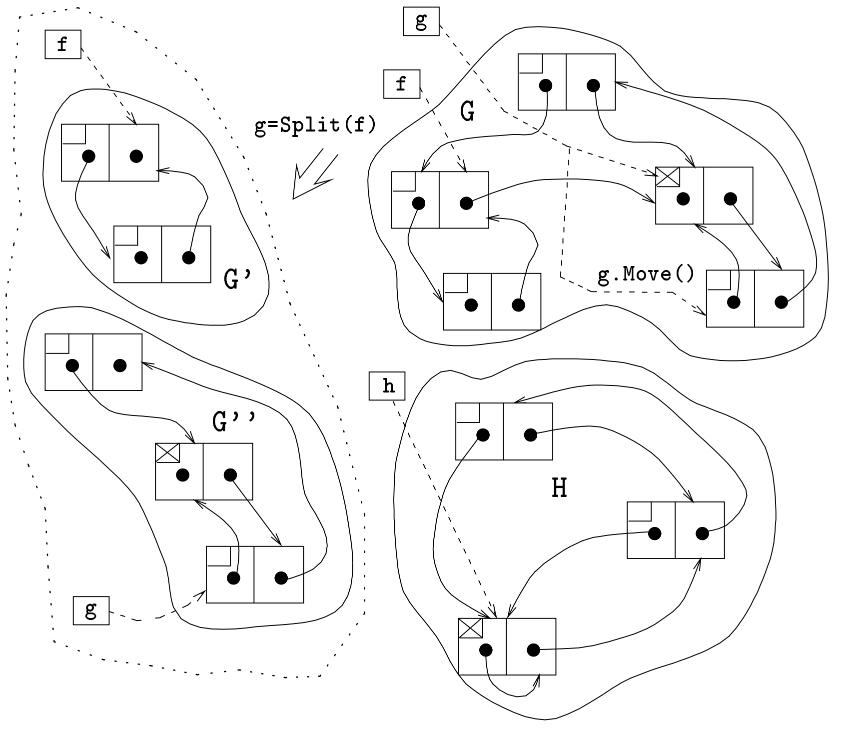


Figure 6: The Reducible to Non-Reducible Flow Graph transformation. In (a) we split the loop body S_2 into two parts (S_2^a and S_2^b), and insert a bogus jump to the beginning of S_2^b. In (b) we also break S_1 into two parts, S_1^a and S_1^b. S_1^b is moved into the loop and an opaque predicate P^T 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.




```
Node g, h;
method P(···,Node f) {
   /* 1 */  g = Move_2(g);
            h = Move_3(h);
   /* 2 */  h = Insert_{2,1}(h);
                  ⋮
   /* 3 */  x.R(···, Move_2(f));
                  ⋮
   /* 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 ···
                  ⋮
   /* 10 */ g = Split_2(f);
   /* 11 */ if (f == g)^F ···
}
```

Figure 7: Opaque predicates constructed from objects and aliases. Only introduced (bogus) code is shown. We construct a dynamic structure made from `Node`s. Each `Node` has a boolean field `Token` and two pointer fields (represented by black dots) which can point to other nodes. The generated 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.

---

manufacture opaque predicates that are *cheap*, *resilient*, as well as *stealthy* within a pointer environment.

Obviously, we must prevent a deobfuscator from identifying the ADT by simple pattern matching. There are three obvious techniques available to an ambitious obfuscator:

1. The obfuscator should keep a large library of variants of the `Graph` ADT that it could randomly select between. In fact, several variants could be included with (and used in different parts of) the same application.

2. Invocations of the `Graph` primitives should be subject to the same obfuscations as the user code, including inlining, outlining, and name mangling [6].

3. Rather than including the `Graph` ADT as a stand-alone class, it could be merged with the most similar user-defined class. This way, the bogus `Graph` nodes created by the obfuscated application would be indistinguishable from real objects created by the original application.

For clarity [sic], our examples will avoid any such tricks.

Consider the example `Graph` ADT in Figure 8. It contains operations for adding new nodes to a graph (`Node` and $\texttt{addNode}_i$), traversing a graph ($\texttt{selectNode}_i$), and splitting a graph into components (`reachableNodes` and `splitGraph`). A more complete implementation might contain other operations as well such as merging graphs, inverting graphs (changing the direction of pointers), and testing for various graph properties (connectivity, acyclicity, reachability, isomorphism, etc.).

The `Graph` ADT operations can be combined to create any number of code patterns that can be inserted at various points in the application. Table 1 shows four such patterns.

In Table 1(a) `Insert` inserts a new node at an arbitrary place in the graph.

In Table 1(b) `Move` makes `P` point to an arbitrary node within the graph reachable from `P`. Note that the node `P` pointed to before the move is now unreachable and will be reclaimed by the garbage collector.

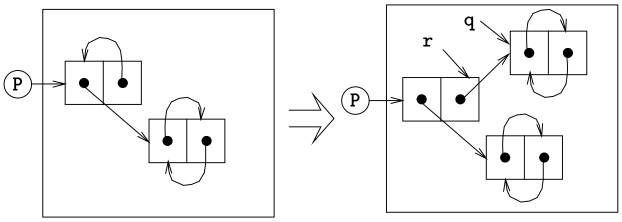
The graphs built by the patterns in (a) and (b) will essentially be tree-shaped. Thus if `P` and `Q` point to nodes in a connected graph, then after one of these operations is performed, `P` and `Q` can still possibly refer to the same node.

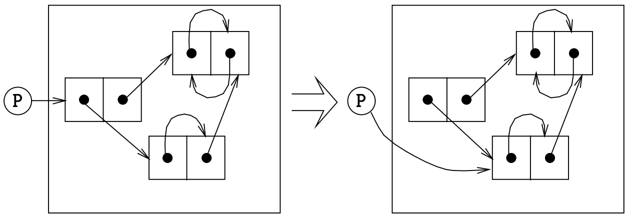
In Table 1(c) the `Link` pattern ensures that we build general graphs by adding an edge from some leaf `b` to an arbitrary node `a`. The requirement that we only add edges from leaves ensures that the graph will remain connected.

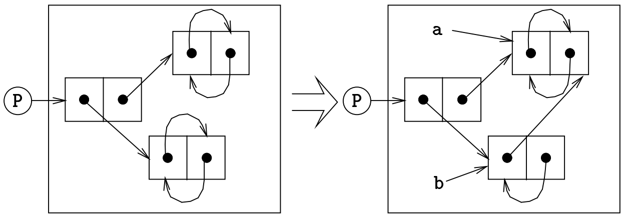
In Table 1(d) the `Split` pattern breaks up the graph pointed to by `P` into two unrelated components. After the split we know that `P` and `Q` point into different components, and regardless of which operations are performed on these components `P` and `Q` will never alias one another.

Figure 9 shows how these patterns can be used to construct opaque predicates in a real program. Further transformations, such as inlining, can be applied to disguise the inserted code.

#	Code Pattern	Example

(a)	`Node Insert`$_{i,j}$`(Node P) {`
	`  if (P==null) return new Node();`
	`  else {`
	`    r = P.selectNode`$_i$`();`
	`    return r.addNode`$_j$`();`
	`  };`
	`}`
		

(b)	`Node Move`$_i$`(Node P) {`
	`  return P.selectNode`$_i$`();`
	`}`
		

(c)	`void Link`$_{i,j}$`(Node P) {`
	`  a = P.selectNode`$_i$`();`
	`  b = P.selectNode`$_j$`();`
	`  if (b.car == b) b.car=a;`
	`}`
		

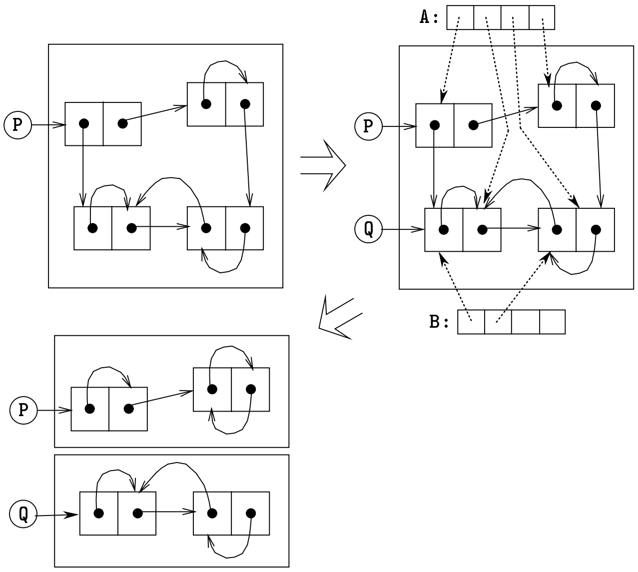
(d)	`Node Split`$_i$`(Node P) {`
	`  Q = P.selectNode`$_i$`();`
	`  Set A = P.reachableNodes();`
	`  Set B = Q.reachableNodes();`
	
	`  Set D = A.setDifference(B);`
	`  P.splitGraph(D,B);`
	`  return Q;`
	`}`
		

Table 1: Code patterns. These procedures (and others like these) are inserted by the obfuscator. The idea is to maintain a number of complex dynamic data structures, and pointers into these structures, which will allow the obfuscator to create resilient opaque predicates. The code patterns defined in this table use the primitives in the `Node` class defined in Figure 8. `Insert(P)` adds a new node to a component, `Move(P)` returns a node reachable from `P`, `Link(P)` adds a link between two nodes reachable from `P`, and `Split(P)` splits a component into two unrelated components. After a `Q=Split(P)` transformation, pointers `P` and `Q` can never alias each other.

```
public class Node {
  public Node car, cdr;

  public Node() {
    this.car = this.cdr = this; }

  /* addNode_i is a family of functions
    which insert a new node after 'this'. */
  Node addNode_1() {
    Node p = new Node(); p.car = this.car;
    return this.car = p; }
  Node addNode_2() {
    Node p = new Node; p.cdr = this.car;
    return this.car = p; }

  /* selectNode_i is a family of functions
    which return a reference to a node
    reachable from 'this'. */
  Node selectNode_1() { return this; }
  Node selectNode_2() { return this.car; }
  Node selectNode_3() { return this.car.cdr; }

  public Node selectNode_4(int n) {
    return (n <= 0)?this:
      this.car.selectNode_4b(n-1);
  }
  public Node selectNode_4b(int n) {
    return (n <= 0)?this:
      this.cdr.selectNode_4(n-1);
  }
```

```
  /* Return the set of nodes reachable
    from 'this'. */
  public Set reachableNodes()
    { return reachableNodes(new Set()); }
  Set reachableNodes(Set reached) {
    if (!reached.member(this)) {
      reached.insert(this);
      this.car.reachableNodes(reached);
      this.cdr.reachableNodes(reached);
    }
    return reached;
  }
  /* A and B are sets of graph nodes.
    Remove any references between nodes
    in A and B. */
  public void splitGraph(Set A, Set B) {
      ...
  }
  private void splitGraph(Set R, Set A, Set B) {
    if (!R.member(this)) {
      R.insert(this);
      this.car.splitGraph(R, A, B);
      this.cdr.splitGraph(R, A, B);

      if (this.diffComp(this.car, A, B))
        this.car = this;
      if (this.diffComp(this.cdr, A, B))
        this.cdr = this;
  }}

  /* Returns True if the current node and */
    node b are in different components */
  private boolean diffComp(Node b, Set A, Set B) {
    return (A.member(this) && B.member(b)) ||
      (B.member(this) && A.member(b));
  }}
}
```

Figure 8: A simple graph ADT to be used for the manufacturing of opaque predicates. Class `Set` (not shown) with operations `insert` and `member` implements sets of objects. It could, for example, be implemented by the Java `HashTable` library class. In this particular implementation of `Node` we make sure there are no `null` pointers by making terminal nodes point to themselves. This simplifies the implementation of the $\texttt{selectNode}_i$ family of functions. The primitives defined in this figure are used by the code patterns in Table 1.

5.2 Opaque Constructs Using Concurrency

Parallel programs are more difficult to analyze statically than their sequential counterparts. The reason is their *interleaving* semantics: n statements in a parallel region

$$\textbf{PAR } S_1; S_2; \cdots; S_n; \textbf{ ENDPAR}$$

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 traffic, 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 highly resilient opaque predicates. The basic idea is very similar to the one used in Section 5.1: 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 graphs created in Figure 7. 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 10 we illustrate these ideas with a much simpler

```
static void RayTrace (Vector scene, ViewDes v) {
  Node p = Insert1,1(null); Insert1,2(p);
  Node q = Insert1,1(null);
  for (int y = 0; y < v.height; y++) {
    if (y >= h - 10)
      Insert4,2(p, (int)(y * 1.5));
    if (y == h - 10)
      q = Split1(p);
    for (int x = 0; x < v.width; x++) {
      if ((y <= v.height - 10) &&
        (Move4(p,x) == Move4b(q,x))^F
          break;
      Ray theRay = v.pixelRay(y, x);
      SceneObject obj = hitObject(theRay, scene);
      if (obj != null) {
        Colour color = obj.surface.color(
          obj.hitPoint, obj.normal, v.eyePoint);
        Graphics.drawPoint(color, x, y);
}}}}
```

Figure 9: An example showing bogus code (in *italics*) inserted into a small Java routine. The code is constructed so that `p` and `q` will never point into the same dynamic structure.

```
class S extends Thread {
  public void run() {
    while (true) {
      int R = (int) (Math.random() * 65536);
      M.X = R*R; Thread.sleep(3);
}}
class T extends Thread {
  public void run() {
    while (true) {
      int R = (int) (Math.random() * 9300);
      M.Y = 7*R*R; Thread.sleep(2);
      M.X *= M.X; Thread.sleep(5);
}}}
public class M {
  public static int X, Y;
  public static void main(String argv[]) {
    S s = new S(); s.start();
    T t = new T(); t.start();
    if ((Y - 1) == X)^F  <= [1]
      System.out.println("Bogus code!");
    s.stop(); t.stop();
}}
```

Figure 10: In this example, the predicate at point $\boxed{1}$ will always evaluate to `False`. Two threads `s` and `t` occasionally wake up to update global variables `M.X` and `M.Y` with new random values. Notice that `s` and `t` are involved in a data-race on `M.X`, but that this does not matter as long as assignments are atomic. Regardless of whether `s` or `t` wins the race, `M.X` will hold the square of a number.

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, $7y^2 - 1 \neq x^2$.

For inherently sequential applications opaque predicates based on introduced bogus threads will be highly unstealthy. In such cases we can instead make use of Java's *finalizers*. A finalizer is a method that will be invoked on an object at some (unspecified) time after it has become unreachable and before it is garbage collected. Figure 11 gives an example of how opaque predicates can be constructed by combining finalizers with the `Graph` ADT of Section 5.1.3.

6 Deobfuscation

To be able to evaluate the resilience of obfuscating transformations, it is necessary to consider what tools are available to an automatic deobuscator. So far we have assumed that these tools mainly analyze the obfuscated program statically. For example, the simple opaque predicates in Figure 3 can be cracked by a global data flow analysis, the predicate $(7y^2 - 1 = x^2)^F$ can be cracked by a theorem prover, and static slicing techniques can be used to bring together logically related pieces of code which the obfuscator has dispersed over the program.

Deobfuscators can also use *dynamic* analysis. An obfuscated program can, for example, be instrumented to analyze the outcome of all predicates. Any predicate that always returns `True` (`False`) over a large number of test runs may warrant further study, since it may turn out to be an opaque P^T (P^F) predicate.

One possible counter-measure against dynamic analysis is to design opaque predicates in such a way that several predicates have to be cracked at the same time. The obfuscator can, for example, introduce opaque predicates with side-effects. If, in the example below, 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. (This particular example does not work in Java, since Java does not detect integer overflow.)

$$\left\{ \begin{array}{l} S_1; \\ \quad \ldots \\ S_2; \end{array} \right\} \quad \overset{\mathcal{T}}{\Longrightarrow} \quad \begin{array}{l} \texttt{int k=0;} \\ \texttt{bool } Q_1\texttt{(x) \{} \\ \quad \texttt{k+=}2^{31}\texttt{; return } (P_1^T)\texttt{\}} \\ \texttt{bool } Q_2\texttt{(x) \{} \\ \quad \texttt{k-=}2^{31}\texttt{; return } (P_2^T)\texttt{\}} \\ \\ \texttt{\{if } (Q_1\texttt{(j)}^T)\ S_1; \\ \quad \ldots \\ \texttt{if } (Q_2\texttt{(k)}^T)\ S_2;\texttt{\}} \end{array}$$

7 Discussion

Generating opaque predicates is an important task for an obfuscator. There are, however, many other practical problems that must be resolved before building a usable obfuscator. We will discuss some of these issues next.

7.1 The Power of Obfuscation

The control flow transformations presented here are only a few of a large catalogue of obfuscations which target every aspect of a program. Some of these are closely related to code optimizations such as inlining, outlining, cloning, parallelization, and various loop optimizations [2]. Other important transformations target the data structures created by the application or the static structure of the program, such as the module structure and inheritance rela-

```
class A {
  private Node p;
  public A(Node p, Node q) {
    this.p = p;
    q = Split_2(p); }
  public void finalize() { Insert_{2,1}(p); }
}

class B {
  private Node q; private int i;
  public B(Node q, int i) {
    this.q = q; this.i = i; }
  public void finalize() {
    Insert_{2,2}(q); Link_{2,1}(q); }
}

public class Main {
  public static void main(String argv[]) {
    Node p = Insert_{2,2}(null); Insert_{2,1}(p);
    Node q = Insert_{1,1}(null);
    A a = new A(p, q);
    B b = new B(q, 5);
        ⋮
    a = b = null; ⇐ [1]
        ⋮
    p = Move_2(p); q = Move_3(q);
    if (p == q)^F ··· ⇐ [2]
}}
```

Figure 11: In this example we combine Java's *finalizers* with the graph-manipulation operations of Section 5.1.3. The finalizers may run at any time (or not run at all) after the objects `a` and `b` have been released at point $\boxed{1}$. Regardless, pointers `p` and `q` will point to different structures at point $\boxed{2}$.

tionships [6]. The extra complexity that an obfuscator adds to a program will depend on the complex interaction between all the different types of transformations which have been applied to it.

7.2 The Cost of Obfuscation

What effect will obfuscation have on the execution behavior of an application? There are three main issues:

Code bloat Our obfuscator obscures a program primarily by hiding the real control flow behind introduced bogus control flow. As a result, the obfuscated program will be larger than the original.

Data bloat Opaque predicates based on alias analysis rely on the obfuscated program building complex dynamic data structures at runtime. Hence, the obfuscated program will generate more dynamic data than the original.

Cycle bloat Every introduced instruction (that is not part of a dead code section) will be executed by the interpreter. Consequently, the obfuscated program will execute more instruction cycles than the original.

Out of these three problems, cycle bloat is the least serious. Most introduced instructions are in dead code sections guarded by opaque predicates. These predicates will often consist of simple pointer or integer comparisons that will contribute little to the total runtime of the application.

Code bloat can have detrimental effect on mobile programs since increased code size will result in increased downloading time. Once down-loaded, the obfuscated program may run slower due to deteriorated cache- and paging behavior.

The most serious problem is data bloat. First of all, more dynamic data means an increased workload for the garbage collector, and, again, higher cache miss rates. More seriously, an application that previously ran successfully on a particular memory configuration may, after obfuscation, not run at all since it now exhausts the available heap space.

7.3 Selecting Transformations

Figure 2 shows the overall design of our Java obfuscator which is currently under construction. It is designed to achieve maximal obfuscation potency and stealth and to minimize the space and time costs discussed in the previous section.

The obfuscator builds several internal data structures. An *appropriateness table* maps each *source code object* (ie. every class, method, basic block, etc. that may be obfuscated) to a set of transformations that would be stealthy, cheap, resilient, and potent for that particular object. To find stealthy transformations we simply compare the set of language constructs already used by the object (*pragmatic information*) to the constructs introduced by the transformation.

Not every part of a program contains trade secrets. Hence, different parts of the same program will need different levels of obfuscation. Therefore, each source code object is given an *obfuscation priority* describing its required level of protection. This can either be provided explicitly by the user, or it can be computed using some heuristic based on the static structure of the program. The source code objects are obfuscated in priority order. After a transformation has been applied to an object, its priority is decreased based on the potency and resilience of the transformation.

As seen in Figure 2, control flow graphs are annotated with execution counts, either statically estimated or provided through profiling. These are used to guide the selection of transformations and opaque predicates, so that frequently executed parts of the application are not obfuscated by very expensive transformations and new dynamic memory is not allocated in inner loops.

8 Conclusion

We have shown that it is possible to obfuscate the control flow of an application by inserting irrelevant conditionals and loops. The resilience of such obfuscations (the extent to which they will stand up to attack from automatic deobfuscators) depends on the resilience of the inserted predicate. The main contribution of this paper is the insight that it is possible to base the manufacturing of resilient predicates on the intractability of static analysis problems such as the analysis of aliasing, concurrency, and data dependence.

While all transformations described in this paper have been cast in terms of Java, it should be clear that most apply equally well to other languages. In fact, since our obfuscator targets Java class files it is already able to obfuscate programs written in a variety of languages. The reason, of course, is the existence of translators from many languages (including Ada and Scheme) into Java source or bytecode [25].

Acknowledgments: We would like to thank Todd Proebsting, Chris Fraser, Mark Burgess, and Buz Uzgalis for stimulating discussions.

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] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *ACM SIGPLAN Notices*, 25(6):296–310, June 1990.

[4] 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.

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

[6] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997. `http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborsonLow97a`.

[7] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN PLDI'94*, pages 230–241, Orlando (Florida, USA), June 1994. ACM. SIGPLAN Notices, 29(6).

[8] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *POPL'96*, pages 1–15, St. Petersburg Beach, Florida, 21–24 January 1996.

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

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

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

[12] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

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

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

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

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

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

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

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

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

[22] G. Ramalingam. The undecidability of aliasing. *ACM TOPLAS*, 16(5):1467–1471, September 1994.

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

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

[25] Robert Tolksdorf. Programming languages for the Java virtual machine, 1997. `http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html`.

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

[27] Uwe G. Wilhelm. Cryptographically protected objects. In *RenPar'9*, May 1997. `http://lsewww.epfl.ch/~wilhelm/CryPO.html`.