Breaking Abstractions and Unstructuring Data Structures

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

Abstract

To ensure platform independence, mobile programs are distributed in forms that are isomorphic to the original source code. Such codes are easy to decompile, and hence they increase the risk of malicious reverse engineering attacks.

Code obfuscation *is one of several techniques which has been proposed to alleviate this situation. An* obfuscator *is a tool which – through the application of code transformations – converts a program into an equivalent one that is more difficult to reverse engineer.*

In a previous paper [5] we have described the design of a control flow obfuscator for Java. In this paper we extend the design with transformations that obfuscate data structures and abstractions. In particular, we show how to obfuscate classes, arrays, procedural abstractions and built-in data types like strings, integers, and booleans.

1 Introduction

Mobile programs are distributed in architecture-neutral formats (such as Java bytecode [8]) that contain much of the same information as the original source code. While this achieves platform independence, it also makes programs easy to decompile and reverse engineer.

This is of particular concern to small software developers who can ill afford to protect their software secrets through *legal* [17] means against larger and more powerful competitors [12]. As an alternative, several forms of *technical* [1, 7] protection against theft of software secrets have been suggested:

Server-Side Execution The user connects to the software developer's site to run the program remotely, paying a small amount of electronic money every time. A software thief will never gain physical access to the application and will be unable to reverse engineer it. The downside is that the application will perform much worse than if it had run locally.

Native Code When down-loading the application, the user's site identifies its architecture, and the corresponding native code version of the application is transmitted. Digital signatures should be attached to the code to assure authenticity and harmlessness. Decompilation of the native code is still possible [3], but much more difficult, if, as is usual, symbol naming and type information is suppressed.

Encryption Encrypting [11, 21] the application will only protect against theft if the entire decryption/execution process takes place in hardware. If the code is executed in software by a virtual machine interpreter it will always be possible to intercept and decompile the decrypted code.

Obfuscation Before distributing the application, the software developer runs it through an automatic *obfuscator*. This tool transforms the program into one that is functionally identical to the original but which is more difficult to decompile and reverse engineer.

Unlike server-side execution and hardware-based encryption schemes, code obfuscation can never completely protect an application from malicious reverse engineering efforts. Rather, obfuscation should be seen as a cheap way of making reverse engineering so technically difficult that it becomes economically infeasible. To ensure this, the techniques employed by an obfuscator have to be powerful enough to thwart attacks by automatic *deobfuscators* that attempt to undo the obfuscating transformations.

The remainder of the paper will examine various code transformations that obfuscate the abstractions and data structures used in an application. 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, 5, and 6 present a catalogue of obfuscating transformations. Section 7 summarizes our results.

2 The design of a Java obfuscator

Figure 1 outlines the design of our Java obfuscation tool. In a first phase the obfuscator reads and parses all refer-

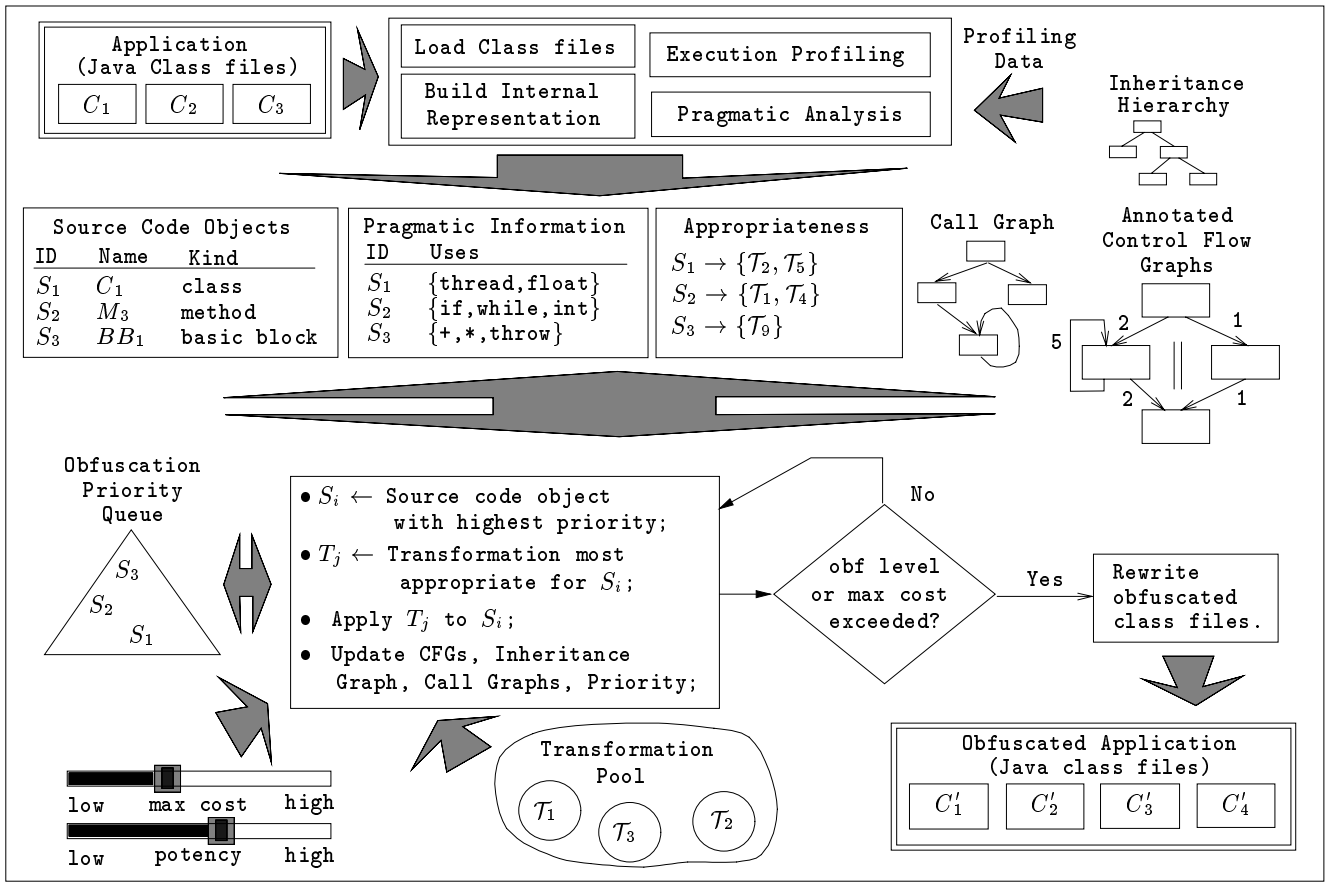


Figure 1: Architecture of a Java obfuscator.

enced class files. The second phase builds various internal data structures:

- A *Source Code Object Table* stores information about all parts of the program which may be the subject of obfuscation: methods, classes, variables, etc.
- An *inheritance graph* stores the class structure of the program.
- *Control flow graphs* are built for all methods. The CFGs are annotated with execution counts (estimated or provided through profiling) and *pragmatic information*. Execution counts are used to guide the obfuscator so that frequently executed parts of the application are not obfuscated by very expensive transformations. Pragmatic information expresses what sort of language constructs a class/method contains.
- An *appropriateness* function – mapping each source code object to the transformations appropriate for that object – is constructed from the pragmatic information. This function is used to select *stealthy* obfuscations, i.e. obfuscations that introduce code that is as similar as possible to the original code.
- A *priority queue* of source code objects is constructed based on user input and/or heuristics. The queue ensures that sensitive source code objects (i.e. parts of the program that the programmer particularly wants to protect from theft) are given higher levels of obscurity than "bread-and-butter" code.

The third phase repeatedly applies code transformations to the application until the required level of obfuscation has been achieved or the maximum execution time/space cost accepted by the user has been exceeded. Finally, the obfuscator generates a new version of the application; obfuscated but functionally equivalent to the original one.

3 Obfuscating transformations

Obfuscating transformations were first introduced in Collberg [5]. $P \xrightarrow{\mathcal{T}} P'$ is an obfuscating transformation, if

a) $\mathcal{T}$ transforms source program P into target program P',

b) P and P' have the same *observable behavior* (except in cases of non-termination or error-termination), and

c) $\mathcal{T}$ is a *potent* transformation, i.e. it renders P' more obscure than 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.

3.1 Transformation quality

Collberg [5] also introduces the concept of *quality* of an obfuscating transformation $\mathcal{T}$, a combination of four measures:

potency The potency of $\mathcal{T}$ measures how much more *obscure* (or *complex* or *unreadable*) $\mathcal{T}$ renders the application.

resilience The resilience of $\mathcal{T}$ measures how well the transformation holds up under attack from an automatic deobfuscator. 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 variable names or abstractions) 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.

stealth The stealth of $\mathcal{T}$ measures how well obfuscated code blends in with the rest of the program. If $\mathcal{T}$ introduces new code that differs wildly from what is in the original program it will be easy to spot for a reverse engineer.

cost The cost of $\mathcal{T}$ measures the execution time/space penalty which a transformation incurs on an obfuscated application. While some trivial transformations (such as scrambling identifiers) 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.

3.2 Increasing potency

Before we can design any obfuscating transformations we must first define what it means for a program P' to be more obscure than a program P. Any such measure of potency will, at best, be approximate since we cannot hope to measure exactly a human's ability to understand a program.

Fortunately, we can draw upon the vast body of work in the *Software Complexity Metrics* branch of Software Engineering. 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.

Of particular interest to us are the Henry [10], Chidamber [2], and Munson [14] metrics.

The Munson metric states that the complexity of a program 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; and the complexity of a record increases with the number and complexity of its fields.

The Henry metric states that the complexity of a function F increases with the number of formal parameters to F, and with the number of global data structures read or updated by F.

The Chidamber metric applies to object oriented programs. The complexity of a class C increases with the number of methods in C, the depth (distance from the root) of C in the inheritance tree, the number of other classes to which C is coupled, and the number of methods that can be executed in response to a message sent to an object of C.

Other metrics express that the complexity of a program grows with the number of predicates it contains (McCabe [13]) and with the nesting level of conditional and looping constructs (Harrison [9]).

3.3 Classifying transformations

There are some aspects of program understandability that are not covered directly by software metrics. For example, it should be obvious that there is much valuable information about a program in comments, strings, and identifiers, although these do not enter into any metrics formula.

Similarly, according to the Munson metric a two-dimensional array is more complex than a one-dimensional one. This fails to capture the fact that 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 the array is folded into a one-dimensional structure the Munson metric would indicate that the transformed program is less complex than the orig-

inal one, when, in fact, much useful information has been lost.

With this in mind we can attempt to classify obfuscating transformations according to the kind of information they target. *Layout* transformations are typical of current Java obfuscators such as Crema [20]. They remove source code formatting and scramble identifiers. *Control* transformations increase the McCabe and Harrison metrics by introducing predicated branches. *Data* transformations increase the Munson, Henry, or Chidamber metrics. *Abstraction* transformations remove programmer-defined abstractions or introduce spurious ones.

In this paper we describe transformations that obfuscate built-in data types and data and procedural abstractions introduced by the programmer.

3.4 Opaque predicates

Most control transformations and, as we will see, some data transformations, rely on the existence of *opaque predicates*. Informally, a predicate P is opaque if its value is known *a priori* to the obfuscator, but this value is difficult for the deobfuscator to deduce. For a predicate P we write P^F (P^T) if P always evaluates to `False` (`True`).

As an example, consider the following transformation where the obfuscator has introduced a bogus if-statement:

```
main () {                    main () {
  S1;                          S1;
  S2;          ⇒T⇒             if (7y^2 − 1 ≠ x^2)^T  S2;
  S3;                          S3;
}                            }
```

In spite of the introduced if-statement, statement S_2 will always execute. The reason is that the opaque predicate $7y^2 - 1 \neq x^2$ will always evaluate to `True`. The transformed code in this example is resilient to attack by any deobfuscator ignorant of elementary number theory.

Being able to create opaque predicates which are difficult for an obfuscator to crack is a major challenge to a creator of obfuscation tools, and the key to many highly resilient obfuscating transformations. Collberg [5] shows how it is possible to manufacture cheap and resilient opaque predicates based on intractable problems such as alias analysis.

4 Obfuscating data abstractions

In this section we will discuss transformations that obscure the data abstractions used in the source application. Most of the transformations are designed to directly increase the Munson or Chidamber metrics.

4.1 Modify inheritance relations

In current object-oriented languages 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 [18], 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.

Extending the inheritance hierarchy tree. According to the Chidamber metric, 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. As shown in Figures 2 (a) and (b), there are two basic ways in which we can increase this complexity: we can split up (factor) a class or insert a new, bogus, class.

When, as in Figure 2(a), we factor a class C into classes C_1 and C_2 we cannot arbitrarily move C's methods and instance variables into the resulting classes. To see this, let $C = (\{V\}, \{M\})$ where method M references instance variable V. Any factoring of C must respect the scope of V. For example, we cannot factor C into C_1 and C_2 where $C_1 = (\{\}, \{M\}), \Delta C_2 = (\{V\}, \{\})$, and $C_2 = C_1 \oplus \Delta C_2$.

To deal with this problem we build a dependence graph G for class C. The nodes of G are the members of C, and C itself. There is an edge $a \rightarrow b$ in G if the declaration of a must be in scope for b. If there is a path $C \rightsquigarrow y$ in G, then y must be declared in the child class C_2. If there is a path $x \rightsquigarrow y$ in G then either x and y are both declared in the same class or x is declared in the parent class C_1. See Figure 3 for an example.

Another 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 2(d). We can also insert bogus code which appears to create instances of all introduced classes. For example, the statement ⌜`if` (P^F) `x=new` C_1⌝ appears to create an instance of class C_1.

False refactoring. Figure 2(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 [15]. Refactoring is a two-step process. First, it is detected

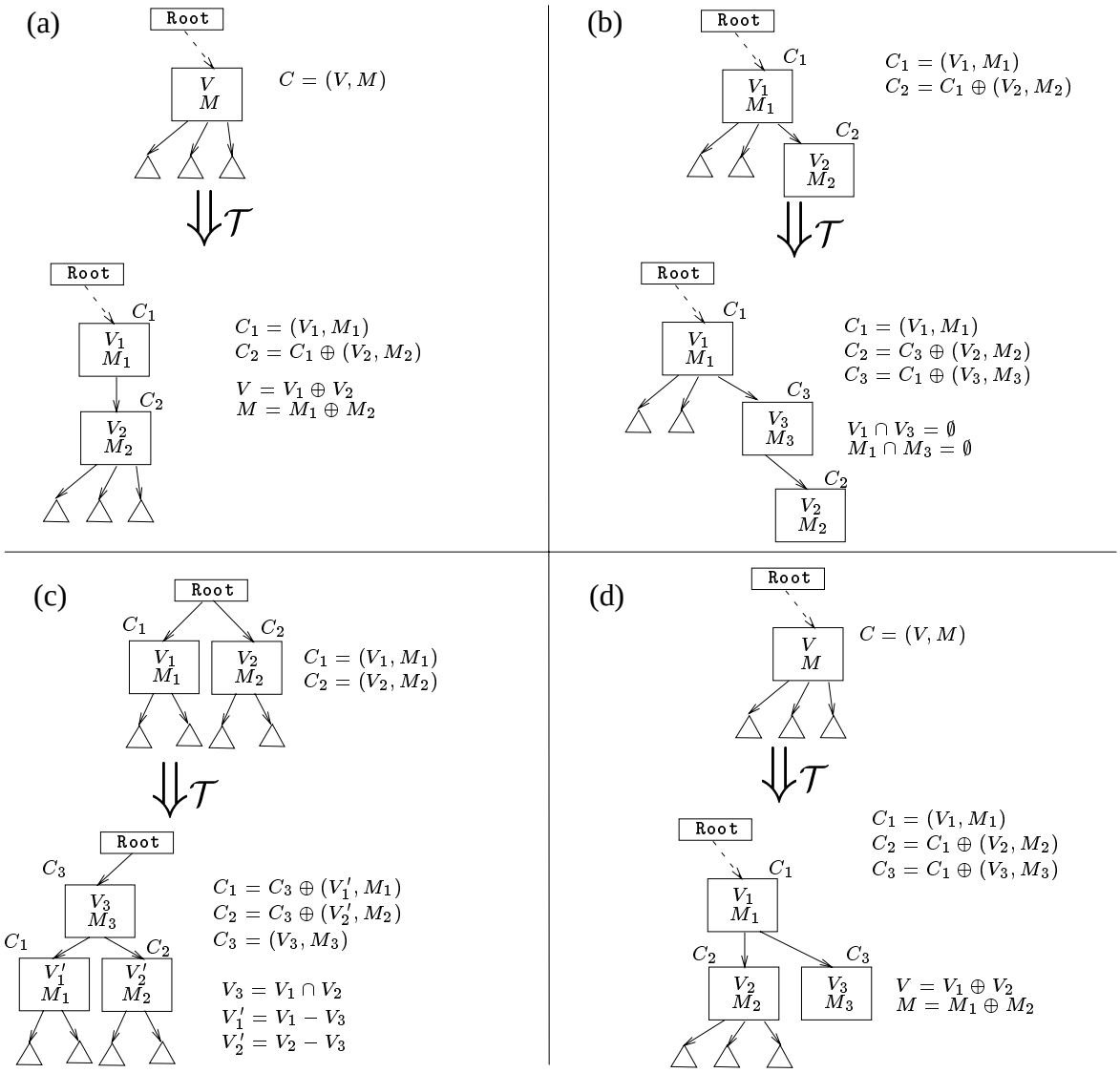


Figure 2: Modifications of the inheritance hierarchy. `Root` is the root of the inheritance tree (`Object` in Java). Triangles represent subtrees. There is an arrow from class C_1 to C_2 if C_2 inherits from C_1. The two basic operations, class factoring and class insertion, are shown in (a) and (b), respectively. After factoring class C, all references to C in the program should be replaced by C_2. 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 C_3 is created for C_1. In (c) two classes C_1 and C_2 without common behavior are given the same bogus parent C_3.

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.

4.2 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).

```
class C {
  float f;
  C v;
  C (float s){f=s};
  void P(){f=8.0};
  void Q(){v=new C; P()};
}
```

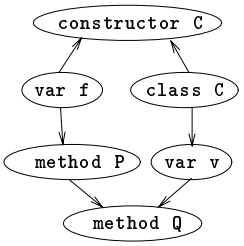


Figure 3: Example of a dependency graph built to facilitate class factoring. `Q`, `v` and the constructor `C` must all be declared in the child class. `f` and `P` may either be declared in the parent or the child class, but if `f` is put into the child class then so must `P`.

Figure 4 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 (3-5) of Figure 4 show how two integer arrays `B` and `C` can be interleaved into a new array `BC`. 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`.

5 Obfuscating Procedural Abstractions

In this section we will discuss transformations that obscure the procedural abstractions used in the source application. Some transformations affect the Henry or McCabe metrics. Others merely break up user-defined abstractions or introduce new bogus abstractions, and hence destroy the "natural" structure of the programmer's code.

5.1 Table interpretation

One of the most effective (and expensive) transformations is *table interpretation*.[1] 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 virtual machine and executing a different section of the obfuscated application. See Figure 5 for an example.

Since there is usually 1–2 orders 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 very high level of protection.

While the potency of this transformation is very high, the resilience is rather low. A deobfuscator could always just inline the code for each bytecode instruction prior to decompilation. There are two ways to increase the resilience. First, the bytecode string could be converted to a program that produces it, as explained in Section 6.2. Secondly, the original code can itself be obfuscated – for example by inserting bogus predicated branches protected by opaque predicates – prior to being translated to the specialized bytecode. This is illustrated by transformation $\mathcal{T}_1$ in Figure 5.

[1] Thanks to Buz for pointing this out.

5.2 Inline and outline methods

Inlining is, of course, a important code optimization technique. It is also an extremely useful obfuscating 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 6 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 end of P's code and the beginning of Q's code into a new procedure R.

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 [6] and select the appropriate code by branching on the type of m. Hence, even after inlining and removal of methods, the obfuscated code may still contain some traces of the original abstractions.

5.3 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 7 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.

6 Obfuscating built-in data types

In this section we will present transformations that obscure the basic data types (such as integers and strings)

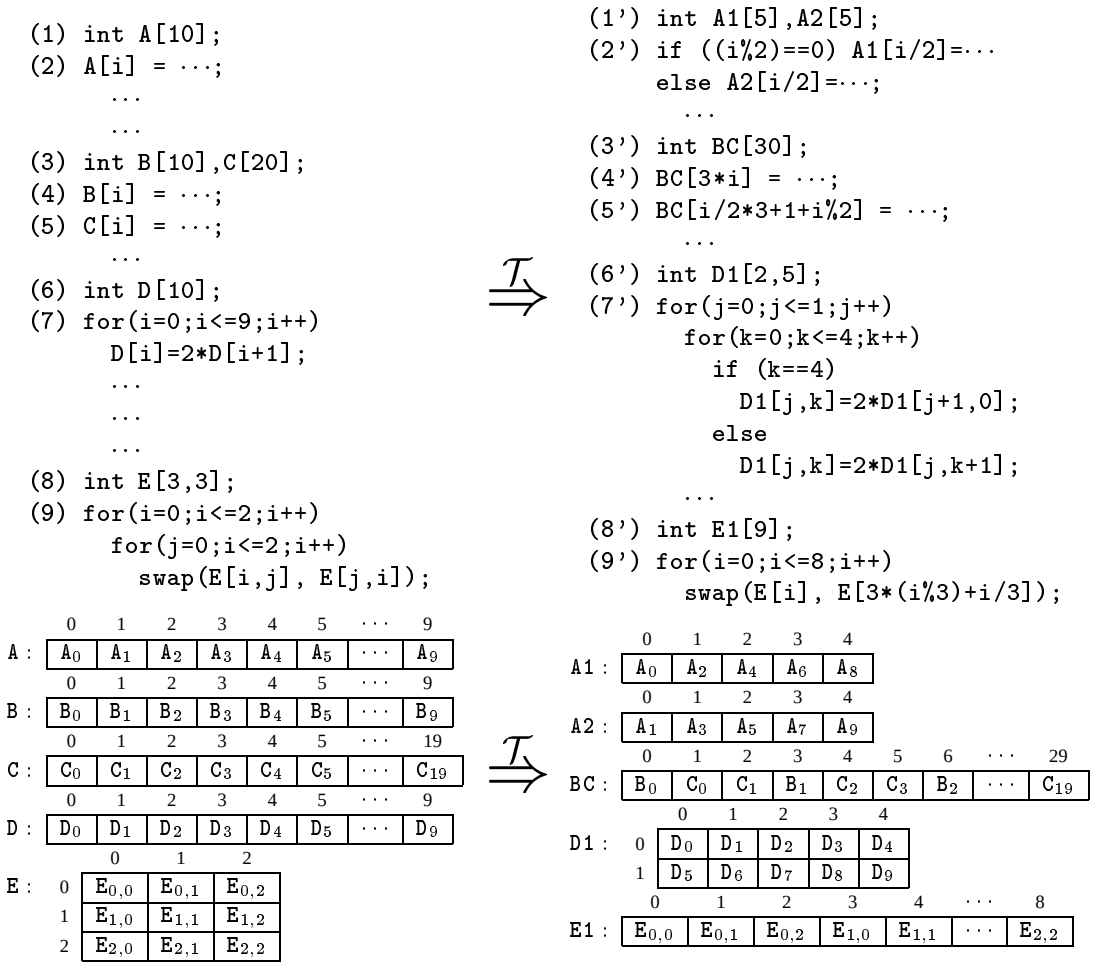


Figure 4: Array Restructuring. Array splitting (statements (1-2)), array merging (statements (3-5)), array folding (statements (6-7)), and array flattening (statements (8-9)). ⌜`int X[n]`⌝ is a shorthand for ⌜`int[] X = new int[n]`⌝.

used in the source application. Generally, designing such transformations is difficult since these types form such an integral part of most programming languages. For this very reason the transformations are often high in cost and low in stealth. Nevertheless, combined with other transformations these obfuscations can sometimes be quite effective.

6.1 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 and resilience 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 8(a) shows a possible choice of representation for split boolean variables. The table indicates that boolean variable V has been split into two short integer variables p and q. If $p = q = 0$ or $p = q = 1$ then V is `False`, otherwise, V is `True`.

Given this new representation, we have to devise 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 8(c) and (d), respectively. Given two boolean vari-

```
int Sum(int A[]) {
  int i, sum=0;
  int n=A.length;
  for (i=0;i<n;i++)
    sum += A[i];
  return sum;
}
```

$\Downarrow \mathcal{T}_1$

```
int Sum'(int A[]) {
  int i, sum=0;
  int n=A.length;
  for (i=0;i<n;i++)
    if (p != q)^T
      sum += A[i];
  return sum;
}
```

$\Rightarrow \mathcal{T}_2$

```
int Sum(int A[]) {
  int sum=0, i=0, pc=0;
  int s[]=new int[5], sp=-1;
  loop:  while (true)
    switch("fcghiabced".charAt(pc)) {
      case 'a':  sum += s[sp--]; pc++; break;
      case 'b':  i++; pc++; break;
      case 'c':  s[++sp] = i; pc++; break;
      case 'd':  if (s[sp--] > s[sp--]) pc -= 8;
                 else break loop; break;
      case 'e':  s[++sp] = A.length; pc++; break;
      case 'f':  pc += 7; break;
      case 'g':  s[sp] = A[s[sp]]; pc++; break;
      case 'h':  s[++sp] = (p != q)?1:0; pc++; break;
      case 'i':  if (s[sp--]==1) pc+=2 else pc++; break;
    }
  return sum;
}
```

Figure 5: The Java method `Sum` on the left is obfuscated by translating it into the bytecode `"fcghiabced"`. This code is then executed by a stack-based interpreter specialized to handle this particular virtual machine code. This is akin to Proebsting's superoperators [16]. To increase the resilience of this transformation, `Sum` is first obfuscated by inserting a bogus if-statement whose opaque predicate (which depends on two un-aliased pointer variables `p` and `q`) will always evaluate to `True`. Even after deobfuscating the interpreter, the resulting code is still obfuscated.

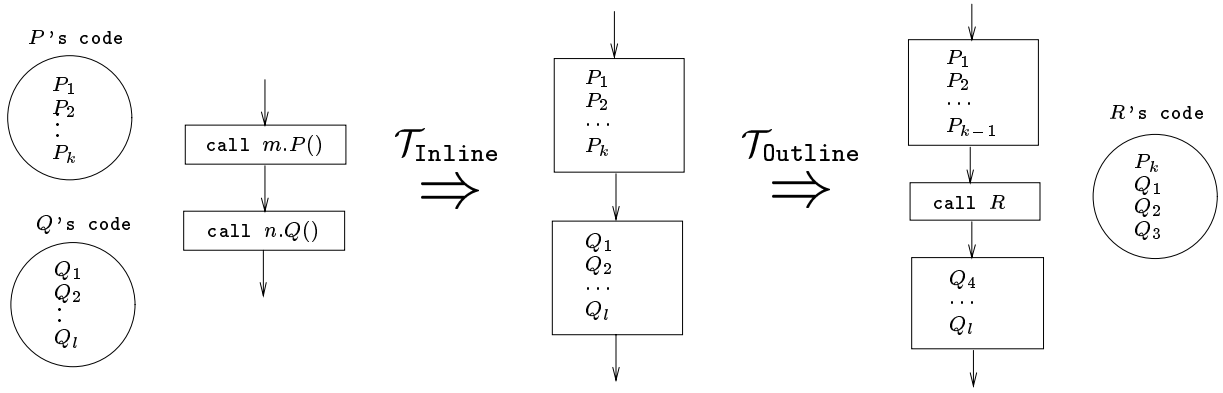


Figure 6: Inlining and outlining transformations.

ables $V_1 = [p, q]$ and $V_2 = [r, s]$, $\ulcorner V_1 \& V_2 \urcorner$ is computed as $\ulcorner \texttt{AND}[2p + q, 2r + s] \urcorner$.

In Figure 8(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 8(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 $\ulcorner$`A&B`$\urcorner$.

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 lookup tables of Figure 8(b-d) are not constructed at obfuscation-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 8(e).

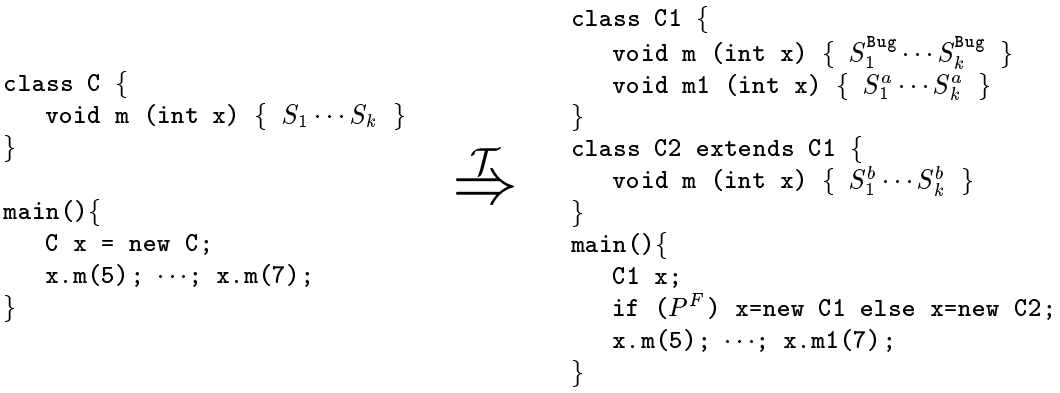
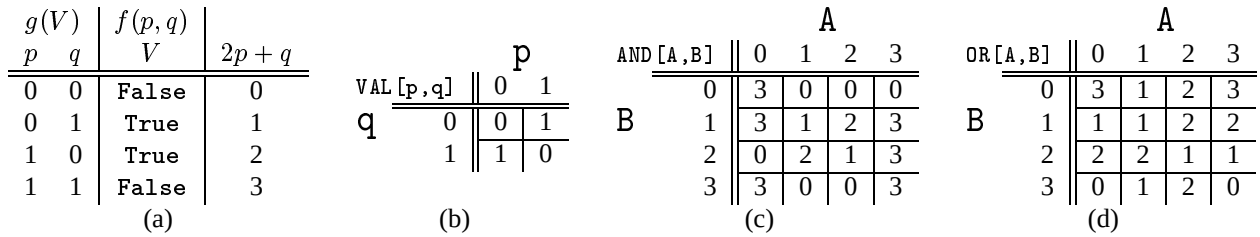


Figure 7: Cloning methods. `C2.m` and `C1.m1` have been generated by applying different obfuscating transformations to the body of `C.m`. The calls ⌜`x.m(5)`⌝ and ⌜`x.m1(7)`⌝ look as if they were made to two different methods, while in fact they go to different-looking methods with identical behavior.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



$g(V)$ p	q	$f(p,q)$ 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
q=1	1	0

(b)

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

(c)

OR[A,B]	A=0	A=1	A=2	A=3
B=0	3	1	2	3
B=1	1	1	2	2
B=2	2	2	1	1
B=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;           (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 8: Variable splitting example. Tables (b-d) are used to compute boolean operations. They are either constructed at obfuscation-time and stored as static data in the obfuscated application, or generated at run-time by the obfuscated application itself.

6.2 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 – 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 9. 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 would be unstealthy in most codes. Much higher potency and resilience may be achieved if the `G`-function were broken up into smaller components embedded into the "normal" control flow of the source program.

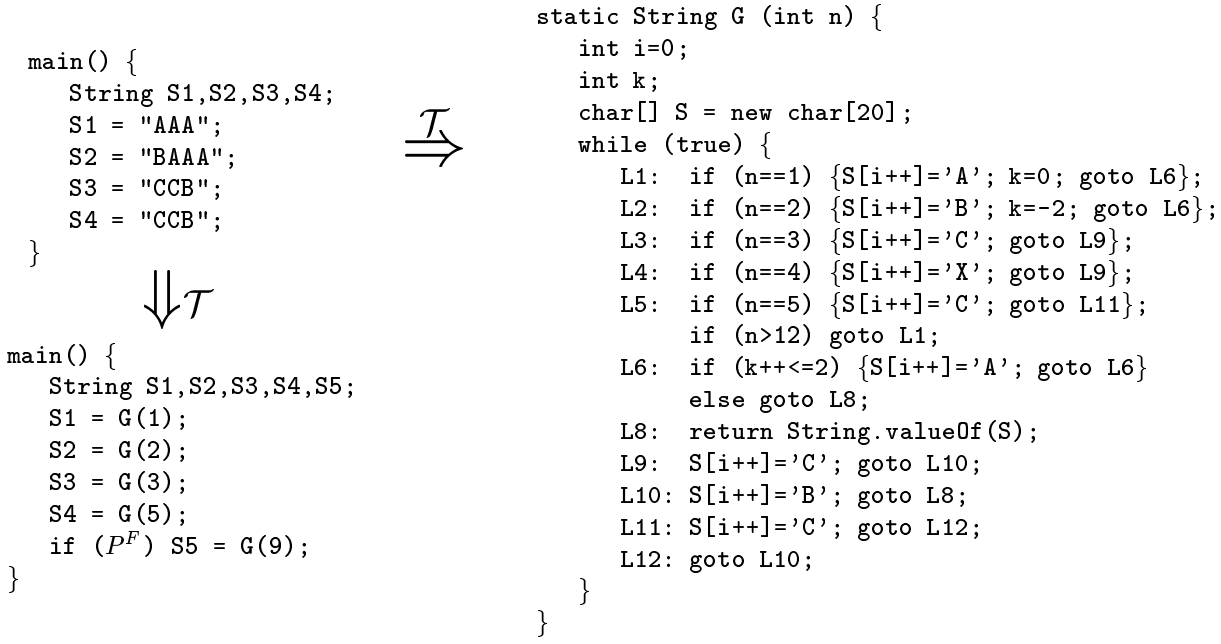


Figure 9: A function producing the the strings "AAA", "BAAAA", and "CCB". Note that this type of code cannot be coded directly in Java, since the language lacks gotos. It can, however, be coded at the bytecode level.

6.3 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

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

we get the arithmetic identities in Figure 10(a). Some simple examples are given in Figure 10(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 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 10(b) we could insert operations that appear to merge Z's two halves, for example by bitwise rotation: $\ulcorner$`if (`P^F`) Z = rotate(Z,5)`$\urcorner$.

7 Summary

In a previous paper [5] we showed that it is possible to obfuscate the control flow of an application with a high degree of *resilience* (resistance to attack by automatic deobfuscators) and at low time/space *cost*. In this paper we have shown that data structures and abstractions can also be obfuscated, in many cases with only minor impact on execution time/space cost.

The transformations presented in this paper are only a few of a large catalogue [4] of obfuscations which target every aspect of a program. 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.

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, our obfuscator (which targets Java class files) 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 [19].

Acknowledgments

We thank the anonymous referees for their insightful comments on a draft of this paper. We would also like to thank Todd Proebsting, Chris Fraser, Mark Burgess, Andrew Wright, and Bob Uzgalis for stimulating discussions.

(a)

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

(b)

```
(1) int X=45,Y=95;          (1') long Z=167759086119551045;
(2) X += 5;                 (2') Z += 5;
(3) Y += 11;          ⇒     (3') Z += 47244640256;
(4) X *= c;                 (4') Z += (c-1)*(Z & 4294967295);
(5) Y *= d;                 (5') Z += (d-1)*(Z & 18446744069414584320);
```

Figure 10: 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`.

References

[1] David Aucsmith. Tamper resistant software. In *Information Hiding*, pages 317–334, May/June 1986. LNCS 1174.

[2] 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.

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

[4] 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/index.html`.

[5] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *SIGPLAN-SIGACT POPL'98*. ACM Press, San Diego, CA, January 1998. `http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborsonLow98a/index.html`.

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

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

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

[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.

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

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

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

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

[15] 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-superclasses.ps`.

[16] Todd Proebsting. Optimizing ANSI C with superoperators. In *POPL'96*. ACM Press, January 1996.

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

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

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

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

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