The Obfuscation Executive[*]

Kelly Heffner Christian Collberg
Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA
{kheffner,collberg}@cs.arizona.edu

Technical Report TR04-03

Abstract

Code obfuscations are semantics-preserving code transformations used to protect a program from reverse engineering. There is generally no expectation of complete, long-term, protection. Rather, there is a trade-off between the protection afforded by an obfuscation (i.e. the amount of resources an adversary has to expend to overcome the layer of confusion added by the transformation) and the resulting performance overhead.

An obfuscation tool will generally apply a series of obfuscation algorithms to the same application. While each individual obfuscation may add a trivial amount of confusion, the layering of and interaction between the different transformations can result in a highly obfuscated application.

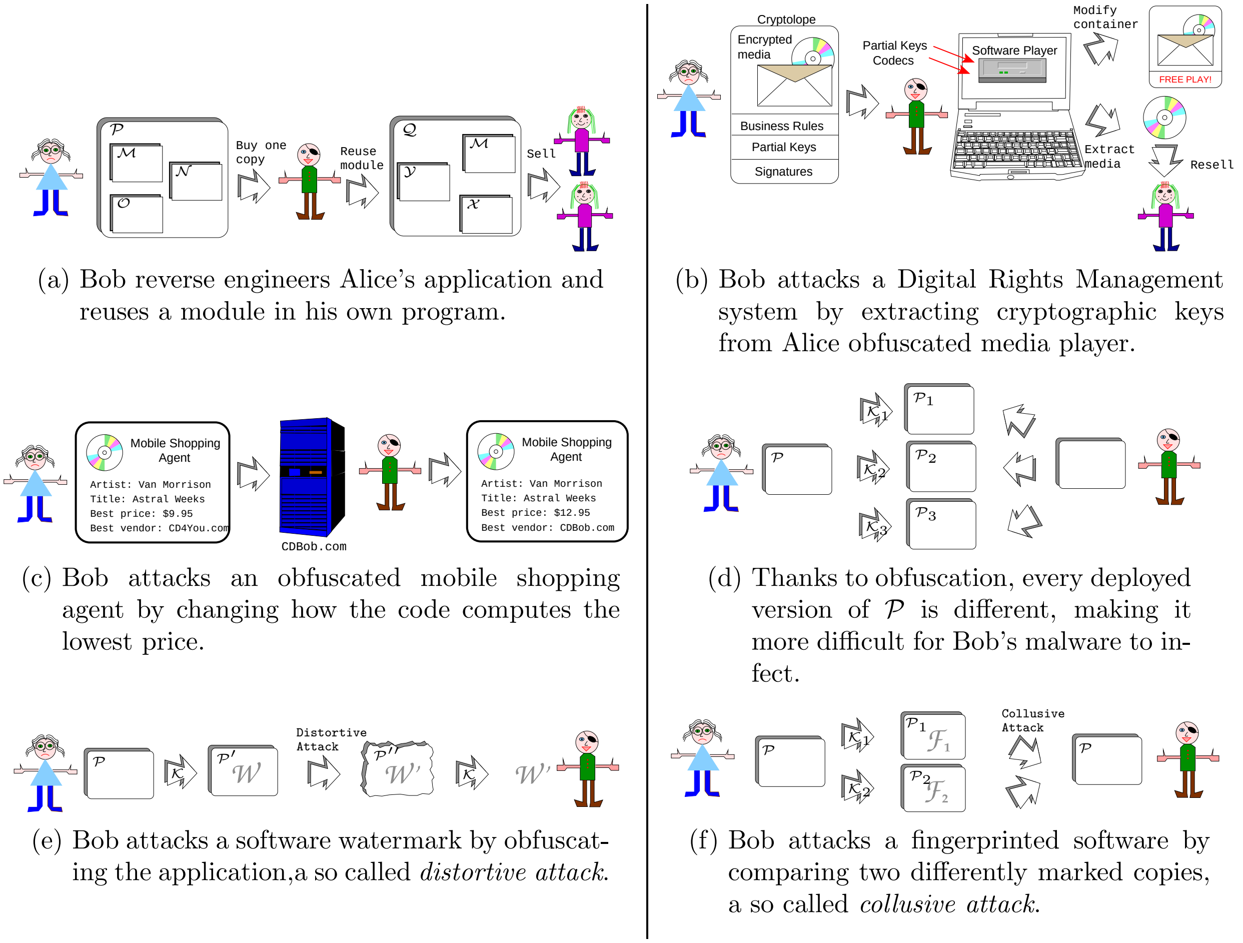
In this paper we examine the problems that arise when constructing an *Obfuscation Executive*. This is the main loop in charge of a) selecting the part of the application to be obfuscated next, b) choosing the best transformation to apply to this part, c) evaluating how much confusion and overhead has been added to the application, and d) deciding when the obfuscation process should terminate.

1 Introduction

A *code obfuscator* is a tool which—much like a code optimizer—repeatedly applies semantics-preserving code transformations to a program. However, while an optimizer tries to make the program as fast or as small as possible, the obfuscator tries to make it as *incomprehensible* as possible. Obfuscation is typically applied to programs in order to protect them from being reverse engineered or to protect a secret stored in the program from being discovered.

In this paper we will describe the design of an *Obfuscation Executive* (OE), implemented within the SandMark software protection research tool [9]. The OE is the overall loop that applies obfuscation algorithms to parts of the program to be protected. In many ways the OE functions similar to a compiler's optimization pass: it reads and analyzes an application and repeatedly applying semantics-preserving transformations until some termination condition has been reached. Ideally, the executive should be able to pick an optimal set of transformations and an optimal set of program parts to obfuscate. The only necessary user interaction is to indicate to the tool what "optimal" means: i.e. how much execution overhead the user can accept, how much obfuscation he wants to add, and which parts of the application are security- or performance-critical. The two major issues with this process is the order in which transformations are applied (the "phase-ordering-problem") and how to decide that the process should terminate. In the case of a code optimizer the order of transformations is usually fixed. The optimizer typically terminates when there are no more changes to the application or when all transformations have been tried at least once. As we will see from this paper, in the case of the OE neither phase-ordering nor termination is this simple.

[*] This work was supported in part by the National Science Foundation under grant CCR-0073483 and the Air Force Research Lab under contract F33615-02-C-1146.



(a) Bob reverse engineers Alice's application and reuses a module in his own program.

(b) Bob attacks a Digital Rights Management system by extracting cryptographic keys from Alice obfuscated media player.

(c) Bob attacks an obfuscated mobile shopping agent by changing how the code computes the lowest price.

(d) Thanks to obfuscation, every deployed version of $\mathcal{P}$ is different, making it more difficult for Bob's malware to infect.

(e) Bob attacks a software watermark by obfuscating the application,a so called *distortive attack*.

(f) Bob attacks a fingerprinted software by comparing two differently marked copies, a so called *collusive attack*.

Figure 1: Applications of code obfuscation

.

The remainder of this paper is organized as follows. In Section 2 we describe the overall design of the OE. In Section 3 we describe how dependencies between transformations are modeled. In Section 4 we describe the details of the obfuscation loop. In Section 5 we discuss related work, and in Section 6 we summarize our results.

1.1 Applications of Obfuscation

We will start by considering some applications of code obfuscation. In particular, we are going to examine scenarios where software—or a secret hidden in software—is protected by code obfuscation. We will also see that obfuscating transformations can be used to facilitate *attacks* against software. Figure 1 illustrates these scenarios.

Malicious Reverse Engineeering, 1(a): As more code is shipped in easily decompilable intermediate code formats such as Java bytecode and MSIL [23], software developers become more conserned with preventing competitors from decompiling their code and discovering its overall design or a particularly valuable algorithm. It is currently believed that no amount of obfuscation will offer complete protection against a determined adversary. Rather, the goal is to make Alice's code so convoluted that Bob finds it no easier to reuse Alice's code than to rewrite it from scratch himself.

Digital Rights Management, 1(b): DRM is a technique for protecting against media piracy. The idea is and wrap up digital content in a virtual container (a *cryptolope*), along with *business rules* which describe how the media might be played, sold, traded, rented, etc., and the associated costs. The contents can only be enjoyed through a special player which contains cryptographic keys to unlock the media. An adversary who is able to decompile and examine the player's code can locate the hidden keys which will allow him to enjoy the media for free. For this reason, software players must be heavily obfuscated [2] and tamper-proofed [4].

Malicious Mobile Agents, 1(c): A mobile shopping agent wanders between on-line stores to find the best deal for a particular item. An unscrupulous store might manipulate the agent to make their own deal seem most advantageous. Hohl [18] suggests that obfuscating the agent can be used to slow down such an attack.

Artificial Diversity, 1(d): Code obfuscation techniques have been applied to operating systems to protect them against class attacks by malware, such as viruses and worms [5,7]. The idea is to randomize code such that a malicious agent will not be able to locate or take advantage of a known vulnerability. Of course, viruses themselves make use of obfuscation techniques to avoid detection by virus scanners, to spectacular success.

Distortive Attacks, 1(e): Software fingerprints [14] are unique customer identifiers embedded into a program. The idea is for a vendor to be able to trace the origins of her pirated software back to the pirate who bought it. A pirate, however, can obfuscate a program prior to reselling it, in the hopes of destroying the fingeprint.

Collusive Attack Protection, 1(f): An attacker can also launch a collusive attack against a fingerprinted program by buying two differently marked copies and comparing them to discover the location of the fingerprint. To prevent such an attack the vendor should apply a different set of obfuscations to each distributed copy, ensuring that comparing two copies of the same program will yield little information.

1.2 Obfuscation Algorithms

A large number of *Software Complexity Metrics* [6,15,17,20,22,24] have been defined to measure the readability, understandability, and maintainability of software. Code obfuscating transformations are designed to maximize these complexity metrics while minimizing the resulting performance penalty. Typically, algorithms are made up of combinations of the following operations: **fold/flatten** turn a d-dimensional construct into $d+1$ or $d-1$-dimensional ones; **split/merge** turn a compound construct X into two constructs $\{a, b\}$ or two constructs a and b into a compound construct X; **box/unbox** add or remove a layer of abstraction; **ref/deref** add or remove a level of indirection; **reorder** swap two adjacent constructs; **rename** assign a new name to a labeled construct. Here, *construct* refers to any programming language object the obfuscator can manipulate. For example, if A is a vector then **fold**(A) turns A into a two-dimensional array. If P is a static method then **ref**(P) turns it into a virtual one. If B is a basic block then **split**(B) breaks it into two parts by inserting a bogus branch (a so called *opaque predicate* [13]). Many more such transformations have been described in the literature [7,11,12,26].

Obfuscating transformations are characterized by their *potency* (the amount of confusion they add), their *resilience* (the extent to which they can be undone by a *de-obfuscator*), and their *cost* (the performance penalty they incur on the obfuscated application) [13].

2 The Obfuscation Loop

At the heart of any OE is a loop that chooses a part of the application to obfuscate, chooses an appropriate obfuscating transformation from a pool of candidate algorithms, and then applies the transformation. After the transformation has completed, the loop computes how much the code has changed and decides if the process should continue.

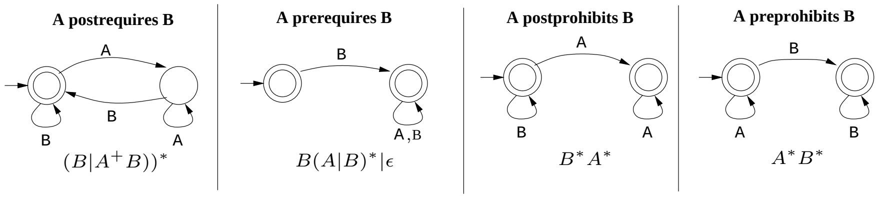


Figure 2: An FSA for each dependency type.

Unfortunately, there are a number of complications. First of all, neither the amount of protection we would like to achieve nor the amount of overhead we can accept are uniform over the application. Some subroutines may be performance-critical, others not. Some subroutines may be security-critical, others not. Secondly, given any non-trivial set of obfuscating transformations there will be restrictions on the order in which these should be applied. The reason is that an obfuscating transformation *destroys* structures in the application. This makes the obfuscated application more difficult to analyze, and, as a result, it might not be possible to apply any further transformations. Finally, not all obfuscations can be applied to all application objects. For example, obfuscations which rename classes [26,27] cannot be applied to classes that will be loaded dynamically by name. Also, if we are using obfuscations to hide a particular structure in the application (such as a watermark or a set of cryptographic keys) we cannot apply transformations that will destroy these structures.

2.1 Transformation Dependencies

SandMark's OE understands six types of dependencies between obfuscations and tools such as software watermarkers that use them. These are pre/post-suggestions, pre/post-requirements, and pre/post-prohibitions:

postsuggestion: Assume that a method splitting transformation `MethodSplit` splits a method `computePrimes` into two methods named `computePrimes_FirstHalf` and `computePrimes_SecondHalf`. Choosing such suggestive names simplifies debugging the transformation. To make sure that these names are not kept in the final obfuscated application, `MethodSplit` will use a *postsuggestion* to indicate that a name obfuscator should be run after the split.

presuggestion: Assume that a Java method signature scrambling algorithm `PackFormals` turns all the parameters in a method into an array of objects. It would, for example, turn the method ⌜`foo(Integer a,String b,char c){}`⌝ into ⌜`foo(Object[] args,char c){}`⌝. This algorithm will only affect object parameters, not parameters of primitive types. Therefore, in order to make the transformation useful, a transformation that promotes primitives to objects should be run before this algorithm is run. This transformation would turn ⌜`foo(Integer a,String b,char c){}`⌝ into ⌜`foo(Integer a,String b,Character c){}`⌝ after which `PackFormals` would produce ⌜`foo(Object[] args){}`⌝. A dependency of this type is called a *presuggestion*.

postrequirement: Assume an software watermarking algorithm (such as the CT [10]) that embeds the watermark in a data structure in the application. To simplify implementation and debugging the watermarker creates a class `Watermark` that contains the code for building the mark: ⌜ `class Watermark { Watermark left, right; void createGraph() {...} }` ⌝ A call to the method `createGraph()` is embedded into the application. Obviously, this is not stealthy. Therefore the watermarking algorithm requires that an inlining transformation and a name obfuscating transformation be run after the watermarking algorithm. This is a *postrequirement* dependency.

prerequirement: Assume that there exists two obfuscations `PublicizeFields` (which makes every field of a class **public**) and `InlineMethod` which performs inter-class inlining. Since an inlined method body is not allowed to reference any **private** fields we would make `InlineMethod` *prerequire* `PublicizeFields`.

postprohibition: Assume a software watermarking algorithm that embeds a secret watermark into the application by changing the frequency of different instruction patterns. Stern [25] presents such an algorithm. Any changes to the method bodies of the watermarked code may destroy the mark. Therefore, any obfuscating algorithm that modifies method bodies should not be run after this mark has been embedded. This type of dependency is called a *postprohibition*.

preprohibition: Assume an obfuscating transformation `MergeArrays` that performs alias analysis to detect the location of two arrays to merge. This algorithm should not be run after any algorithm that makes alias analysis difficult. Collberg [13] presents such algorithms. This type of dependency is called a *preprohibition*.

postsuggestion/presuggestion: Suggestions are similar to requirements in the resulting language of transformations that they allow. However, while breaking a requirement will put the program in a corrupt state or make a software watermark obvious, a suggestion is just a hint to the OE by the obfuscation author that certain transformations work well together.

In our current SandMark implementation each obfuscation and watermarking algorithm specifies the effects that it may have on the code. It also specifies *properties* of other algorithms that are postrequired, presuggested, etc. For example, a method splitting transformation `MethodSplit` might list `OBFUSCATE_METHOD_NAMES` as a postsuggestion, indicating to the OE that some algorithm (any one will do) that has this property should be run after `MethodSplit`. In general,

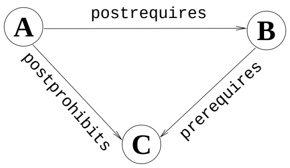
- to fulfill a requisite dependency **one** algorithm with the specified property must be run;
- to fulfill a suggestion dependency any **one** algorithm with the specified property could be run; and
- to fulfill a prohibition dependency **no** algorithm with a specified property should be allowed to run.

In general, simple obfuscators do not need to specify many dependency relationships with other obfuscators. However, when using obfuscation with software watermarking, dependencies are necessary to make sure that the obfuscations successfully camouflage the watermark without destroying the structures that it is embedded in. From a software design standpoint, the dependency framework also allows for more complex obfuscators to be created modularly and without concern for the transformations performed around it.

3 Modeling Dependencies

Our goal is to construct an OE algorithm to honor transformation dependencies and to find the "optimal" set of transformations to apply to the "optimal" set of application objects. In fact, we are interested in constructing *families* of such algorithms, to be targeted at the many and varied applications of obfuscation seen in Figure 1. The most important result in this paper is the design of a model which encodes transformation dependencies, obfuscation potency and performance overhead, as well as the desired level of obfuscation and overhead of each application object. The model is based on weighted finite state automata and can be used as the basis for many on-line and off-line algorithms.

Let us first consider an example with three transformations A, B, and C. A postprohibits a property that C has, A postrequires a property that only B has, and B prerequires a property that only C has:



```
class c1{
  m1(){...}
  m2(){...}
}
class c2{
  m(){...}
}
```

Trans-	Obfuscation			Prop-	Requirement		Prohibition	
formation	**Level**	**Potency**	**Overhead**	**erties**	**Pre**	**Post**	**Pre**	**Post**
A	Method	1	0.9	p_1	p_2			
B	Method	0.5	0.3	p_2, p_3		p_1		
C	Class	0.1	0.4	p_1, p_3				
D	Class	0.2	0.2	p_2			p_3	
E	Application	0.01	0.1	p_3				p_1

Figure 3: A running example. A and B are method level obfuscators, C and D are class level obfuscators, and E applies to an entire application.

The order in which these transformations can be run is CAB, $CBAB$, $CBBAB$, or, more generally, any sequence that matches the regular expression
$\epsilon|C(C|B)^*(A^+B)^*$.

This observation leads us to a model where the dependencies between transformations are represented by a finite state automaton (FSA). The language accepted by this machine contains all the possible sequences of transformations that can be executed. Given this model, the design of a new OE reduces to the problem of constructing an algorithm that chooses a finite subset of the (typically) infinite set of strings generated by the FSA.The heuristics of such algorithms will make use of FSA *edge weights* representing the "goodness" of traversing each edge.

We will next describe how the FSA is built, then how edge weights are computed, and, finally, in Section 4 describe an on-line OE algorithm that makes use of the FSA model.

3.1 Building the FSA

Each type of dependency has an equivalent regular language. It suffices to consider prohibition and requirement dependencies since suggested dependencies can be modeled by modifying the FSA edge weights. This will be shown in Section 3.2.

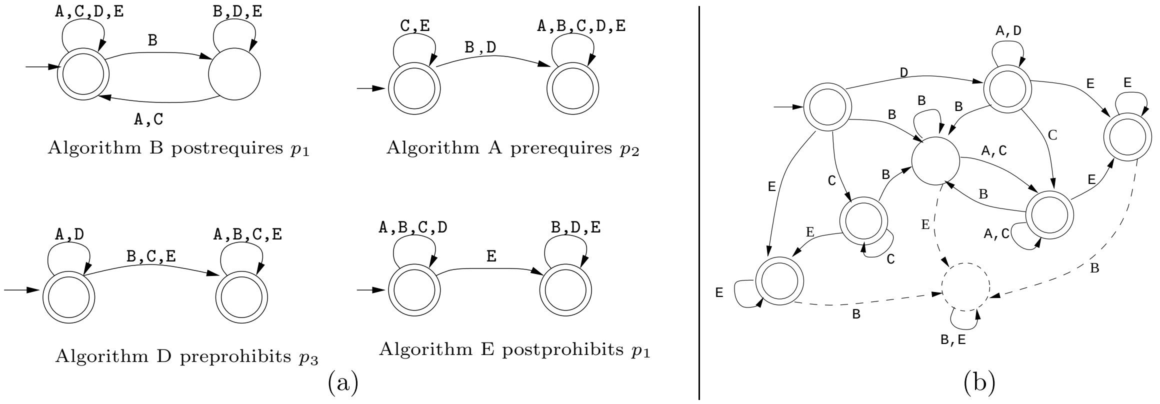
Figure 2 shows the four FSAs that correspond to each dependency type. The figures show transitions only for the two transformations involved in the dependency; for any transformation that is not in the dependency, the transition is just a self-loop.

To build the regular language for the entire set of obfuscating transformations we take the intersection of the languages from each dependency and Σ^*. The resulting language is the set of all possible sequences in which the transformations could be applied. To model the fact that dependencies apply to properties of transformations, rather than transformations themselves, we simply replace a single transformation in Figure 2 with all of the transformations that have a particular property.

Consider the running example in Figure 3 which describes five obfuscating transformations A, B, C, D, and E with three properties p_1, p_2, and p_3. Figure 4(a) shows the models for the four dependencies in the example. Taking the intersection of the languages represented by these FSAs we get the FSA in Figure 4(b), which models every possible candidate sequence allowed by the dependencies.

To make full use of the finite state machine model, we must integrate the idea that transformations are run on application objects, not always the entire application. By running a transformation on one application object, possibly fulfilling prerequirements and prohibiting other transformations, the result affects only a subset of objects. In order to keep track of these object-level changes, the target of the transformation must be included with each transformation in the sequence. Thus, each symbol in our alphabet for the sequence becomes an ordered pair (*transformation*, *target*).

We should also note that modifications to single application objects do not just affect that object. Obfuscating a method may affect not only the class that the method is in, but the entire application. This



Algorithm B postrequires p_1

Algorithm A prerequires p_2

Algorithm D preprohibits p_3

Algorithm E postprohibits p_1

(a)

(b)

Figure 4: The transformation properties from the running example in Figure 3 produces the four FSAs in (a). Taking the intersection of the generated languages produces the FSA in (b). Note that no paths exist to an accepting state from some states (dashed in (b)) so those nodes would be removed .

is true, for example, of an obfuscation that reorders method arguments. For simplicity we will assume that when a transformation is run on an object x, the effect is spread to all objects that contain x, and all objects that x contains. We will call this the *range* of an object. This is a conservative approximation on the real spread. A less coarse approximation would be advantageous but more difficult to compute.

During FSA construction the states represent sets of application objects. We will refer to the set of objects for a given state q as $s(q)$. We will refer to the set of objects that is in the range of an application object x as $r(x)$.

First we will construct the FSA for a prerequirement dependency, where a transformation T prerequires a property p. We define an FSA (Q, Σ, δ, q_0, F) with the following properties:

- Q, the set of states, is composed of the power set of the set of all application objects that are a target of T.
- Σ is the set of ordered pairs (T, x) where T is a transformation and x is a target application object for that transformation.
- The transition function δ is given in Figure 5.
- q_0 is the state such that $s(q_0) = \emptyset$.
- $F = Q$ is the set of accepting states.

Figure 6 shows the partial FSA for the preprohibition of property p_3 before transformation D.

Next, we construct the FSA for a postrequirement dependency, where transformation T postrequires property p. The FSA is identical to the previous one, except for the transition function in Figure 5 and that $F = \{q_0\}$. The transition functions for FSAs for a preprohibition and postprohibition dependencies are similar and shown in Figure 5.

3.2 Building the Probabilistic FSA

The FSA generates a language of strings of (*obfuscation*, *target*) tuples. These strings represent a series of obfuscations to run on the application. Obviously, some strings are more desirable than others in that they result in more highly obfuscated programs with lower performance penalty. To capture this, we create a probabilistic FSA by giving each edge an *edge weight*. In an edge

$$a \xrightarrow{(T,x),w} b$$

w represents the "goodness" of applying transformation T to application object x when the OE is in state a. In Section 4 we will show how this model allows for a very simple, yet effective, on-line OE algorithm.

The FSA edge weight w is a tuple

$$\langle \mathit{Potency}(T), \mathit{Degradation}(T), \mathit{ObfLevel}(x), \mathit{PerfImport}(x) \rangle$$

of four real numbers in the range $[0, 1]$:

- $\mathit{Potency}(T)$ measures the obfuscation potency of T. It is computed by running each obfuscation on a set of benchmarks and computing the change in software complexity. See Section 3.3.
- $\mathit{Degradation}(T)$ measures the performance degradation of T. This is computed by running each obfuscation on a set of benchmarks and computing the change in execution time. See Section 3.4.
- $\mathit{ObfLevel}(x)$ is the desired obfuscation level of application object x, as assigned by the user.
- $\mathit{PerfImport}(x)$ is the importance of performance of application object x, a combination of user assignment and profiling data.

Our model does not specify how a particular OE will make use of the weight tuple. In our current implementation the tuple is mapped down to a single real number which represents the overall goodness of choosing a particular edge. This is explained in Section 3.5. We will next show how to estimate $\mathit{Potency}(T)$ and $\mathit{Degradation}(T)$.

3.3 Estimating Obfuscation Potency

Each obfuscating transformation T is assigned a real number (in the range $[0, 1]$) $\mathit{Potency}(T)$ that represents the relative potency of the transformation in comparison with the rest of the transformations known to the obfuscator. $\mathit{Potency}(T)$ is determined by calculating a set of software engineering metrics, M, on sample programs before and after obfuscation and taking the average of the change in those metrics.

The change in a set of metrics M for a program P on a transformation T is given by

$$\Delta(P, M, T) = \frac{\sum_{m \in M} |m(P) - m(T(P))|}{|M|}$$

where $m(P)$ is a software metric calculated on P and $T(P)$ is P obfuscated by transformation T. We get $\Delta(M, T)$ by averaging $\Delta(P, M, T)$ over all benchmark programs. To compute the obfuscation potency for a transformation T we normalize using the highest and lowest changes in the metrics ($\Delta_{\max}(M)$ and $\Delta_{\min}(M)$):

$$\mathit{Potency}(T) = \frac{\Delta(M, T) - \Delta_{\min}(M)}{\Delta_{\max}(M) - \Delta_{\min}(M)}$$

3.4 Estimating Performance Degradation

Each obfuscating transformation T is assigned a real number $\mathit{Degradation}(T)$ (in the range $[0, 1]$) that represents T's expected performance hit. $\mathit{Degradation}(T)$ is calculated by running every transformation T on a set of "priming" applications. These could either be a set of benchmarks such as the SpecJVM, or the application to be obfuscated itself. $\Gamma(P, T)$ is the raw performance degradation of T on program P:

$$\Gamma(P, T) = \frac{\max(\mathrm{time}(T(P)) - \mathrm{time}(P), 0)}{\mathrm{time}(P)}$$

We get $\Gamma(T)$ by averaging over all benchmark programs. To compute $\mathit{Degradation}(T)$ we normalize using the highest and lowest performance hits ($\Gamma_{\max}$ and $\Gamma_{\min}$):

$$\mathit{Degradation}(T) = \frac{\Gamma(T) - \Gamma_{\min}}{\Gamma_{\max} - \Gamma_{\min}}$$

$\delta(q,\ (t,x)) =$

	requirement	**prohibition**
pre	$\begin{cases} q', & \text{if } q \neq q',\ t \text{ has the property } p, \text{ and} \\ & s(q)+r(x)=s(q'),\ \text{ or} \\ q, & \text{if } t=T \text{ and } x \in s(q),\ \text{ or} \\ q, & \text{if } t \text{ has property } p \text{ and } r(x) \subseteq s(q),\ \text{ or} \\ q, & \text{if } t \neq T \text{ and } t \text{ does not have property } p \end{cases}$	$\begin{cases} q' & \text{if } t \text{ has the property } p, \text{ and} \\ & s(q)+r(x)=s(q'), \text{or} \\ q & \text{if } t=T \text{ and } x \notin s(q),\ \text{ or} \\ q & \text{if } t \text{ has property } p \text{ and } r(x) \subseteq s(q),\ \text{ or} \\ q & \text{if } t \neq T \text{ and } t \text{ does not have property } p \end{cases}$
post	$\begin{cases} q' & \text{if } t=T \text{ and } s(q)+r(x)=s(q'),\ \text{ or} \\ q' & \text{if } t \text{ has the property } p, \text{ and} \\ & s(q)-r(x)=s(q'),\ \text{ or} \\ q & \text{if } t=T \text{ and } r(x) \subseteq s(q),\ \text{ or} \\ q & \text{if } t \text{ has property } p \text{ and } r(x) \cap s(q) = \emptyset, \text{or} \\ q & \text{if } t \neq T \text{ and } t \text{ does not have property } p \end{cases}$	$\begin{cases} q' & \text{if } t=T \text{ and } s(q)+r(x)=s(q') \\ q & \text{if } t=T \text{ and } T \text{ does not have property } p \text{ and} \\ & x \in s(q),\ \text{ or} \\ q & \text{if } t \text{ has property } p \text{ and } r(x) \cap s(q) = \emptyset, \text{or} \\ q & \text{if } t \neq T \text{ and } t \text{ does not have property } p \end{cases}$

Figure 5: Transition functions.

3.5 Computing Edge Weights

Most simple OE algorithms will want to fold the weight tuple

$$w = \langle \mathit{Potency}(T), \mathit{Degradation}(T), \mathit{ObfLevel}(x), \mathit{PerfImport}(x) \rangle$$

into a single real number $\mathit{Fold}(w)$ to represent the goodness of choosing a particular edge. In our current implementation we let

$$\begin{aligned} \mathit{Fold}(w) &= \mathit{Potency}(T) \cdot \mathit{ObfLevel}(x) \cdot \\ &\quad (1 - \mathit{PerfImport}(x)) \cdot (1 - \mathit{Degradation}(T)). \end{aligned}$$

The probability of taking an edge $a \xrightarrow{(T,x),\mathit{Fold}(w)} b$ is thus proportional to how potent the transformation T is and how important it is to obfuscate application object x. It is *inversely* proportional to how performance critical x is and how much T is expected decrease its performance.

4 Algorithms

Once the probabilistic FSA has been constructed, the model is used to find an effective, yet not optimal, obfuscation sequence. Our broad definition for an optimal obfuscation sequence is a sequence such that the application has maximal obfuscation, has minimal performance degradation, and is a minimal sequence. Here, we describe a simple on-line algorithm to determine a sequence. Given the probabilistic FSA and the *Fold* formula from section 3.5, the on-line algorithm computes the obfuscation sequence and performs the associated obfuscating transformations.

The algorithm performs a random walk of the nodes of the FSA, starting in the start node. Each iteration selects an outgoing edge e from the current node S, performs the corresponding obfuscating transformation, and updates The probability of a particular outgoing edge being chosen is proportional to its weight. As the desired obfuscation level of each application object approaches zero, the weights of the edges that represent obfuscating that application object also approach zero. Edges with a zero weight are removed from the FSA. The loop terminates when there are no available edges out of the current, accepting, state. In more detail:

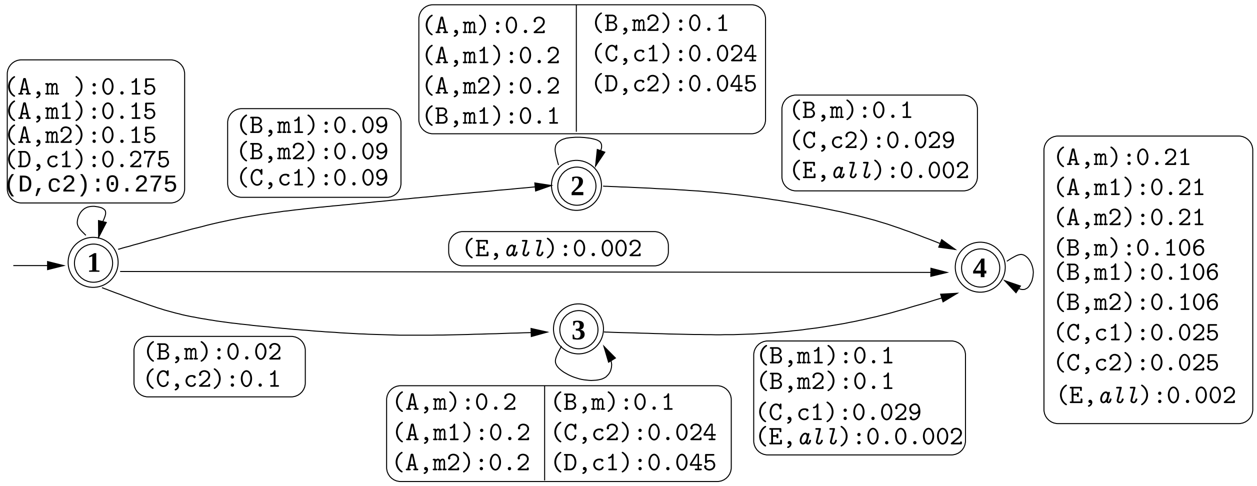


Figure 6: The FSA that models the preprohibition of property p_3 before transformation D. Each edge is labeled with a set of tuples $(T, x) : w$ where T is the obfuscation transformation to run on application object x and w is the weight of tuple (see Section 3.2). Note that for each node, the weights of the outgoing edges sum to 1, which allows the FSA model to be used as a probabilistic FSA .

1. Let S (the current state) be the start state of the FSA.
2. Let E be S's outgoing edges where each edge is $e_i = S \xrightarrow{(T_i,x_i),w_i} U_i$
3. Remove any edge that has a 0 weight. That is, let $E' = E - \{e_j | \mathit{Fold}(w_j) = 0\}$.
4. If $|E'| = 0$ and S is an accepting state, halt.
5. During each iteration the algorithm only looks at the current state and its outgoing edges to select the next state. As a result, it is possible for the loop to get stuck in a non-accepting state. Should this happen we use the weights from the previous iteration. In other words, if $|E'| = 0$ and S is an not an accepting state, set $E = E'$.
6. Compute the probability of choosing edge e_i as $\frac{\mathit{Fold}(w_i)}{\sum_{e_j \in E} \mathit{Fold}(w_j)}$.

   Randomly choose an edge, $e_c = S \xrightarrow{(T_c,x_c),w_c} U_c$ using the computed probabilities.
7. Apply obfuscation T_c to application object x_c.
8. Move to U_c, the destination state of the chosen edge.
9. Calculate the change in metrics for x_c and reduce the obfuscation level of x_c accordingly:

   $$\begin{aligned} &\mathit{ObfLevel}(T_c)_{new} = \\ &\qquad \mathit{ObfLevel}(T_c))_{old} \cdot (1 - (\Delta(x_c, M, T_c))) \end{aligned}$$

   where $\Delta(x_c, M, T_c)$ is the change in metrics described in section 3.3

This algorithm is a random walk of theFSA, using the edge weights to guide traversal. While this algorithm will not yield an optimal obfuscation sequence, it will produce a sequence that obfuscates heavily with acceptable performance degradation. Furthermore, since this algorithm produces a very random obfuscation sequence, attacks against the obfuscated code are difficult. Given a list of the obfuscations available to the loop, the attacker still does not know the subsets of obfuscation that the application objects have been obfuscated with, nor the order in which the obfuscations have been applied. We can use this loop to obfuscate a fingerprinted application each time it is sold. Each copy will become entirely different, allowing us to protect against collusive attacks.

Consider again the running example from Figure 3 and the FSA model shown in Figure 6. The weights are shown as they would be computed before the first iteration of the loop. We assume that the obfuscation level of all objects is 1 and that performance importance is 0. On the first iteration we randomly choose to move from state 1 (the start state) to state 3 by running algorithm `C` on class `c2`. This will cause `c2` to be obfuscated, lowering its remaining *ObfLevel*(`c2`). During the next iteration any edge with `c2` as its target will have a lower weight, lowering its probability to be obfuscated again. Note that moving to state 3 eliminates the possibility of ever running `D` on `c2`. This is because algorithm `D` preprohibits any algorithm with property p_3, such as `C`.

4.1 Implementation

The probabilistic FSA constructed in Section 3 provides a clean model for the dependencies between transformations. It has also allowed us to construct the simple random walk OE algorithm above. However, a straight-forward implementation of these ideas turns out to be impractical. The reason is that for sets of transformations with very few dependencies the size of the FSA grows exponentially. The solution to this problem is to lazily build and walk the FSA concurrently. The algorithm in Section 4 is unchanged except for step 2 where states that have yet to be seen are constructed. As a result, the number of FSA states that is actually generated becomes proportional to the number of obfuscation transformation actually applied.

In our implementation, *Potency*(T) is computed based on a suite of standard software complexity metrics [24]: McCabe's [20] cyclomatic complexity, design complexity, and data complexity measures, Halstead's [15] software science metric, Chidamber and Kemerer's [6] object-oriented metric suite, Harrison and Magel's [16] nesting complexity metrics, Munson's [21] data structure complexity measure, and Henry and Kafura's [17] fan in/out complexity metrics. Together these give a good estimate of the potency of obfuscations that target class hierarchies, control structures, and data structures.

To compute *Degradation*(T) we use a large suite of standard Java benchmarks including SPEC JVM98 (`www.specbench.org/osg/jvm98`) and the Ashes test suite
(`www.sable.mcgill.ca/ashes`).

Our current implementation supports three OEs: a simplistic set-based model (described in Section 4.2 below), the complete FSA-based model, and the lazy FSA model. All are on-line algorithms. We are currently exploring an off-line OE which uses the probabilistic FSA to compute an "optimal" obfuscation sequence ahead of time.

4.2 Evaluation

To evaluate the FSA-based OE algorithm we compare it to two simpler OE algorithms. Each algorithm was run on SpecJVM benchmarks with the desired obfuscation level for each object maximized and no user input about application hotspots. Figure 7 shows the result of running the three OEs. For each benchmark we show the change in code size, execution time, and software complexity metrics.

4.2.1 Random Select OE

The first OE algorithm, Random, does not split the program into obfuscation objects and instead performs obfuscating transformations to the entire program with each pass. Random applies a few heuristics to manage transformation dependencies, specifically enforcing the prerequisite, preprohibition, and postprohibition rules, but makes no effort to ensure postrequisites can be fulfilled. After determining which transformations can not be applied to the program, a transformation is chosen at random from the remaining candidates. If the executive detects that the program has been transformed into a corrupt state, it simply halts. After each obfuscation, it computes the change in complexity metrics and halts when the desired amount of obfuscation has been applied.

The Random implementation fails to find an appropriate cost-benefit trade-off for obfuscation, notably in `check`, `compress`, and `jess` where the change in metrics is low in comparison to the FSA for a large trade-off in size and speed. It is important to note that the random implementation applies each obfuscation to the entire program. The results of the Random OE shows that simply running all of the obfuscations on the program randomly does not produce an acceptable amount of obfuscation; there is a need for an intelligent OE.

4.2.2 Set-Based OE

The second OE algorithm depends on a set-based model. In this algorithm, each application object is associated with a set of obfuscation candidates, transformations that are allowed to be run on that object. In each iteration the loop uses a set of heuristics to trim the candidate set to remove algorithms that have been disqualified by running the previous obfuscation. The target application object is chosen based on the desired obfuscation level for each object and the amount of obfuscation already applied to the object. Another set of heuristics is used to extract a subset of obfuscations that can be run and in most cases will not lead to a corrupted state where required dependencies cannot be fulfilled. A candidate is then chosen at random from this subset. The set heuristics fail to detect cases such as two obfuscating transformations conflicting with each other, yielding only λ as a valid obfuscation sequence. The set-based model halts when each application object reaches the desired obfuscation level.

The set-based model fails to impact the complexity metrics over the entire program, while still incurring the cost of speed and size for the attempted obfuscations. Presumably, this is due to the fact that the set-based implementation chooses the next obfuscation target in isolation to the entire application, without considering which (transformation, object) pair will yield the most obfuscation overall.

4.2.3 FSA-Based OE

As expected, the FSA produces a large increase in software complexity, at the cost of program size and performance. The lack of user input about hotspots is most evident in the cases of extreme performance hit, like `raytrace`. It would also be useful to re-calculate profiling data between each iteration of the obfuscation loop, with the obvious performance implications.

Overall, the numbers show that there is much to be gained over random OEs. More analysis of individual obfuscations and their effect on speed, size, and complexity, using complexity measures to drive the direction of obfuscation, and integrating knowledge of the program designer will lead to a more intelligent OE.

The FSA-based model is superior to the other approaches. It elegantly handles the obfuscation dependencies, without the use of conservative restrictions to guarantee that loop will not go into a corrupted state. The FSA-based model can yield several algorithms using different analyses, whereas the set-based approach is tied to the trimming algorithm and can only be adjusted in the way that the obfuscation target and candidate are chosen. In addition, modeling obfuscation sequences as weighted members of a regular language provides a straightforward solution for deriving a *family* of optimal transformation sequences for use in artificial diversity.

5 Related Work

To the best of our knowledge, [11] is the first description of an OE. This algorithm does not take into account restrictions on transformation ordering. Wroblewski [30] describes an OE for x86 machine code. The OE applies obfuscations in a single pass over the program using a hardcoded sequence of transformations. Lacey [19] presents an algorithm which decides whether applying one optimizing transformation to a piece of code will prevent an other one from being applied.

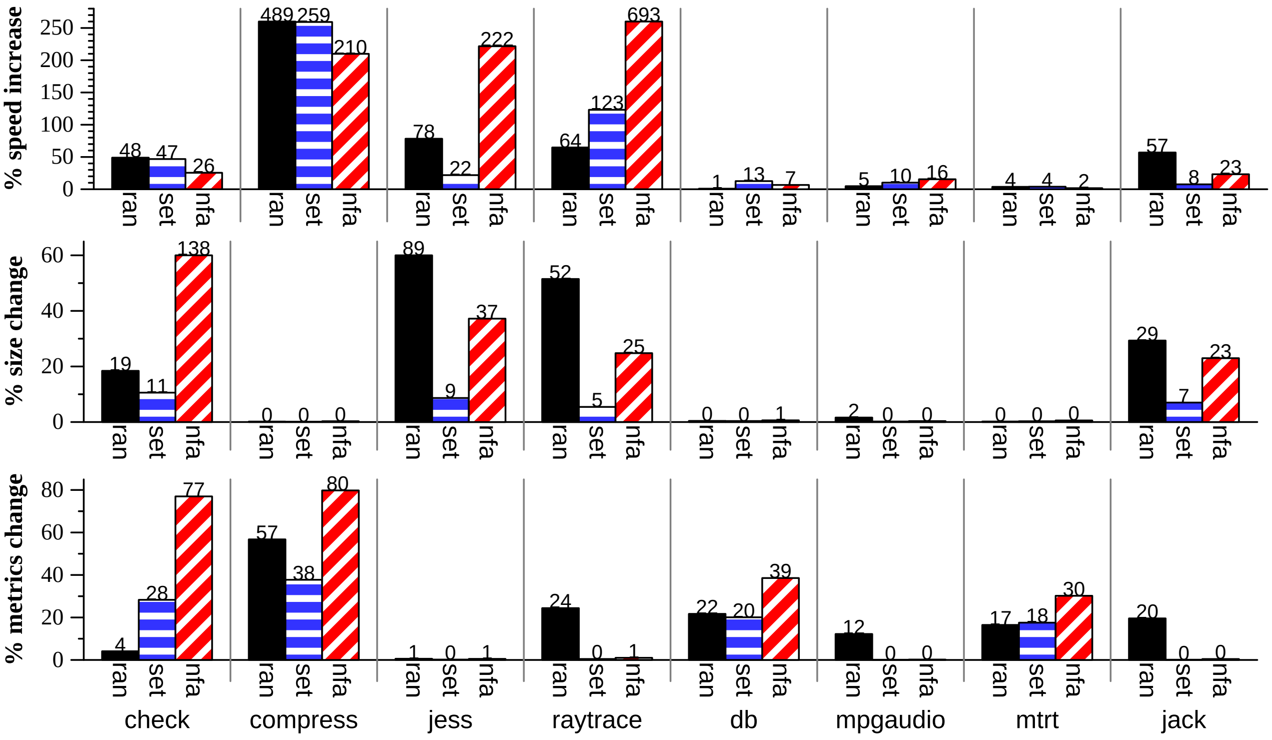


Figure 7: Results from implemented obfuscation loops on SpecJVM.

There are few theoretical results related to obfuscation. Barak [3] shows that there exist programs that cannot be obfuscated. Appel [1] shows deobfuscation to be NP-easy. The proof idea is based on a simple algorithm which nondeterministically guesses the original program S and obfuscation key K. The obfuscating transformation is then run over S and K, verifying that the result is the obfuscated program. To use this algorithm to defeat our random walk obfuscation loop, K must include the seed to our random number generator. However, Appel's algorithm is only valid for obfuscating transformations that are injective; transformations such as name obfuscation or instruction reordering cannot be reversed using Appel's method.

5.1 Phase Ordering and Termination

Determining the order in which to run obfuscation and watermarking algorithms is similar to the phase ordering problem for optimizations in compilers. In the case of optimizers, the phases consist of code analyses and optimizing transformations. While most compilers hardwire the order of the optimizations, some work has been done on automated phase selection and ordering [8,28,29].

However, these papers do not consider any requirements of the target application (such as which parts of the program are performance critical), only the algorithmic aspects of the optimizing transformations themselves. In [8], the phase ordering is generated from specifications that state properties that each algorithm needs, creates, or destroys, and which transformations are beneficial. In the case that multiple orderings are generated, a cost specification is used to decide between them.

Unlike many code optimizers which apply transformations until there are no more code changes, an OE will typically not want to obfuscate every object with every possible algorithm. Obfuscation tends to deteriorate program performance, so the optimality of the sequence of algorithms is not simply the sequence that includes all phases that change the target object. Consideration must be given to the impact on performance, program size, and parameters specified by the user. Also, an obfuscation does not transform the application to a state closer to a fixed point, so the loop must decide when to stop by means other than exhaustion.

6 Summary

There are many real-world security-related applications of code obfuscation. In some cases we may want to protect an entire application from reverse engineering and are not too concerned with any resulting performance penalty. In this case we may be happy to always apply the same sequence of obfuscations uniformly across the application. If, instead, we are using obfuscation to protect against collusive attacks against software fingerprints, we want every obfuscated application to be completely different. For this reason, every generated sequence of obfuscations should be unique. Finally, if we are protecting the integrity of a secret (such as a set of cryptographic keys) within the program, we want the part of the program that stores these secrets to be heavily obfuscated, while other, performance critical, parts should only be obfuscated lightly.

These widely differing requirements have led us to the very general probabilistic NFA model of Section 3. The model encodes all valid obfuscation sequences as well as the goodness of any particular sequence. Two on-line OE algorithms have been implemented that makes use of this model. In Section 4.2 we showed that these algorithms compare favorably to an algorithm based on a simplistic set-based OE model.

The SandMark framework in which the OE algorithms have been implemented can be downloaded from `http://sandmark.cs.arizona.edu`. SandMark consists of 130,000 lines of Java, and includes 42 obfuscation and 17 watermarking algorithms and several tools for automatically and manually analyzing and attacking software protection algorithms.

References

[1] A. Appel. Deobfuscation is in np. `www.cs.princeton.edu/~appel/papers/deobfus.pdf`.

[2] D. Aucsmith. Tamper resistant software: An implementation. In *Information Hiding, First International Workshop*, pages 317–333, May 1996. LNCS 1174.

[3] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of software obfuscation. In *Crypto01*, pages 1–18, 2001. LNCS 2139.

[4] H. Chang and M. Atallah. Protecting software code by guards. In *Workshop on Security and Privacy in Digital Rights Management*, 2002.

[5] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Carnegie-Mellon University, 2002.

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

[7] F. B. Cohen. Operating system protection through program evolution. `all.net/books/IP/evolve.html`, 1992.

[8] W. E. Cohen. *Automatic Construction of Optimizing, Parallelizing Compilers From Specifications*. PhD thesis, Purdue University, 1994.

[9] C. Collberg, G. Myles, and A. Huntwork. Sandmark - a tool for software protection research. *IEEE Security and Privacy*, 1(4):40–49, 2003.

[10] C. Collberg and C. Thomborson. Software watermarking: Models and dynamic embeddings. In *POPL'99*, San Antonio, TX, Jan. 1999.

[11] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

[12] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *ICCL'98*, Chicago, IL, May 1998.

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

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

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

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

[17] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, Sept. 1981.

[18] F. Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In *Mobile Agents and Security*, pages 92–113. Springer-Verlag, 1998. LNCS 1419.

[19] D. Lacey and O. de Moor. Detecting disabling interference between program transformations. `citeseer.nj.nec.com/464977.html`.

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

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

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

[23] T. A. Proebsting and S. A. Watterson. Krakatoa: Decompilation in Java (Does bytecode reveal source?). In *COOTS*, June 1997.

[24] S. Purao and V. Vaishnavi. Product metrics for object-oriented systems. *ACM Comput. Surv.*, 35(2):191–221, 2003.

[25] J. P. Stern, G. Hachez, F. Koeune, and J.-J. Quisquater. Robust object watermarking: Application to code. In *Information Hiding*, pages 368–378, 1999.

[26] P. Tyma. Method of reducing the number of instructions in a program code sequence. US patent 5,903,761, 1999.

[27] H. P. V. Vliet. Crema — The Java obfuscator. `web.inter.nl.net/users/H.P.van.Vliet/crema.html`, Jan. 1996.

[28] D. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. In *PPOPP'90*, pages 137–146, 1990.

[29] D. Whitfield and M. L. Soffa. Automatic generation of global optimizers. In *PLDI'91*, pages 120–129, 1991.

[30] G. Wroblewski. *A General Method of Program Code Obfuscation*. PhD thesis, Wroclaw University, 2002.