# Deobfuscation
## Reverse Engineering Obfuscated Code

Sharath K. Udupa,* Saumya K. Debray,* Matias Madou†


## Abstract

In recent years, code obfuscation has attracted attention as a low cost approach to improving software security by making it difficult for attackers to understand the inner workings of proprietary software systems. This paper examines techniques for automatic deobfuscation of obfuscated programs, as a step towards reverse engineering such programs. Our results indicate that much of the effects of code obfuscation, designed to increase the difficulty of static analyses, can be defeated using simple combinations of straightforward static and dynamic analyses. Our results have applications to both software engineering and software security. In the context of software engineering, we show how dynamic analyses can be used to enhance reverse engineering, even for code that has been designed to be difficult to reverse engineer. For software security, our results serve as an attack model for code obfuscators, and can help with the development of obfuscation techniques that are more resilient to straightforward reverse engineering.


## 1 Introduction

In recent years, code obfuscation has attracted some attention as a low cost approach to improving software security [5, 8, 9, 21, 22, 29]. The goal of code obfuscation is to make it difficult for an attacker to reverse engineer programs. The idea is to prevent an attacker from understanding the inner workings of a program by making the obfuscated program "too difficult" to understand—that is, by making the task of reverse engineering the program "too expensive" in terms of the resources or time required to do so. Obfuscation has also been used to protect "software watermarks" and fingerprints, which are designed to thwart software piracy [2, 8, 9]. The presumption is that making it difficult for attackers to understand the internal workings of a program prevents them from discovering vulnerabilities in the code, and serves to protect the program owner's intellectual property.

It is important to note, however, that code obfuscation is merely a technique. Just as it can be used to protect software against attackers, so too it can be used to hide malicious content. For example, certain kinds of sophisticated computer viruses, e.g., polymorphic viruses, have resorted to using obfuscation techniques to prevent detection by virus scanners [27].

This raises two closely related questions. The first question, from a software engineering pespective, is: *What sorts of techniques are useful for understanding obfuscated code?* For example, suppose we have downloaded, from a web site, a file purporting to be a security patch for some application. Before applying the patch, we may want to verify that the file does not contain any malicious payload. How can we verify this if the contents of the file have been obfuscated? The second question, from a security perspective, is: *what are the weaknesses of current code obfuscation techniques, and how can we address them?* If our obfuscation schemes are ineffective in thwarting attackers from reverse engineering the code, then they are not only useless, but are in fact worse than useless: they increase the time and space requirements of the program, and can contribute to a false sense of security that keeps other security measures from being deployed. Thus, identifying any weaknesses in current obfuscation schemes by developing and testing attack models can lead to better obfuscation schemes and concomitant improvements in software security.

This paper aims to address the questions raised above, regarding techniques for understanding obfuscated code and the strengths and weaknesses of sophisticated obfuscation algorithms. We describe a suite of code transformations and program analyses that can be used to identify and remove obfuscation code and thereby help reverse engineer obfuscated programs. We use these techniques to examine the resilience of the *control flow flattening* obfuscation technique, which has been proposed in the research literature [3, 29] and used in a commercial code obfuscation product by Cloakware [6], against attacks based on combinations of static and dynamic analyses. Our results indicate that from the perspective of reverse engineering, simple dynamic techniques can often be very useful in coping with code obfuscation. From a software security perspective, we show that many obfuscation techniques can be largely neutralized using combinations of simple and well known static and dynamic analyses.

*Department of Computer Science, The University of Arizona, Tucson, AZ 85721, USA. Email: {sku, debray}@cs.arizona.edu
Supported in part by the National Science Foundation under grants CNS-0410918 and CCR-0113633.

†Ghent University, St.-Pietersnieuwstraat 41, B-9000 Ghent, Belgium. Email: mmadou@elis.ugent.be
This research was funded by Ghent University, by the Fund for Scientific Research-Flanders (FWO-Flanders).

```
int f(int i, int j)
{
  int a = 1;
  if (i < j) {
    a = j;
  }
  else
    do {
      a *= i--;
    } while (i > 0);

  return a;
}
```

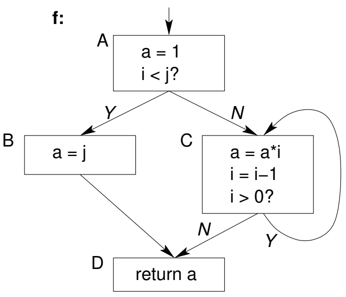



Figure 1: An example program and its control flow graph

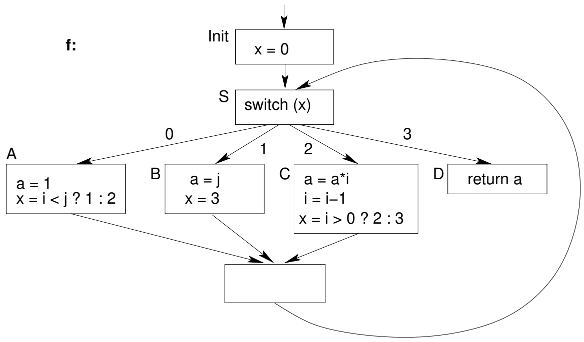



Figure 2: Control flow graph after basic flattening

## 2 Obfuscating Transformations

Conceptually, we can distinguish between two broad classes of obfuscating transformations. The first, *surface obfuscation*, focuses on obfuscating the concrete syntax of the program. An example of this is changing variable names or renaming different variables in different scopes to the same identifier , as carried out by the "Dotfuscator" tool for obfuscating .NET code [25]. The second, *deep obfuscation*, attempts to obfuscate the actual structure of the program, e.g., by changing its control flow or data reference behavior [5, 9]. While the former may make it harder for a human to understand the source code, it does nothing to disguise the semantic structure of the program. It therefore has no effect on algorithms used for reverse engineering, such as program slicing, that rely on code structure and semantics rather than the concrete syntax. For example, it is straightforward to undo most of the effects of Dotfuscator-style variable renaming simply by using a parser to resolve variable references using the scope rules of the language and rename variables accordingly. Deep obfuscation, by contrast, changes the actual structure of the program, and therefore affects the efficacy of semantic tools for program analyses and reverse engineering. Space constraints preclude a more detailed elaboration of different kinds of deep obfuscation techniques, but the interested reader is referred to a discussion and more detailed taxonomy by Collberg *et al.* [10]. For the purposes of this paper, it suffices to note that working around deep obfuscation—which re-

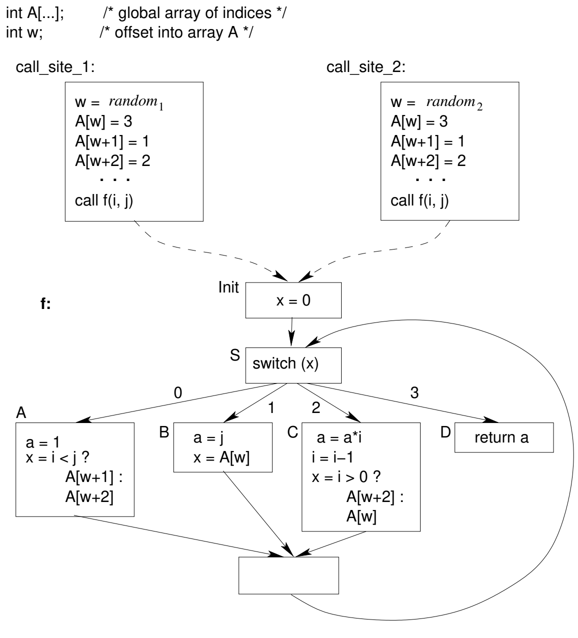



Figure 3: Enhancing flattening with Interprocedural Data Flow

quires reasoning about semantic aspects of the program—is intuitively more difficult than working around surface obfuscation, which is essentially a syntactic issue. This paper is concerned primarily with deep obfuscation techniques that attempt to disguise the control flow logic of a program.

In prior work, we considered the problem of deobfuscating programs that had been subjected to a number of control flow obfuscations based on opaque predicates; we found that for the obfuscations considered (a set of control flow obfuscations implemented in Collbergs' *Sandmark* obfuscation tool for Java programs [7]), most of the obfuscation could be removed using a combination of fairly straightforward static and dynamic analyses [4]. This paper considers a different approach to control flow obfuscation, taken from Chenxi Wang's dissertation [29, 30]. This choice is motivated by three factors. First, based on our experiments, it seems more difficult to break than those we had considered earlier [4]. Second, this approach has been considered by other researchers as well [3], and its resilience is therefore of interest to the research community. Finally, it is a key component of an industrial obfuscation tool by Cloakware Inc. [6].

This section describes the basic control flow obfuscation technique as well as two variants that aim to make the basic approach harder to break.

### 2.1 Basic Control Flow Flattening

Control flow flattening aims to obscure the control flow logic of a program by "flattening" the control flow graph so that all basic blocks appear to have the same set of predecessors and successors. The actual control flow during execution is guided by a *dispatcher variable*. At runtime,

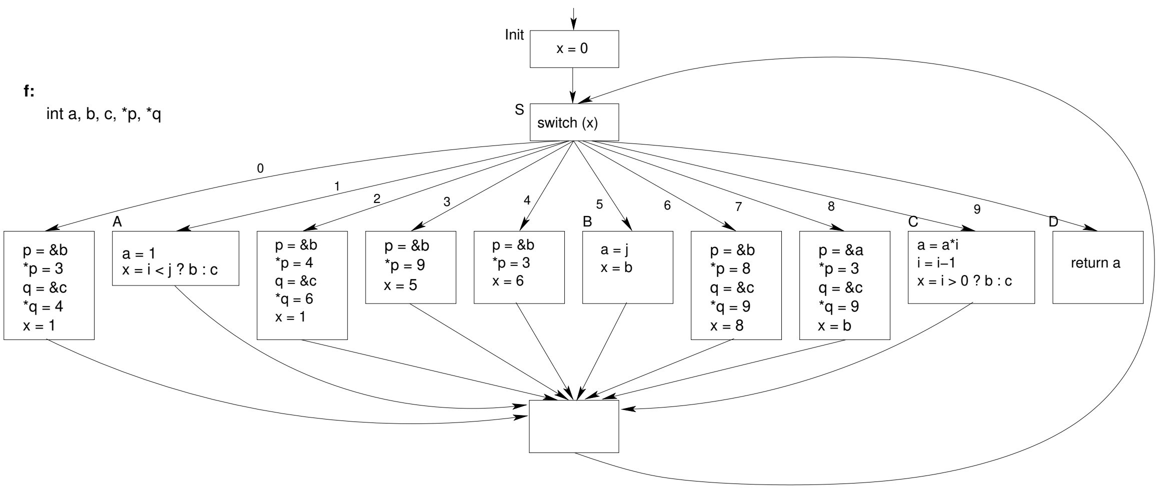



Figure 4: Enhancing flattening with artificial blocks and pointers

each basic block assigns to this dispatcher variable a value indicating which next basic block should be executed next. A *switch* block then uses the dispatcher variable to jump indirectly, through a jump table, to the intended control flow successor.

As an example, consider the program shown in Figure 1. Basic control flow flattening of this program results in the control flow graph shown in Figure 2, where S is the switch block and x the dispatcher variable.[1] The initial assignment to the dispatcher variable x in the block Init is intended to route control to A, the original entry block of f(), when control first enters the function; after this, control flow is guided by assignments to x in the various basic blocks.

## 2.2 Enhancing Flattening I: Interprocedural Data Flow

In the basic control flow flattening transformation discussed in Section 2.1, the values assigned to the dispatch variable are available within the function itself. Because of this, while the control flow behavior of the obfuscated code is not obvious, it can be reconstructed by examining the constants being assigned to the dispatch variable. This, in turn, requires only intra-procedural analysis.

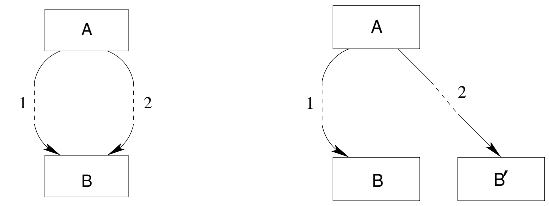
The resilience of the obfuscation technique can be improved using interprocedural information passing. The idea is to use a global array to pass the dispatch variable values. At each call site to the function, these values are written into the global array starting at some random offset within the array (appropriately adjusted to avoid buffer overflows). The offset so chosen may be different at different call sites for the function, and is passed to the obfuscated callee either as a global or as an argument. The obfuscated code then assigns values to the dispatch variable from the global array. Neither the actual locations accessed, nor the contents of these locations, are constant values, and are not evident by examining the obfuscated code of the callee. The code resulting from applying this obfuscation to the program in Figure 1 is illustrated in Figure 3.

## 2.3 Enhancing Flattening II: Artificial Blocks and Pointers

The obfuscation technique detailed above can be extended by adding artificial basic blocks to the control flow graph. Some of these artificial blocks are never be executed, but this is difficult to determine by a static examination of the program because of the dynamically computed indirect branch targets in the obfuscated code. We then add indirect loads and stores, through pointers, into these unreachable blocks. These have the effect of confusing static analyses about the possible values taken on by the dispatch variable. Figure 4 shows the result of applying this to the program of Figure 1.

In our implementation, we add two artificial basic blocks corresponding to each block in the original function: one of these blocks is actually executed at runtime, while the other is simply a decoy added to mislead static analysis. Given a block $B$ in the original program, let the corresponding artificial block that gets executed be denoted by $B'$, and the decoy artificial block be $B''$. Indirect assignments through pointers are added to both these artificial blocks. However, only the assignments in the block $B'$ set the dispatch variable to the appropriate values so as to give the right control flow during execution; the decoy block $B''$, by contrast, sets the dispatch variable to other values, so as to give a misleading picture of control flow. In the original block $B$, the

[1] Strictly speaking, Figure 2 is slightly inaccurate in that it shows that the control flow from basic blocks A, B, and C come together into a single block, at the bottom of the picture, from which it then branches to the top of the switch block S. In practice, control would go directly from each of A, B, and C directly to the top of S. We draw it as shown to reduce the clutter of control flow edges and bring out the essential logic underlyig the transformation. This becomes especially important when we consider enhancements to the basic transformation, as illustrated in Figs. 3 and 4.




(a) Original code (b) After cloning

Figure 5: Code Cloning

value of the dispatch variable that gets loaded is that previously assigned in the artificial block $B'$. Hiding the starting value of the switch variable makes it harder for a static analyzer to deduce which blocks are executed and hence find out the valid definitions of the switch variable.

## 3 Deobfuscation

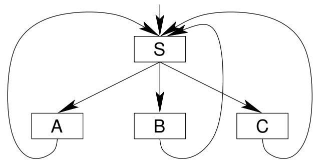
This section describes a number of analyses and program transformations that we have found useful for reverse engineering obfuscated code.

### 3.1 Cloning

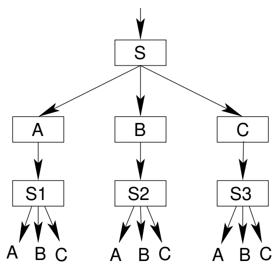
Many obfuscation techniques rely on introducing spurious execution paths into the program to thwart static program analyses [5, 9]. These paths that can never be taken at runtime, but cause bogus information to be propagated along them during program analyses, thereby reducing the precision of information so obtained and making it harder to understand the program logic. This is illustrated in Figure 5(a), where information is propagated between basic blocks A and B along the "actual" control flow path 1 as well as the spurious control flow path 2, the latter having been introduced by the obfuscator. The bogus data flow information propagated along 2 then has the effect of introducing imprecision in the results of program analyses at points where execution paths come together. In Figure 5(a), the results of forward dataflow analyses, such as reaching definitions, are tainted at the entry to $B$, while those of backward analyses, such as liveness analyses, are affected at the exit from $A$.

One way to address this problem is to clone portions of the program in such a way that the spurious execution paths no longer join the original execution paths and taint the information obtained from analysis. The result of applying cloning to basic block $B$ in Figure 5(a) is shown in Figure 5(b). In this case, this results in improved forward dataflow information available at the entry to $B$. In this example, however, cloning does not eliminate the spurious control flow edge $A \to B'$, and so does not improve the backward dataflow information available at the exit from $A$.

This transformation obviously has to be applied judiciously, since otherwise it can cause large increases in code size and further exacerbate the reverse engineering problem. Moreover, since the goal of deobfuscation is to try to identify and remove obfuscation code, this means that in general, cloning has to be applied without knowing, ahead




(a) Original (obfuscated) code




(b) Obfuscated code after cloning

Figure 6: Code Cloning for Control Flow Flattening

of time, which execution paths are spurious and which are not. One possible approach, in such situations, would be to apply cloning selectively at points where multiple control flow paths join, and where the dataflow information propagated along some paths is significantly less precise than that propagated along others. Alternatively, if we know something about the kind of obfuscation that has been applied, it may be possible to apply cloning in a way that exploits this information. For example, it is relatively straightforward to infer that control flow flattening has been applied, because of the distinctive control flow graphs it produces.

For the purposes of this paper, we use cloning in the context of one of our deobfuscator implementations (see Section 4.1), as illustrated in Figure 6. Consider the obfuscated program fragment shown in Figure 6(a), where the basic blocks A, B, and C all transfer control to a *switch*-block S. Cloning creates three copies S1, S2, and S3 of the *switch*-block S, corresponding to the successors A, B, and C respectively. The control flow successors of each of these copies is the set of control flow successors of the original *switch*-block, i.e., each of the copies S1, S2, and S3 has an edge to each of the blocks A, B, and C. In the resulting program, shown in Figure 6(b), the dataflow information entering the *switch*-block S1 is not commingled with that entering the *switch*-block S2 from B or that entering the *switch*-block S3 from C.

### 3.2 Static Path Feasibility Analysis

We use the term *static path feasibility analysis* to refer to constraint-based static analyses to determine whether an (acyclic) execution path is feasible. Given an acyclic execution path $\pi$ with $\bar{x}$ the set of variables live at entry to $\pi$, the idea is to construct a constraint $C_\pi$ such that the logical formula $(\exists\bar{x})C_\pi$ is unsatisfiable only if, for all possible executions of the program, $\pi$ is never executed. $C_\pi$ is thus a conservative approximation to the effects of the execution of the instructions along $\pi$. If $(\exists\bar{x})C_\pi$ can be shown to be unsatisfiable, we can conclude that $\pi$ is unfeasible.

In principle, we can imagine many different ways to construct the constraint $C_\pi$ corresponding to a path $\pi$. For the purposes of this paper, our goal is to take into account the effects of arithmetic operations on the values of variables, effectively obtaining an analysis that resembles constant propagation, but propagates information along a single execution path rather than along all execution paths. To this end, we use linear arithmetic constraints to reason about variable values. The discussion below assumes a low-level program representation, e.g., as three-address code, RTL, or even machine instructions.

Assume that each instruction in the program has a unique name $I_k$. The value of a variable $x$ at the beginning of $\pi$ is denoted by $x_0$, while at intermediate points along the path, the value of $x$ immediately after instruction $I_k$ is denoted by $x_k$. An unknown value is denoted by $\perp$. The constraint $C_\pi$ is constructed as a conjunction of a Constraint $C_k$ for each instruction $I_k$ in $\pi$, as follows:

1. *Assignment*: $I_k \equiv$ '$x = y$'. Then, $C_k \equiv x_k = y_j$, where $I_j$ refers to the most recent instruction in $\pi$ that defined $y$ ($j = 0$ if there is no definition of $y$ in $\pi$ before $I_k$).

2. *Arithmetic*: $I_k \equiv$ '$x = y \oplus z$' for some operation $\oplus$, where $I_i$ and $I_j$ refer to the most recent instructions defining $y$ and $z$ respectively ($i = 0$ if $y$ has not yet been defined along $\pi$, and similarly with $j$). Then, $C_k \equiv x_k = f_\oplus(y_i, z_j)$. Here, $f_\oplus$ expresses the semantics of the operation $\oplus$. If the semantics of $\oplus$ is not known to the analyzer, or if either $y_i = \perp$ or $z_j = \perp$, then $C_k \equiv x = \perp$.

3. *Indirection*: Pointers can be modelled at different levels of precision, with a concomitant tradeoff in analysis speed [15]. A full discussion of pointer analysis is beyond the scope of this paper; we require only that the treatment of pointers be conservative, i.e., that the set of possible targets for a pointer during analysis be a superset of the actual set of targets during any execution.

4. *Branches*: $I_k \equiv$ '**if** $e$ **goto** $L$' for some Boolean expression $e$. In this case,

$$C_k \equiv \begin{cases} e & \text{if } I_k \text{ is a taken branch in } \pi; \\ \neg e & \text{if } I_k \text{ is not taken in } \pi; \end{cases}$$

   Unconditional branches can be treated as a special case where $e \equiv$ *true*, while multi-way branches such as those arising from *switch* statements, can be modelled as a semantically equivalent series of conditional branches.

5. Otherwise, the effects of instruction $I_k$ cannot be modelled by the analyzer. The analysis is aborted in this case, and our system conservatively assumes that $\pi$ is a feasible path.

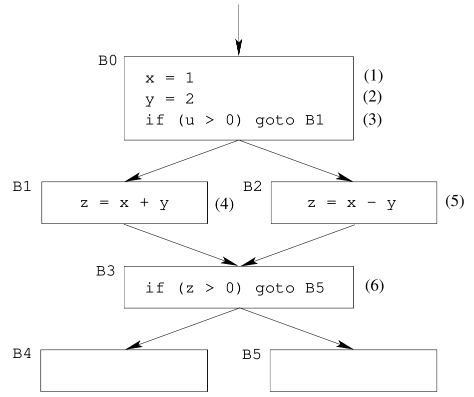



Figure 7: An example of static path feasibility analysis

Once the constraint $C_\pi$ has been constructed in this way, a constraint solver is used to determine its satisfiability. Our current implementation uses the Omega calculator [23] to test satisfiability.

Figure 7 illustrates the use of constraints for static path feasibility analysis. The parenthetical figures to the right of each basic block serve to identify different instructions. Consider the path $\pi$ = `B0`$\rightarrow$`B2`$\rightarrow$`B3`$\rightarrow$ `B5`. The only relevant live variable at the entry to this path is `u`. The corresponding constraint $C_\pi$ is therefore:

$$(\exists u_0)[x_1 = 1 \land y_2 = 2 \land u_0 > 0 \land z_5 = x_1 - y_2 \land z_5 > 0].$$

It is not difficult to see that this constraint is unsatisfiable, which means that the path $\pi$ is unfeasible. Note that conventional constant propagation would obtain `z` = $\perp$ at entry to block `B3`, and thereby conclude that the path $\pi$ is feasible.

Note that this example could also have been handled by cloning block `B3`, which would have the effect of preventing the loss of information resulting from the control flow join of edges `B1`$\rightarrow$`B3` and `B2`$\rightarrow$`B3`, after which constant propagation would give the expected results. Thus, path feasibility analysis and cloning can be seen as complementary techniques.

### 3.3 Combining Static and Dynamic Analyses

Conventional static analyses, such as that of Section 3.2, are inherently conservative,[2] so the set of edges resulting from purely static deobfuscation techniques are, in general, a superset of the actual set of edges. Conversely, dynamic analyses, such as program tracing or edge profiling, cannot take into account all the possible input values to a program,

[2] This follows from soundness considerations, which cause static analyses to propagate information along a superset of the execution paths that may actually be taken by a program during execution. This observation need not hold if soundness is sacrificed, as with some recently-proposed analyses [13, 14].

and therefore are able to observe only a subset of all its possible execution paths.

The dual natures of these two approaches to program analysis suggests that we try to combine them. This can be done in two ways. We can begin with an underapproximation to the set of control flow edges obtained via dynamic analysis, then use static analysis to add back some control flow edges that could be taken. Alternatively, we can begin with an overapproximation to the set of control flow edges edges obtained via static analysis, then use dynamic analysis to remove some control flow edges (or paths) that are not actually taken. In either case, the result may contain either more or less edges than the original program, i.e., when we combine static and dynamic analyses the result cannot be guaranteed to be either sound or precise. Nevertheless, from the perspective of reverse engineering and program understanding, such combined analyses can be very useful for overcoming the limitations of purely static and purely dynamic analyses.

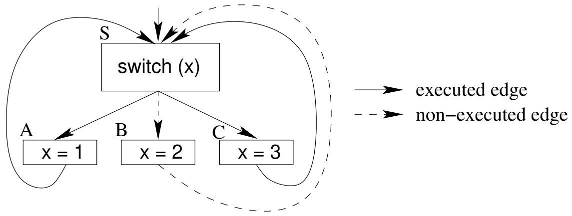
For the work described in this paper, we used a static analysis to improve the results of dynamic analysis by adding back some control flow edges that could possibly be taken. The essential idea behind our approach is based on the following *gedankenexperiment*: suppose we know, somehow, which control flow edges can actually be taken during execution. Then, we can simply mark these edges and propagate dataflow information only along such marked edges, thereby avoiding the imprecision resulting from propagating information along edges that can never be taken at runtime. Conventional static analyses can then be thought of as the degenerate case where all edges are marked. We can improve on this situation by using dynamic analyses to identify edges that are actually taken during execution and marking only these edges, then propagating dataflow information along these marked edges, as follows:

1. Initially mark only those edges that are identified as taken by the dynamic analysis.

2. Carry out constant propagation on the program, propagating information only along marked edges.

   If a conditional branch is encountered where only one the outgoing control flow edges is taken during execution, but where the outcome of the branch cannot be uniquely determined from the constant propagation, add the branch that is not taken during execution into the set of control flow edges that can be taken, and mark it.

In our implementation, the effect of this approach is to prune the dataflow information propagated into *switch* blocks. As an example, consider the following control flow fragment, where solid arrows represent control flow edges that are taken during execution, while dashed arrows correspond to edges that are never taken:




In this example, basic block B is never executed, so the control flow edges S→B and B→S are not marked and have no information propagated along them. The assignment '`x=2`' in block B is therefore not considered for static analysis; this results in the value 2 not being considered to be a possible value for the variable `x` at the *switch*.

## 4 Experimental Evaluation

We evaluated our ideas using two different binary rewriting systems for the Intel x86 platform: PLTO [24] and DIABLO [11]. We implemented three control flow flattening obfuscations described in Wang's dissertation and discussed in Section 2 in these tools, and used these to obfuscate ten programs from the SPECint-2000 benchmark suite. While these programs happen to be written in C, our experiments were carried out on program binaries.

Each of our benchmarks was compiled using *gcc* version 3.2.2, at optimization level `-O3`, with additional command-line flags to produce statically linked relocatable binaries, and the resulting binaries processed using the obfuscators mentioned above. Functions containing (indirect jumps resulting from) `switch` statements were not obfuscated because our obfuscators currently are not able to process the resulting control flow. Library functions were also excluded, because in most cases such functions contain non-standard control flow, e.g., where control jumps from one function into another without using the normal call/return mechanism for inter-procedural control transfers. Static characteristics of these benchmarks are shown in Table 1, which compares the original programs with those resulting from basic control flow flattening.[3] Overall, Table 1 shows that our tools obfuscate most user functions in the program (on average, about 88%). As expected, obfuscation causes the number of control flow edges to increase, though the scale of the increase—a factor of roughly 55× to 60×—is larger than we had expected.

Control flow deobfuscation involves deleting spurious control flow edges that have been added by the obfuscator. To evaluate the efficacy of various deobfuscation techniques, therefore, we compare the deobfuscated program $P_{deobf}$ with the original program $P_{orig}$ to classify any errors made by the deobfuscator in deleting control flow edges. In principle, there are two kinds of such errors that can occur: first, $P_{deobf}$ may contain some edge that does not appear in

[3] The differences in the number of functions, basic blocks, and edges reported by PLTO and DIABLO arise partly because they linked in different versions of the standard C library, and partly due to some differences in code transformations carried out by the two tools, e.g., DIABLO carries out some tail-call optimization before obfuscation.

| *Program* | *Original* | | *Obfuscated* | | *Effects of Obfuscation* | |
|---|---|---|---|---|---|---|
| | Functions ($F_{orig}$) | Edges ($E_{orig}$) | Functions ($F_{obf}$) | Edges ($E_{obf}$) | $F_{obf}/F_{orig}$ | $E_{obf}/E_{orig}$ |
| *bzip2* | 42 | 2,655 | 30 | 157,192 | 0.714 | 59.21 |
| *crafty* | 104 | 12,172 | 89 | 4,309,502 | 0.855 | 352.05 |
| *gap* | 825 | 43,079 | 768 | 1,973,980 | 0.930 | 45.82 |
| *gcc* | 1,792 | 99,516 | 1,398 | 8,816,058 | 0.780 | 88.59 |
| *gzip* | 73 | 2,916 | 59 | 107,882 | 0.808 | 37.00 |
| *mcf* | 19 | 799 | 19 | 16,756 | 1.000 | 20.97 |
| *parser* | 180 | 12,299 | 174 | 684,904 | 0.966 | 55.69 |
| *twolf* | 165 | 14,799 | 157 | 1,277,410 | 0.951 | 86.32 |
| *vortex* | 638 | 39,229 | 615 | 1,969,734 | 0.963 | 50.21 |
| *vpr* | 252 | 8,948 | 211 | 310,210 | 0.837 | 34.67 |
| GEOM. MEAN: | | | | | 0.876 | 59.43 |

(a) PLTO

| *Program* | *Original* | | *Obfuscated* | | *Effects of Obfuscation* | |
|---|---|---|---|---|---|---|
| | Functions ($F_{orig}$) | Edges ($E_{orig}$) | Functions ($F_{obf}$) | Edges ($E_{obf}$) | $F_{obf}/F_{orig}$ | $E_{obf}/E_{orig}$ |
| *bzip2* | 35 | 2,167 | 34 | 168,032 | 0.971 | 77.54 |
| *crafty* | 102 | 11,853 | 86 | 2,701,600 | 0.843 | 227.92 |
| *gap* | 809 | 44,431 | 738 | 2,963,737 | 0.912 | 66.70 |
| *gcc* | 1,071 | 80,168 | 685 | 1,801,553 | 0.639 | 22.47 |
| *gzip* | 44 | 1,871 | 35 | 99,486 | 0.795 | 53.17 |
| *mcf* | 18 | 605 | 18 | 16,908 | 1.000 | 27.97 |
| *parser* | 185 | 10,301 | 174 | 714,223 | 0.940 | 69.34 |
| *twolf* | 165 | 12,772 | 156 | 1,553,117 | 0.945 | 121.60 |
| *vortex* | 620 | 32,048 | 599 | 1,298,439 | 0.966 | 40.52 |
| *vpr* | 103 | 2,305 | 84 | 44,288 | 0.815 | 19.21 |
| GEOM. MEAN: | | | | | 0.876 | 55.1 |

(b) DIABLO

Table 1: Static characteristics of original and obfuscated benchmark programs

$P_{orig}$; and second, $P_{deobf}$ may not contain some edge that appears in $P_{orig}$. We term the first kind of error *overestimation errors* (written $\Delta_{over}$), and the second kind of errors *underestimation errors* (written $\Delta_{under}$):

$$\begin{aligned} \Delta_{over} &= |\{e \mid e \in P_{deobf} \text{ and } e \notin P_{orig}\}| \\ \Delta_{under} &= |\{e \mid e \notin P_{deobf} \text{ and } e \in P_{orig}\}| \end{aligned}$$

Since the input to the deobfuscator is the obfuscated program, we express the overestimation and underestimation errors relative to the number of edges in the input obfuscated program.

### 4.1 Basic Flattening

We first consider programs obfuscated using the basic control flow flattening technique described in Section 2.1. This turns out to be straightforward to deobfuscate using purely static techniques. We considered two different approaches: the DIABLO implementation used cloning (Section 3.1) followed by conventional constant propagation to disambiguate control flow; the PLTO implementation used Constraint-based Path Feasibility Analysis (Section 3.2).

The results of deobfuscation are shown in Table 2(a). For each of our implementations, we consider two metrics: the *extent of deobfuscation*, i.e., the number of obfuscation edges that we were able to remove via the deobfuscation process; and *precision*, which gives the number of overestimated and underestimated edges, as discussed above. It can be seen that the PLTO implementation, using constraint-based path feasibility analysis, is able to recover the original programs completely, without any error. The DIABLO implementation, which uses code cloning followed by constant propagation, is able to remove over 99% of the obfuscation edges. The resulting programs still have a small amount of overestimation errors (0.72% on average), due to edges that did not appear in the original programs. This is to a great extent an artifact of the program transformation used: the cloning process introduces a number of additional control flow edges into the program, and these are not all eliminated by the constant propagation. It turns out that most of them could be eliminated quite easily by an additional phase of liveness analysis and jump-chain collapsing (i.e., where a jump to a jump is replaced by a single

| Program | PLTO Added | PLTO Removed | PLTO % Over | PLTO % Under | DIABLO Added | DIABLO Removed | DIABLO % Over | DIABLO % Under |
|---|---|---|---|---|---|---|---|---|
| *bzip2* | 154,537 | 154,537 | 0.00 | 0.00 | 165,865 | 164,657 | 0.73 | 0.00 |
| *crafty* | 4,297,330 | 4,297,330 | 0.00 | 0.00 | 2,689,747 | 2,685,374 | 0.16 | 0.00 |
| *gap* | 1,930,901 | 1,930,901 | 0.00 | 0.00 | 2,919,306 | 2,900,564 | 0.64 | 0.00 |
| *gcc* | 8,716,542 | 8,716,542 | 0.00 | 0.00 | 1,801,553 | 90,893 | 0.60 | 0.00 |
| *gzip* | 104,996 | 104,996 | 0.00 | 0.00 | 97,615 | 96,821 | 0.81 | 0.00 |
| *mcf* | 15,957 | 15,957 | 0.00 | 0.00 | 16,303 | 15,944 | 2.20 | 0.00 |
| *parser* | 672,605 | 672,605 | 0.00 | 0.00 | 703,922 | 698,700 | 0.74 | 0.00 |
| *twolf* | 1,262,611 | 1,262,611 | 0.00 | 0.00 | 1,540,345 | 1,533,774 | 0.43 | 0.00 |
| *vortex* | 1,930,505 | 1,930,505 | 0.00 | 0.00 | 1266,391 | 1,255,663 | 0.85 | 0.00 |
| *vpr* | 301,262 | 301,262 | 0.00 | 0.00 | 41,983 | 41,226 | 1.80 | 0.00 |
| GEOM. MEAN: | | | 0.00 | 0.00 | | | 0.72 | 0.00 |

(a) Basic Flattening

| *Program* | Added | Removed | % Over | % Under |
|---|---|---|---|---|
| *bzip2* | 154,537 | 116,896 | 23.95 | 0.00 |
| *crafty* | 4,297,330 | 3,051,105 | 28.92 | 0.00 |
| *gap* | 1,930,901 | 1,177,850 | 38.15 | 0.00 |
| *gcc* | 8,716,542 | 4,936,993 | 42.87 | 0.00 |
| *gzip* | 104,996 | 74,111 | 28.63 | 0.00 |
| *mcf* | 15,957 | 15,198 | 4.50 | 0.00 |
| *parser* | 672,605 | 464,098 | 30.44 | 0.00 |
| *twolf* | 1,262,611 | 820,698 | 34.59 | 0.00 |
| *vortex* | 1,930,505 | 1,351,354 | 29.40 | 0.00 |
| *vpr* | 301,262 | 165,695 | 43.70 | 0.00 |
| GEOM. MEAN: | | | 26.89 | 0.00 |

(b) Flattening with Interprocedural Data Flow

| *Program* | Added | Removed | % Over | % Under |
|---|---|---|---|---|
| *bzip2* | 165639 | 130743 | 21.76 | 0.56 |
| *crafty* | 4403750 | 3169697 | 28.21 | 0.01 |
| *gap* | 2365955 | 1655983 | 31.23 | 0.03 |
| *gcc* | 9609646 | 5830097 | 39.94 | 0.01 |
| *gzip* | 125508 | 97539 | 23.69 | 0.36 |
| *mcf* | 22335 | 22375 | 1.60 | 1.69 |
| *parser* | 786423 | 590215 | 26.09 | 0.02 |
| *twolf* | 1401063 | 973949 | 31.18 | 0.03 |
| *vortex* | 2275709 | 1735787 | 25.00 | 0.02 |
| *vpr* | 386508 | 259889 | 34.21 | 0.08 |
| GEOM. MEAN: | | | 21.40 | 0.06 |

(c) Flattening with Artificial Blocks and Pointers

**Key**:

Added: Number of edges added due to obfuscation $= E_{obf} - E_{orig}$ (see Table 1).
Removed: Number of edges removed by the deobfuscator.
% Over: Overestimation error relative to number of edges in obfuscated program $= \Delta_{over}/E_{obf}$.
% Under: Underestimation error relative to number of edges in obfuscated program $= \Delta_{under}/E_{obf}$.
$\Delta_{over}$, $\Delta_{under}$ are defined in Section 4.

Table 2: Deobfuscation results

jump to the final target). However, we did not do this for the purposes of this paper.

### 4.2 Flattening with Interprocedural Data Flow

For flattening with interprocedural data flow (Section 2.2), we used only the PLTO implementation, using static path feasibility analysis by itself as well as in combination with dynamic execution tracing (Section 3.3).

In this case, because our path feasibility analysis is purely intra-procedural in nature, it is unable to achieve any deobfuscation.

We do somewhat better when the static analysis is combined with dynamic tracing. The results are shown in Table 2(b). The resulting deobfuscated programs have some overestimation errors, ranging from 4.5% for the *mcf* benchmark to 43.7% for *vpr*, with an overall mean of 26.9%. There is no underestimation error for any of the benchmarks. It is significant that even though the underlying static analysis is purely intra-procedural, and has no deobfuscation effect by itself, the effect of combining it with dynamic analysis is to remove $100 - 26.9 \simeq$ about 73% of the obfuscation edges. Note that the combination of static and dynamic analyses makes a difference only for functions that are actually executed: for functions that are not executed on our test inputs, we do not consider any edges to be removed, and all of their obfuscation edges are counted towards the overestimation error in Table 2(b).

### 4.3 Flattening with Artificial Blocks and Pointers

For flattening with dummy blocks and pointers (Section 2.3), we again used only the PLTO implementation, using static path feasibility analysis by itself as well as in combination with dynamic execution tracing (Section 3.3).

The static path feasibility analysis is unable to deobfuscate this case, because it currently does not handle indirect memory accesses through pointers.

Deobfuscation improves when static and dynamic analyses are combined. The results are shown in Table 2(c). In this table, the values in the column labelled 'Added' differ from the corresponding values in Table 2(b) because the addition of artificial blocks introduces some additional control flow edges in this case. As in the case of flattening with interprocedural data flow, all of the obfuscation edges for functions that are not executed are counted towards the overestimation error. Overestimation error ranges from 1.6% for *mcf* to just under 40% for *gcc*, with an overall mean of 21.4%. There is a small amount of underestimation error as well in this case, ranging from 0.01% for *crafty* and *gcc* to 1.7% for *mcf*, with an overall mean of 0.06%. In other words, deobfuscation removes $100-(21.4+0.06) \simeq 78\%$ of the obfuscation edges.

### 4.4 Discussion

Our results indicate that automated techniques can be used to remove much of the effects of a nontrivial control flow obfuscation technique.

Basic control flow flattening turns out to be relatively straightforward to deobfuscate using purely static techniques: the PLTO implementation is able to reconstruct the original control flow graphs completely in this case, while the DIABLO implementation incurrs a mean overestimation error of about 0.7% (much of which can be easily removed using straightforward off-the-shelf techniques such as liveness analysis, which identifies variables that will never be used before being redefined [1], and jump-chain collapsing, which replaces a chain of jumps by a single jump to the final target). The total time taken by the PLTO-based deobfuscator ranges from about 7 seconds for *mcf* (constraint generation: 2.5 sec; constraint solution: 4.5 sec) to about 21 minutes for *gcc* (constraint generation: 631.5 sec; constraint solution: 640.1 sec). The DIABLO-based implementation uses an interprocedural constant propagation algorithm, originally written for code optimization purposes, without any deobfuscation-oriented modifications. In this case, interprocedural constant propagation across the huge number of control flow edges in the obfuscated programs—a number that increases quadratically during deobfuscation due to the cloning transformation described in Section 3.1—turns out to be relatively expensive, with deobfuscation times of several hours. We believe this can be reduced somewhat by applying constant propagation intra-procedurally rather than inter-procedurally: this would reduce the memory requirements for constant propagation, and thereby reduce the overhead due to paging in the virtual memory system.

The two enhancements to the basic control flow flattening technique we studied, using interprocedural data flow and memory indirection, turn out to be harder to deobfuscate. Purely static techniques, at the level of sophistication of the static analyses we used, turn out to be inadequate for these obfuscations. However, when they are combined with dynamic analyses, we are able to eliminate about 73% of the spurious edges introduced by obfuscation with interprocedural data flow, and about 78% of those introduced by obfuscation with artificial blocks and memory indirection. The total time required for constraint generation and solution in these cases is similar to those for basic control flow flattening, ranging from 7 sec to 22 mins for the case of interprocedural data flow, and from 8.5 sec to 24 mins for the case of artificial blocks and indirection.

These results represent an encouraging first step in automated deobfuscation of obfuscated programs. There are many directions in which this work can be extended, e.g., by improving the sophistication of our static analyses to incorporate interprocedural analysis or pointer alias analysis, or by using more sophisticated test case generation to improve code coverage for dynamic analysis. At the same time, it is likely that an obfuscator would use many different kinds of code obfuscation in concert, and interactions between these different obfuscations would be likely to affect the quality of deobfuscation. A study of these topics remains work for the future.

## 5 Related Work

There does not appear to be a great deal of prior work on reverse engineering obfuscated code. Kapoor [16] and Kruegel *et al.* [18] discuss algorithms for disassembling obfuscated binaries. Lakhotia and Kumar discuss techniques to handle obfuscated procedure calls in binaries [19, 20]. The focus of these works, as well as the techniques used, are very different from those described here.

A number of researchers have considered the use of dynamic analysis—either by itself, or in conjunction with static analysis—for reverse engineering [17, 26, 28]; Stroulia and Systä give an overview [26]. Much of this work focuses on dealing with legacy software, e.g., for determining modularization and semantic clustering or understanding high level design patterns, and for visualizing dynamic system behavior. All of this is fundamentally different from the work described here, which has the dual aims of identifying techniques to help reverse engineer obfuscated code, and for evaluating the strengths and weaknesses of code obfuscation techniques. In particular, our work focuses on using *simple* static and dynamic analyses to reverse engineer programs that have specifically been engineered to make reverse engineering difficult.

The idea of combining static and dynamic analyses is discussed by Ernst [12].

## 6 Conclusions

Code obfuscation has been proposed by a number of researchers as a means to make it difficult to reverse engineer software. Obfuscating transformations typically rely on the theoretical difficulty of reasoning statically about certain kinds of program properties. This paper shows, however, that it may be possible to bypass much of the effects of some obfuscations by a combination of static and dynamic analyses. In particular, we examine the problem of deobfuscating the effects of *control flow flattening*, a control obfuscation technique proposed in the research literature and used in commercial code obfuscation tools. Our results show that basic control flow flattening can be removed in a relatively straightforward way using purely static techniques, while enhancements to the basic technique can be largely deobfuscated using a combination of static and dynamic techniques.

## References


[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1985.

[2] G. Arboit. A method for watermarking java programs via opaque predicates. In *Proc. 5th. International Conference on Electronic Commerce Research (ICECR-5)*, 2002.

[3] L. Badger, L. D'Anna, D. Kilpatrick, B. Matt, A. Reisse, and T. Van Vleck. Self-protecting mobile agents obfuscation techniques evaluation report. Technical Report Report No. #01-036, NAI Labs, March 2002.

[4] S. Chandrasekharan. An evaluation of the resilience of control flow obfuscations. Undergraduate Honors Thesis, Dept. of Computer Science, The University of Arizona, Tucson, Dec. 2003.

[5] W. Cho, I. Lee, and S. Park. Against intelligent tampering: Software tamper resistance by extended control flow obfuscation. In *Proc. World Multiconference on Systems, Cybernetics, and Informatics*, 2001.

[6] S. Chow, Y. Gu, H. Johnson, and V. A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *Proc. 4th. Information Security Conference (ISC 2001)*, Springer LNCS vol. 2000, pages 144–155, 2001.

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

[8] C. Collberg and C. Thomborson. Software watermarking: Models and dynamic embeddings. In *Proc. 26th. ACM Symposium on Principles of Programming Languages*, pages 311–324, January 1999.

[9] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation – tools for software protection. *IEEE Transactions on Software Engineering*, 28(8), August 2002.

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

[11] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere. Link-time optimization of arm binaries. In *Proc. 2004 ACM Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, pages 211–220, 7 2004.

[12] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis, Portland, OR*, pages 24–27, May 2003.

[13] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, January/February 2002.

[14] S. Guyer and K. McKinley. Finding your cronies: Static analysis for dynamic object colocation. In *Proc. ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA'04)*, pages 237–250, October 2004.

[15] M. Hind and A. Pioli. Which pointer analysis should I use? In *Proc. 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[16] A. Kapoor. An approach towards disassembly of malicious binaries. Master's thesis, University of Louisiana at Lafayette, 2004.

[17] R. Kazman and S. J. Carrière. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering: An International Journal*, 6(2):107–138, April 1999.

[18] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proc. 13th USENIX Security Symposium*, August 2004.

[19] E. U. Kumar, A. Kapoor, and A. Lakhotia. DOC – answering the hidden 'call' of a virus. Virus Bulletin, April 2005.

[20] A. Lakhotia and E. U. Kumar. Abstract stack graph to detect obfuscated calls in binaries. In *Proc. 4th. IEEE International Workshop on Source Code Analysis and Manipulation*, pages 17–26, September 2004.

[21] C. Linn and S.K. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. 10th. ACM Conference on Computer and Communications Security (CCS 2003)*, pages 290–299, October 2003.

[22] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Software obfuscation on a theoretical basis and its implementation. *IEEE Trans. Fundamentals*, E86-A(1), January 2003.

[23] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Comm. ACM*, 35:102–114, August 1992.

[24] B. Schwarz, S. K. Debray, and G. R. Andrews. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.

[25] Preemptive Solutions. Dotfuscator. `www.preemptive.com/products/dotfuscator`.

[26] E. Stroulia and T. Systä. Dynamic analysis for reverse engineering and program understanding. *ACM SIGAPP Applied Computing Review*, 10(1):8–17, 2002.

[27] Symantec Corp. Understanding and managing polymorphic viruses. Technical report, 1996.

[28] T. Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, Dept. of Computer and Information Sciences, University of Tampere, Finland, 2000.

[29] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *Proc. International Conference of Dependable Systems and Networks*, July 2001.

[30] Chenxi Wang. *A Security Architecture for Survivability Mechanisms*. PhD thesis, Department of Computer Science, University of Virginia, October 2000.