Metrics-based Evaluation of Slicing Obfuscations

Anirban Majumdar, Stephen Drape and Clark Thomborson
Department of Computer Science
The University of Auckland, New Zealand.
email: {anirban, stephen, cthombor}@cs.auckland.ac.nz

Abstract

An obfuscation aims to transform a program, without affecting its functionality, so that some secret data within the program can be hidden for as long as possible from an adversary armed with reverse engineering tools. Slicing is a reverse engineering technique which produces a subset of a program that is dependent on a particular program point and is used to aid in program comprehension. Thus slicing could be used as a way of attacking obfuscated programs.

In this contributions, we highlight a few obfuscating transforms that were proposed in an earlier publication for making slicing attacks difficult to mount and then illustrate an experimental design to evaluate the strength of these transforms with respect to well defined slicing metrics.

Keywords: Obfuscation, Slicing, Metrics, Evaluation

1 Introduction

The goal of software protection through code obfuscation is to transform the source code of an application to the point that it becomes unintelligible to automated program comprehension and analysis tools [13]. The motivation for obfuscating code arises from the problem of software piracy. A piece of distributed software may contain secrets (tangible artefacts such as cryptographic keys or intellectual artefacts such as a patented algorithm) which a software pirate may want to steal or tamper. This could be facilitated by malicious *reverse engineering* — a process of discovering the technological principles of a system through analysis of its structure, function and operation [5]. With the advent of platform independent mobile code, reverse engineering has become a serious challenge to the software industry and commercially successful software such as the Skype internet telephony client [4], and most license-control systems rely, at least in part, on obfuscation for their security.

The first formal definition of obfuscation was given by Collberg *et al.* [6, 7] where an obfuscator was defined in terms of a semantic-preserving transformation function $\mathcal{O}$ which maps a program $\mathcal{P}$ to a program $\mathcal{O}(\mathcal{P})$ such that if $\mathcal{P}$ fails to terminate or terminates with an error, then $\mathcal{O}(\mathcal{P})$ may or may not terminate. Otherwise, $\mathcal{O}(\mathcal{P})$ must terminate and produce the same output as $\mathcal{P}$. Barak *et al.* [2] strengthened the formalism of Collberg *et al.* by defining an obfuscator, $\mathcal{O}$, in terms of a *compiler* that takes a program, $\mathcal{P}$, as input and produces an obfuscated program, $\mathcal{O}(\mathcal{P})$, as output such that $\mathcal{O}(\mathcal{P})$ is *functionally equivalent* to $\mathcal{O}(\mathcal{P})$, the running time of $\mathcal{O}(\mathcal{P})$ is at most *polynomially larger* than that of $\mathcal{P}$, and $\mathcal{O}(\mathcal{P})$ simulates a *virtual black-box*. It was observed in [8] that the *virtual black-box* property of [2] is too strong. Obfuscators will be of practical use even if they do not provide perfect black boxes. Therefore, the focus of research in obfuscation has shifted to constructing obfuscating transforms that are *difficult* (but not necessarily *impossible*) for an adversary to reverse engineer.

Our motivation is derived from the difficulty of empirically evaluating the obfuscatory strength of seemingly resilient obfuscating transforms. Majumdar *et al.* [12] (and separately in [18]) observed that in order to do such evaluation, we need to be able to answer the following question — "What kinds of tools and program analyses are suitable for evaluating a particular obfuscating transform?" An immediate observation that follows from this question is the fact that it is probably impossible to come up with one general purpose reverse engineering tool that will deobfuscate *any* obfuscating transform (thinking from an adversary's perspective, an adversary will have to use different techniques to deobfuscate a program obfuscated with different kinds of transforms). As a corollary, we can argue that it is also ambitious to design an obfuscating transform that will withstand *all* possible reverse engineering attacks.

In the domain of software engineering, program slicing is widely used for program maintenance and comprehension [3, 10, 17]. These techniques form the basis of reverse engineering since the goal of these techniques is to identify *relevant* parts of the code from other unnecessary details. Indeed, this is what a software pirate intends to do when he/she attempts to steal or change the relevant parts (of her/his interest) in the code with the intention of reusing

the extracted module in illegal derivative software or invalidating the code licensing routine. The main challenge in deterring such kinds of code comprehension attacks is to intertwine the relevant code with other irrelevant sections so that the attacker fails to recognise the portions of interest from the code. In [9], the authors proposed ways to design obfuscations (called *slicing obfuscations*) by introducing dependencies between seemingly unrelated portions of code to try to defend against slicing attacks. In this contribution, we will demonstrate an experimental design for evaluating the obfuscatory strength (measured in terms of various slicing metrics) of those proposed transforms in the restricted security model of static slicing attacks.

2 Experimental Design

Since slicing can be used to aid program comprehension, we should design obfuscations that make slicing less useful. In this paper we present some experiments in which we manufacture obfuscations that are designed to decrease the effectiveness of slicing. We have based our experiments on CodeSurfer, a static program slicer for code written in C [1], which uses system dependence graphs (SDGs), an intermediate structure for representing programs [11]. Slicing using SDGs is the most precise and complete slicing method currently available [17]. CodeSurfer is capable of backward slicing, forward slicing, and chopping. A backward slice includes all program points that affect a given point in the program. A forward slice includes all program points that are affected by a given point in the program. A chop includes all program points that are affected between a source program point and a sink program point. For our illustrations in this paper, we use the backward slicing feature of CodeSurfer from program output statements.

For our experiments, we consider a set of five C programs and discuss how we can obfuscate them with the aim of making slicing less effective. We have restricted our programs to use a subclass of common program constructs (assignments, output statements, conditionals and loops) and thus our obfuscations are applicable to languages other than C. We also restrict ourselves to intra-procedural slices. In the rest of this section we discuss how we set up our experiments by describing what properties were are going to measure, how we compute these measurements and what transformations we are going to use as obfuscations.

2.1 Slicing Metrics

The goal of our experiments is to investigate whether our slicing obfuscations, which were proposed in [9], make slicing less effective as a tool for program comprehension. To help us evaluate our obfuscations we will consider some metrics used to assess program slicing.

Meyers and Binkley [14] studied five slice-based metrics which were proposed as measures of the quality of software. Three of the metrics (*Tightness*, *Coverage* and *Overlap*) were originally presented by Weiser [20] and the other two (*MinCoverage* and *MaxCoverage*) were proposed by Ott and Thuss [15].

For each program (method) M in our experiments we will concentrate on variables which are output (for instance, as part of a **printf** statement) and so we have a set of output variables V_O. Our slicing point will be the last output statement for each output variable. For each $v_i \in V_O$ the backwards slice for v_i is denoted by SL_i, SL_{int} is defined to be $\bigcap_i SL_i$ (*i.e.* the intersection of all the slices) and $|\ldots|$ denotes the size. Here are the definitions taken from [14] for the five slice-based metrics that we will use.

Definition 1 (Tightness Metric)**.** Tightness measures the number of statements common to every slice.

$$T(M) = \frac{|SL_{int}|}{|M|}$$

Definition 2 (Minimum Coverage)**.** Minimum coverage is the ratio of the smallest slice in a method to it's length.

$$Min(M) = \frac{1}{|M|} \min_i |SL_i|$$

Definition 3 (Coverage)**.** Coverage compares the length of slices to the length of the entire method.

$$C(M) = \frac{1}{|V_O|} \sum_{i=1}^{|V_O|} \frac{|SL_i|}{|M|}$$

Definition 4 (Maximum Coverage)**.** Maximum coverage is the ratio of the largest slice in a method to it's length.

$$Max(M) = \frac{1}{|M|} \max_i |SL_i|$$

Definition 5 (Overlap)**.** Overlap is a measure of how many statements in a slice are found in all the other slices.

$$O(M) = \frac{1}{|V_O|} \sum_{i=1}^{|V_O|} \frac{|SL_{int}|}{|SL_i|}$$

These metrics give values in the range between 0 and 1 (so we give our values as percentages) and, in terms of slicing, good slices are indicated by low metric values. Our intention in this paper is to see whether the obfuscations from [9] increase these slicing metrics. We use these metrics to evaluate how our obfuscations affect the effectiveness of slicing. Our aim when performing slicing obfuscations is to increase some (and ideally *all*) of the slicing metrics.

We use CodeSurfer to compute the backward slice for each output variable. To compute the metrics for each slice

we need to measure the size of the method and well as the size of the slices in a consistent manner. CodeSurfer has a feature which allows the user to compute sets of program points (*i.e.* nodes in the SDG) using a *set calculator*. We can use this calculator to compute measurements for slices as well as for methods and we can perform set operations.

For a particular method M, the size $|M|$ is obtained by computing the number of program points contained within the method itself (and not any that are contained in calls to other methods). The set of program points for a backwards slice from a point in M may contain points from other methods, such as method calls, and so the size of a slice may actually be bigger than $|M|$. Since in this paper we are only concentrating on a single method M, when computing the size of the slice we will only consider the part of the slice that is contained in M. Using standard flattening techniques, we inline the code of every method into one method M which contains the desired slicing criterion (output variables). Our results can be seen in Table 1 which we discuss in more detail in Section 4.

2.2 Transformations

For completeness, we provide a brief overview of some of the slicing obfuscations from [9] which we will use to obfuscate our candidate programs. Suppose that we have an output variable y. To manufacture a slicing obfuscation for y we have to ensure that the slice for the obfuscation contains points that were not included in the original slice for y. We can do this by creating dependencies on y on any variable x that was not in the original slice for y. We will use the following three transformations (which are taken from [6]) to create dependencies for y on x.

Bogus Predicate We can add a bogus predicate q^T, which is always true, as follows:

$$x = G;\ S \Rrightarrow x = G;\ \textbf{if}\,(q^T)\ S;\ \textbf{else}\ y = H(x);$$

We can add a false predicate similarly.

Adding to a while loop We can add a new variable j that depends on both x and y as follows (provided that $c \Rightarrow p$):

$$\textbf{while}\,(c)\,S \Rrightarrow \textbf{while}\,(c\ \&\&\ p(j))\,\{S;\ j = F(x,y);\}$$

Variable Encoding An encoding for y has the form $y \Rrightarrow \alpha * y + \beta$. We can adapt this so that y depends on x as follows: $y \Rrightarrow \alpha * y + \beta * x$. We have three rewrite rules:

- A use of y: $\quad U(y) \Rrightarrow U((y - \beta * x) \div \alpha)$
- A definition of y:

$$y = E \Rrightarrow y = \alpha * E[\tfrac{y-\beta*x}{\alpha}/y] + \beta * x$$

- A definition of x:

$$x = V(y) \Rrightarrow \{t = y - \beta * x;\ x = V(t \div a);\ y = t + \beta * x;\}$$

where t is a fresh, temporary variable.

3 Obfuscating Example Programs

For our experiments, we considered obfuscating five different programs with the goal of creating dependencies between all of the output variables. In the rest of this section we briefly describe each of the programs and the obfuscations that we performed. We restrict our obfuscations to the transformations that we discussed in Section 2.2.

3.1 Word Count

The word count program takes in a file and outputs the number of lines (nl), words (nw) and characters (nc). The results for slicing for these three variables can be found in Table 1 where the original program is called *wc*.

As an obfuscation we added this simple bogus predicate

$$\textbf{if}\,(nl > nc)\ nw = nc + nl;\ \textbf{else}\,\{\textbf{if}\,(nw > nc)\ nc = nw - nl;\}$$

before the three values are printed. This single obfuscation adds dependencies between the three variables and so the slice for each variable contains the definitions for the other variables. We called this obfuscated method *wc-obf1* and the metric values for this method can be found in Table 1.

3.2 Product and Sum

The classic slicing example of a method calculating the product ($prod$) and sum (sum) of the first n positive integers was considered next. Our program contained a **while** loop using a variable i which counts up from 0 to n.

We first created a dependency for $prod$ on sum by adding the condition of $\vee sum < 0$ in the guard of the **while** loop. Next we need to create a dependency for sum and so we added a bogus predicate p^T around the statement $prod = prod * i$. The metrics values for three methods can be seen in Table 1. The method *ps* relates to the original unobfuscated method, in *ps-obf1* the loop dependency was created and finally we added the predicate in *ps-obf2*.

3.3 Search Sort

The search sort program (obtained from [19]) takes an argument n from the user, performs different sorts and searches on n elements and then displays the time taken to do each one. So the two output variables at each stage

Method M	$\|M\|$	$\|V_O\|$	For each v_i the slice size $\|SL_i\|$						$\|SL_{int}\|$	$T(M)$	$Min(M)$	$C(M)$	$Max(M)$	$O(M)$
wc	36	3	nl	15	nw	20	nc	10	7	19.4%	27.8%	41.7%	55.6%	50.6%
wc-obf1	42	3	nl	30	nw	30	nc	30	28	66.7%	71.4%	71.4%	71.4%	93.3%
ps	21	2	$prod$	12	sum	12			7	33.3%	57.1%	57.1%	57.1%	58.3%
ps-obf1	22	2	$prod$	16	sum	13			11	50.0%	59.1%	65.9%	72.7%	76.7%
ps-obf2	26	2	$prod$	19	sum	19			17	65.4%	73.1%	73.1%	73.1%	89.5%
search	107	2	n	9	$secs$	11			2	1.9%	8.4%	9.3%	10.3%	20.2%
search-obf1	120	2	n	45	$secs$	11			10	8.3%	9.2%	23.3%	37.5%	56.6%
search-obf2	127	2	n	49	$secs$	48			46	36.2%	37.8%	38.2%	38.6%	94.9%
rov	124	2	$fuel$	23	$dist$	46			19	15.3%	18.5%	27.8%	37.1%	62.0%
rov-obf1	129	2	$fuel$	60	$dist$	46			45	34.9%	35.7%	41.1%	46.5%	86.4%
rov-obf2	132	2	$fuel$	62	$dist$	60			59	44.7%	45.5%	46.2%	47.0%	96.7%
scatter	143	3	si	116	ru	111	i	9	8	5.6%	6.3%	55.0%	81.1%	34.3%
scatter-obf1	148	3	si	132	ru	132	i	132	131	88.5%	89.2%	89.2%	89.2%	99.2%
scatter-obf2	150	3	si	139	ru	139	i	139	138	92.0%	92.7%	92.7%	92.7%	99.3%

Table 1. Table of results for our five example programs with their obfuscations. There is a separate row in the table for recording the slicing metric values of the example programs and their obfuscated counterparts. The row labelled wc, for example, records the slicing metric values for the unobfuscated instance of Word Count example; whereas, *wc-obf1* indicates the metric values when slicing obfuscations have been applied. The columns from $|M|$ to $|SL_{int}|$ reflect measures with respect to the number of SDG nodes. The latter columns indicate the slicing metric values.

are the number of elements n and the number of seconds $secs$. This program is different from those considered so far as it contains different methods which the main method calls. However the results for all of these methods are discarded and only the time taken to perform each method is computed. Thus slices for both n and $secs$ only contain statements from the main method. So for our experiments we only consider changing statements in the main method. (Obviously when obfuscating we should aim to obfuscate the other methods in the program but this is beyond the scope of this paper.) The metric results for the main method *search* can be found in Table 1.

As n is constant throughout the program (it is inputted by the user) we attempt to make n vary by adding a variable encoding. We would like to create a dependency for n on sec but n is declared as an integer and $secs$ is a float. So we declare a new integer variable k which we define as $k =$ (**int**) $10 * secs$ and we perform the transformation $n \Rrightarrow n + k$. By the replacement rules for variable encoding we need to redefine k every time that $secs$ is redefined and so we can use a different declaration for k. Note that this kind of transformation could make the value of n overflow and so we should put in checks to ensure that this does not happen. The results for this obfuscation can be found in Table 1 for the method *search-obf1*.

To create some dependencies for $secs$ we placed two bogus predicates near the end of the method. The value of $secs$ is obtained by computing the difference between two different clock values contained in $c1$ and $c2$. So, for example, we can change the first assignment to:

if $(secs >= 0)\ c1 =$ **clock()**; **else** $c1 = n/$**clock()**;

The results after adding these predicates can be found in Table 1 for the method *search-obf2*.

3.4 Rover

The rover program checks whether a plan for manoeuvring a vehicle around an obstacle to a given target, with a limited amount of fuel, satisfies certain constraints. It simulates a land rover vehicle which needs to be driven around a rock on the Martian surface by giving it coordinates as input [19]. However, the vehicle goes off-track from the specified target and it has limited amount of fuel. The challenge is to bring the vehicle within 5km of the target by inputting a sequence of coordinates which will also not exhaust the fuel before it reaches the target. Here we consider two output variables $fuel$ and $dist$. The results for the original method can be seen in the *rov* row of Table 1.

So that $fuel$ depends on $dist$ we perform the following variable encoding $fuel \Rrightarrow fuel + dist$. Note that we have to add an extra temporary variable t to perform the encoding. The values for this obfuscation can be found in the *rov-obf1* row of Table 1.

The value of $dist$ is $\sqrt{dx^2 + dy^2}$ where dx and dy are two other variables. To create a dependency for $dist$ we

changed an assignment to dx as follows:

$dx = E; \Rrightarrow$ **if** $(fuel >= dist)\ dx = E;$ **else** $dx = bogus;$

By the variable encoding above we know that since $dist \geq 0$ then $fuel \geq dist$. The row *rov-obf2* in Table 1 contains the values for this obfuscation.

3.5 Scattering

The scattering program is a typical physics problem dealing with the famous Rutherford's scattering experiment for finding the size of the nucleus of an atom (from Tao Pang's book "An Introduction to Computational Physics" [16]). The original program contained several procedures for performing integration using Simpson's rule, finding roots using the Secant method, and finding the first and second order derivatives with the three-point formula. We flattened the procedures by inlining them in the *main()* procedure (which we call *scatter*) and we consider three output variables si, ru and i.

The variable i is used repeatedly as an induction variable in different parts of the method. So we link up different uses of i by adding variable encodings to change initialisations of i and by creating bogus predicates. Also, we observe in the program source code that the statement $b = b0 + i * db;$ maintains the invariant $i \geq b \ \wedge \ b0 \leq b$. So we introduce the following bogus predicates:

$$si = E; \ \Rrightarrow \ \textbf{if}\ (i + 1 > b)\ si = E;\ \textbf{else}\ si = bogus;$$
$$ru = E; \ \Rrightarrow \ \textbf{if}\ (b0 <= b)\ ru = E;\ \textbf{else}\ ru = bogus;$$

into the code. The row *scatter-obf1* in Table 1 contains the values for these obfuscations. In an attempt to provide maximum code coverage, we introduce other similar bogus predicates to include more statements in SL_{int}. The row *scatter-obf2* in Table 1 indicates the changes brought about by the addition of these predicates.

4 Results

The results are depicted in Table 1. Each of the rows indicate the number of SDG nodes and slicing metric values for each unobfuscated example program (abbreviated by program name) and its obfuscated counterpart(s) (indicated by abbreviated program name followed by $obf\#$ where $\#$ indicates the obfuscated instance). The table has been partitioned lengthways in five sections corresponding to groups of our five example programs. The columns $|M|$ to $|SL_{int}|$ reflect measures with respect to the number of SDG nodes. These are used to calculate the slicing metric values as defined in Section 2.1. The columns $T(M)$ to $O(M)$ indicate the corresponding slicing metrics.

From Table 1 we can see that we have increased the values of *all* the metrics and we can also see that we do not significantly increase the size of the methods (the worst is a 24% increase for *ps* but the size of the slices increase by 58%). One exception is the metric values obtained for the search sort program. We see from Table 1 that its metric values are much lower than those obtained for other programs. This exception arose since we deliberately avoided inlining the searching and sorting modules of the program in order to observe any interesting characteristics of non-inlining.

The biggest gain was in the scattering example for the variable i. By adding an extra 7 nodes (only a 5% size increase) we managed to increase the size of the slice for i by 130 nodes. We achieved this gain by observing that in the original scattering program, i was used repeatedly as an induction variable and so we added dependencies between the different uses of i.

One aspect that is key to increasing the metrics values is to ensure that we increase $|SL_{int}|$ (*i.e.* the intersection of all the slices) and so that we increase the slice sizes and the values of the Tightness and Overlap metrics. To increase $|SL_{int}|$ we need to add dependencies between all of the variables in V_O.

5 Conclusions and Future Work

In this paper, we have built an experimental framework and conducted experiments to observe the effects of applying slicing obfuscations (defined in [9]) on standard slicing metrics described in [14]. The objective was to substantiate the fact that slicing obfuscations indeed made the slicing metrics worse. This is evident from the results obtained in Table 1. Even though the size of our experiment was relatively small, it can be used in a much wider scope of obfuscating generic programs against static slicer attacks.

Our obfuscations were created manually and so one area for future work is to consider automating the process of adding obfuscating transforms. One particular concern for automation is the development of heuristics to decide where to place slicing obfuscations in order to maximise the effects of the transforms. Also, for some of our programs we added more than one obfuscation and so, we have created a composite obfuscation. Further work is needed study the effects of composing obfuscations together and, in particular, does the order in which we add obfuscations (and the order in which we consider the output variables) matter?

Even with these simple transformation we have still managed to decrease the effectiveness of slicing which was our stated goal. When faced with an attacker who is armed with more than just a slicer we will obviously have to design more complicated transformations. This will involve creating predicates that are harder for an attacker to understand, using different program constructs such as arrays and

pointers and dealing with inter-procedural constructs. Another interesting area for future work is to see if a similar approach can be used to design obfuscations for protecting programs against other forms of static analysis attacks.

References

[1] Paul Anderson and Tim Teitelbaum. Software inspection using CodeSurfer. In *Proceedings of the Workshop on Inspection in Software Engineering (WISE 2001)*, Paris, France, July 2001. IEEE Computer Society.

[2] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 1–18. Springer-Verlag, 2001.

[3] David Binkley and Mark Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, pages 44–53, Washington, DC, USA, 2003. IEEE Computer Society.

[4] Phillipe Biondi and Fabrice Desclaux. Silver needle in the Skype. Presentation at BlackHat Europe, March 2006. URL: www.blackhat.com/html/bh-media-archives/bh-archives-2006.html.

[5] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[6] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

[7] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196, New York, NY, USA, 1998. ACM Press.

[8] Stephen Drape. *Obfuscation of Abstract Data-Types*. DPhil thesis, Oxford University Computing Laboratory, 2004.

[9] Stephen Drape, Anirban Majumdar, and Clark Thomborson. Slicing aided design of obfuscating transforms. In *IEEE/ACIS ICIS 2007: To appear in proceedings of the International Computing and Information Systems Conference (ICIS 2007)*, Melbourne, Australia, 2007. IEEE Computer Society.

[10] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng.*, 17(8):751–761, 1991.

[11] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.(TOPLAS)*, 12(1):26–60, 1990.

[12] Anirban Majumdar, Antoine Monsifrot, and Clark Thomborson. On evaluating obfuscatory strength of alias-based transforms using static analysis. In *Proceedings of ADCOM2006*, Mangalore, India, 2006. IEEE Computer Society.

[13] Anirban Majumdar, Clark D. Thomborson, and Stephen Drape. A survey of control-flow obfuscations. In *Information Systems Security, Second International Conference, ICISS 2006, Kolkata, India*, pages 353–356, December 2006.

[14] Timothy M. Meyers and David Binkley. Slice-based cohesion metrics and software intervention. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 256–265, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Linda M. Ott and Jeffrey J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Software Metrics Symposium*, pages 78–81, 1993.

[16] Tao Pang. *An Introduction to Computational Physics*. Cambridge University Press, 1997. URL: www.physics.unlv.edu/∼pang/cp.html.

[17] Frank Tip. A survey of program slicing techniques. Technical report, CWI (Centre for Mathematics and Computer Science) CS-R9438, Amsterdam, The Netherlands, 1994.

[18] Sharath K. Udupa, Saumya K. Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 45–54, Washington, DC, USA, 2005. IEEE Computer Society.

[19] University of Texas at Austin CS1713 Course. URL: www.cs.utexas.edu/users/djimenez/utsa/cs1713-3/c.

[20] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.