# Slicing Aided Design of Obfuscating Transforms

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

***Abstract*—An obfuscation aims to transform a program, without affecting its functionality, so that some secret information 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 aims to produce a subset of a program which 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. Can we design obfuscations which are more resilient to slicing attacks?**

**In this paper we present a novel approach to creating obfuscating transforms which are designed to survive slicing attacks. We show how we can utilise the information gained from slicing a program to aid us in manufacturing obfuscations that are more resistant to slicing. We give a definition for what it means for a transformation to be a slicing obfuscation and we illustrate our approach with a number of obfuscating transforms.**

Keywords: Obfuscation, Slicing, Program Transformation

## I. Introduction

Numerous PhD dissertations and publications have been written on the issue of software protection by means of code obfuscation following the publication of a technical report by Collberg *et al.* [6] in 1997. The intent of obfuscation is to transform the source code of an application to the point that it becomes unintelligible to automated program comprehension and analysis tools. The motivation for research in obfuscation stems from the problem of software piracy. A software pirate will make every effort to steal or change the logic contained in commercially distributed code by reverse engineering; and, the software developer would want to prevent the pirate from stealing those code secrets. With the advent of platform independent mobile code, reverse engineering has become a serious challenge to the software industry [14]. Commercially successful software such as the Skype internet telephony client [5], the SDC Java DRM [19], and most license-control systems rely, at least in part, on obfuscation for their security.

In [16] it was observed that most obfuscating transforms in the existing literature were designed with the goal of being resilient against *all* possible reverse engineering attacks. This is ambitious since Barak *et al.* [2] proved it impossible to design a completely-secure software black-box for any broad class of programs. A recent DPhil thesis [9] designed data obfuscations in the functional programming domain targeted at deterring adversaries armed with static theorem provers. Continuing this trend, in this paper, we describe a novel and promising way to design obfuscating transforms by using an attack tool. Our approach is based on the maxim that "attack is the best form of defence". Specifically, we will show how to use a static slicer to manufacture obfuscations that are resistant to slicing.

## II. Program obfuscation

The first formal definition of obfuscation was given by Collberg *et al.* [6], [7]. They defined an obfuscator 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 $\mathcal{P}$.

Collberg *et al.* classified obfuscating transforms into three general categories:

- **Layout obfuscations:** This class of transforms changes or removes useful information from the intermediate language code or the source code without affecting the instructions that contribute to actual computation. Most common obfuscating techniques fall in this category, such as removing debugging information and comments, and scrambling/renaming identifiers.
- **Data obfuscations:** This category of transforms is targeted at the data and data structures contained in the program. Using data-obfuscating transforms, data encoding can be changed, variables can be split or merged, and arrays can be split, folded or merged.
- **Control-flow obfuscations:** The objective of this category of transforms is to alter the flow of control within the code. Examples of control-flow obfuscations are: reordering of statements, methods, loops and hiding the actual control flow behind irrelevant conditional statements.

Layout obfuscations have been extensively used in commercial obfuscators like the Dotfuscator [12]. In this contribution, we will focus on two other classes of obfuscation; namely, data and control-flow obfuscations since we believe layout obfuscations such as comment stripping and identifier renaming are generally only good for confusing a human adversary.

It was observed in [9] 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 obfuscation research has shifted to finding obfuscating transforms that are *difficult* (but not necessarily *impossible*) for an adversary to reverse engineer.

## III. Using static slicer for mounting attacks

In the domain of software engineering, program slicing is widely used for program maintenance, integration, modularisation, and comprehension [3], [4], [10], [18], [20]. These techniques form the basis of reverse engineering since the primary objective in these techniques is to abstract away the *relevant* parts of the code from other unnecessary details. Indeed, this is what a software pirate intends to do when he/she makes an attempt 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.

### A. Experimental Design

We have based our experiments on CodeSurfer, a static program slicer for code written in C [1]. It 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 [20]. 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 on program output statements.

Mobile code (such as Java byte code, Microsoft's MSIL) is considered more vulnerable to reverse engineering attacks than binary executables. Nonetheless we have designed our experiments using the constructs of the C language. Our design choice was influenced by two primary factors:

- We want to substantiate the rather suppositional claims in the literature that it is easy to mount static slicing attacks on simple obfuscating transforms [7]. Since we do not know how to arbitrarily generate hard problem instances, we limit ourselves to manufacturing resilient obfuscating transforms using simple program constructs in this paper. Therefore, we choose the language C and restrict our obfuscations on a subclass of program constructs (assignments, output statements, conditionals and loops) that is common for all imperative languages.
- Secondly, it was difficult to find a full-fledged working slicer for a language other than C. Experience with using third-party tools for experimentation suggests that most of the tools built as part of academic projects are in their prototypical phase and not well maintained [15]. The only well-known static slicer for Java programs is Indus [13]; again, it is an academic project and is yet to be empirically evaluated for correctness and performance. CodeSurfer is an exception — it has been widely used since the release of the prototype Wisconsin Program-Slicing Project in 1996 (based on the slicing algorithm by Horowitz *et al.* [11]) and has been extensively evaluated in published literature [3], [4], [17].

```
int sumprod (int n) {
  int i = 0;
  int x = 0;
  int y = 1;
  while (i < n) {
    i++;
    x = x + i;
    y = y * i;        }
  out(x);
  out(y);        }
```

Fig. 1. Method to calculate the sum and product of the first $n$ positive integers (the backwards slice from **out**$(y)$ is indicated by underlined statements).

### B. Slicing Notation

We use the notation

$$slice(P, s, \mathcal{V})$$

to denote a backwards slice of program $P$ from the statement $S$ with respect to the set of variables $\mathcal{V}$. The statement $s$ and the set $\mathcal{V}$ are called the *slicing criterion*. We restrict the set $\mathcal{V}$ to variables which are output — an output variable could be as part of a **printf** statement or passed to another method. We will use the statement **out** to denote an output. In this paper we will be concerned only with backwards slices from **out** statements.

If we apply an obfuscation $\mathcal{O}$ to a program $P$ we insist that the obfuscation does not change the input/output behaviour of the program. Our obfuscations will impose a relationship between the original and obfuscated variables and so we can write $\mathcal{O}(\mathcal{V})$ to denote the set of variables in $\mathcal{O}(P)$ and the output statement $s$ remains but it may take different arguments. Note that we consider obfuscation as refinement [8] to establish a "relationship" between the original and obfuscated programs — it is beyond the scope of this paper to discuss this further (more details can be found in [9]).

### C. An example

Throughout the rest of the paper we will consider the program given in Figure 1 as a running example. The statement **out**$(x)$ produces the sum of the integers 1 to $n$, which we write as $sum(1..n)$ and **out**$(y)$ produces the product $prod(1..n)$. In Figure 1 we have also indicated the backwards slice from **out**$(y)$ by underlining.

Now let us perform a simple obfuscation on the sum/product example by adding some statements involving a dummy variable $d$. We can use a slicer on this program to produce a backwards slice from **out**$(y)$ with respect to the variable $y$. The obfuscation and the slice are shown in Figure 2. We can see that the statements contained in the slice do not include the dummy variable $d$ (and in fact the slice is exactly the same as the one in Figure 1). We obtain a similar result if we perform a backwards slice from **out**$(x)$.

```
int sumprodobf (int n) {
  int i = 0;
  int x = 0;
  int y = 1;
  int d = x + y;
  while (i < n) {
    i++;
    x = x + i;
    y = y * i;
    d = d * x + i + y;          }
  out(x);
  out(y);          }
```

Fig. 2. Simple obfuscation of the sum/product example (the backwards slice from **out**$(y)$ is indicated by underlined statements).

This illustrates that a slicer can be used as an attack against this simple obfuscation. The question we will try to address in this paper is: *Can we manufacture better obfuscations that will withstand similar slicing attacks?*

## IV. Using Slicing to Define Obfuscation

As mentioned earlier, slicing is often used to aid program comprehension. As obfuscation is used to make programs harder to understand we should aim to create obfuscations that make slicing less useful — such obfuscations will be called *slicing obfuscations*. But what do we mean by "less useful"? In [17] various metrics are given which measure the effectiveness of a slice. These metrics rely on the size of a slice and so a first attempt at defining a slicing obfuscation is that such an obfuscation creates larger slices.

*First attempt:* An obfuscation $\mathcal{O}$ is a *slicing obfuscation* for a program $P$, an output statement $s$ and set of variables $\mathcal{V}$ if it increases the size of the slice, *i.e.*

$$|slice(P, s, \mathcal{V})| < |slice(\mathcal{O}(P), \mathcal{O}(s), \mathcal{O}(\mathcal{V}))|$$

For our restricted language we could measure the size by counting the number of assignments, loops, conditional and output statements.

Let us consider the following (trivial) program $Q$:

$$\begin{array}{ll} s_1: & x = 2; \\ s_2: & y = 3; \\ o_1: & \textbf{out}(x); \\ o_2: & \textbf{out}(y); \end{array}$$

Then $slice(Q, o_2, \{y\}) = s_2; o_2$ and so just contains two statements.

A simple transformation $\mathcal{O}(Q)$ is:

$$\begin{array}{ll} s_1: & x = 2; \\ s_3: & y = 1; \\ s_4: & y++; \\ s_5: & y++; \\ o_1: & \textbf{out}(x); \\ o_2: & \textbf{out}(y); \end{array}$$

and now $slice(\mathcal{O}(Q), o_2, \{y\}) = s_3; s_4; s_5; o_2$ and so we have a bigger slice. We can continue performing this kind of transformation and so we can arbitrarily increase the size of the slice — thus we can never obtain a maximal slice. We could just consider the statements in the slice that came from the original program but our transformations may change every statement of the original program and so this would not be a practical measure.

Our example transformation does not affect any of the statements that are left out of the slice (*i.e.* $s_1$ and $o_1$). A more suitable obfuscation would try to increase the size of the slice by including more of the statements that are left behind. We call the statements that are omitted from a slice the *orphaned* statements of a slice. We can define:

$$orphan(P, s, \mathcal{V}) = P \backslash slice(P, s, \mathcal{V})$$

Using this expression, we can define a slicing obfuscation as follows:

*Definition:* An obfuscation $\mathcal{O}$ is a **slicing obfuscation** for a program $P$, an output statement $s$ and set of variables $\mathcal{V}$ if it decreases the number of orphaned statements, *i.e.*

$$|orphan(P, s, \mathcal{V})| > |orphan(\mathcal{O}(P), \mathcal{O}(s), \mathcal{O}(\mathcal{V}))|$$

For the example $Q$ above, suppose that we perform the following obfuscation, $\mathcal{O}(Q)$:

$$\begin{array}{ll} s_1: & x = 2; \\ s_3: & y = x + 1; \\ o_1: & \textbf{out}(x); \\ o_2: & \textbf{out}(y); \end{array}$$

Now, $slice(\mathcal{O}(Q), o_2, \{y\}) = s_1; s_3; o_2$ and this obfuscation has decreased the number of orphaned statements as well as increasing the size of the slice.

Now consider the program and the backwards slice for $y$ given in Figure 1. We can see that slicing at **out**$(y)$ leaves us with the following statements:

```
int x = 0;
...
  x = x + i;
...
out(x);
```

According to our slicing definition of obfuscation we would like to reduce the number of orphaned statements. So, how can we obfuscate our example so that a slice contains the two orphaned assignments to $x$?

## V. Example Transforms

In this section, we will discuss some program transformations that are suitable as slicing obfuscations. To help us to explain how these transformations operate we will use the example given in Figure 1. We assume that our slicing criterion is the statement $s \equiv$ **out**$(y)$ and the set of variables is $\{y\}$. By slicing at $s$, the slice contains all the statements which contribute to the value of $y$.

When obfuscating we should aim that the backwards slice for $y$ should include all the statements (except for output statements for variables other than $y$) which occur before the slicing point. Thus when manufacturing slicing obfuscations we should try to create dependencies between variables.

## *A. Rewriting expressions*

We can create a dependency on $x$ by rewriting assignments to $y$. Suppose that we have an assignment to $x$ which depends on some other variable $a$. Then we have a sequence of statement in which neither $x$ nor $a$ is redefined followed by an assignment to $y$. So we are looking for the following code pattern:

$$x = F(a);\ S;\ y = E; \tag{1}$$

where $x$ and $a$ are not redefined in $S$. Now find the expression (if it exists) $G$ such that $G(F(a)) = a$ and then rewrite the assignment to $y$ as

$$y = E + a - G(x);$$

This rewrite is not suitable for our sum/product example as the assignment to $x$ (which is $x = x+i$) also includes a use of $x$. But we can store the old value of $x$ in a temporary variable and so we could write (for some fresh variable $t$)

$$\begin{aligned} &t = x;\\ &x = t + i;\\ &y = y * i + x - t - i; \end{aligned}$$

But since both assignments to $x$ and $y$ include a use of $i$ we can write the assignments as follows:

$$\begin{aligned} &t = x;\\ &x = t + i;\\ &y = y * (x - t); \end{aligned}$$

The slice from $s$ now contains the assignments to $x$ (and $t$).

## *B. Adding a bogus predicate*

To include a statement $x = G$ in a slice for $y$ we can transform it to

$$\begin{aligned} &x = G;\\ &\textbf{if } (p^F)\ y = H(x); \end{aligned}$$

where $p^F$ is a false predicate and $H$ is an expression depending on $x$. As we appear to have set up that the definition of $y$ depends on $x$ then the statement $x = G$ will be included in the slice for $y$. Another possibility is the following transformation:

$$\begin{array}{lcl} x = G; & \Rrightarrow & x = G; \\ S; & & \textbf{if } (q^T)\ S;\ \textbf{else } y = H(x); \end{array}$$

where $S$ is a statement and $q^T$ is a true predicate.

For our sum/product example, let us perform the following transformation:

$$x = x+i;\ \Rrightarrow\ \textbf{if } (i < 5 \mid\mid x < y)\ x = x+i;\ \textbf{else } y = x*i;$$

In Figure 3 we can see that after slicing the whole method for $y$ we will be left with the statement **out**($x$).

How did we produce the predicate above? Since $x = sum(1..i)$ and $y = prod(1..i)$ then for $i \geq 4$ we have $x < y$. Immediately before this expression we have $i++$ and so we

```
int bogus(int n) {
  int i = 0;
  int x = 0;
  int y = 1;
  while (i < n) {
    i++;
    if (i < 5 || x < y)
    x = x + i;
    else y = x * i;
    y = y * i;         }
  out(x);
  out(y);          }
```

Fig. 3. Adding a bogus predicate (with the slice from **out**($y$))

need to have $i < 5$ rather than $i < 4$. To construct predicates we can use properties of the program (such as invariants) that the programmer knows but may be difficult for an attacker to find out.

## *C. Variable Encoding*

Another data obfuscation is a variable encoding [6] which transforms a variable $v$ into the expression $\alpha * v + \beta$ where $\alpha$ and $\beta$ are constants. This particular transformation is not very useful for creating dependencies but we can adapt it so that $v$ is transformed to $F(v, w)$ where $F$ depends on $v$ and another variable $w$. For example, we can take $F(v, w) = \alpha * v + \beta * w$. Applying this kind of transformation means that $v$ will be dependent on $w$.

How can we apply this transformation? Suppose that $v \Rrightarrow \alpha * v + \beta * w$ and we will assume that we have exact arithmetic and that our variables (and constants) are integers. We transform a use of $v$, say $U(v)$, to $U(\frac{v - \beta * w}{\alpha})$. An assignment to $v$ is transformed as follows:

$$\begin{aligned} v = E;\ \Rrightarrow\quad & v = \alpha * E' + \beta * w;\\ & \text{where } E' = E[\tfrac{v - \beta * w}{\alpha}/v] \end{aligned} \tag{2}$$

A use of $w$ is left unchanged but since $v$ now depends on $w$ whenever we define $w$ we have to define $v$ as well. We use a fresh temporary variable $t$ to store the original value of $v$ and so we transform an assignment to $w$ as follows:

$$w = U(v);\ \Rrightarrow \left\{ \begin{aligned} &t = \tfrac{v - \beta * w}{\alpha};\\ &w = U(t);\\ &v = \alpha * t + \beta * w; \end{aligned} \right\} \tag{3}$$

We need to exhaustively apply these transformations to the whole method.

For our sum/product method, let us transform $y$ so that it becomes $y + x$. The initialisations of $x$ and $y$ stay the same and the output becomes **out**($y - x$). Using Equation (3) the assignment for $x$ in the loop becomes

$$\begin{aligned} &\textbf{int } t = y - x;\\ &x = x + i;\\ &y = t + x; \end{aligned}$$

Using Equation (2), the assignment to $y$ becomes:

$$y = (y - x) * i + x;$$

```
int varEnc(int n) {
  int i = 0;
  int x = 0;
  int y = 1;
  while (i < n) {
    i++;
    x = x + i;
    y = (y + i - x) * i + x;          }
  out(x);
  out(y - x);          }
```

Fig. 4. Adding a variable encoding (with the slice from **out**$(y-x)$)

So, putting these transformations together we have

$$\begin{array}{rcl} & & \textbf{int } t = y - x; \\ & & x = x + i; \\ x = x + i; & \Rightarrow & y = t + x; \\ y = y * i; & & y = (y - x) * i + x; \end{array}$$

We can remove the temporary variable $t$ and so:

$$\begin{array}{rcl} x = x + i; & \Rightarrow & x = x + i; \\ y = y * i; & & y = (y + i - x) * i + x; \end{array}$$

This transformation has the advantage that it does not rely on any opaque predicates which are usually hard to manufacture [7]. The transformed method can be seen in Figure 4, and we show the backwards slice from **out**$(y-x)$.

## *D. Adding to the guard of a* ***while*** *loop*

Since we have a **while** loop in our example we can add predicates to the guard to create dependencies. We have two choices:

$$\textbf{while } (c)\ S \Rrightarrow \textbf{while } (c\ \&\&\ p)\ S$$
$$\textbf{while } (c)\ S \Rrightarrow \textbf{while } (c\ ||\ q)\ S$$

provided that $c \Rightarrow p$ (*i.e.* $p$ is true when $c$ is true) and $\neg c \Rightarrow \neg q$ — this last condition can be weakened so that the first time $c$ is false (*i.e.* when the loop exits) then $q$ must also be false.

For our sum/product method, we can add a predicate to the guard which depends on $x$ and thus in the **while** loop $y$ will depend on $x$. So, for example

$$\textbf{while } (i < n)\ S \Rrightarrow \textbf{while } (i < n\ ||\ x < 0)\ S$$

We can also add a new, fresh variable $j$ to the loop with which we can create dependencies on $x$ and $y$. Let us suppose that we add the statement $j = j + y - x$ into the loop. Before the loop, we initialise $j$ by adding the statement **int** $j = 2$. With this initial value, we can prove that the value of $j$ is always positive in the loop and so we change the header to

$$\textbf{while } (i < n\ \&\&\ j > 0)$$

The full method can be seen in Figure 5 (including the slice from **out**$(y)$). This transformation has the advantage that it adds dependencies to both $x$ and $y$ and so slicing for $x$ (or $y$) removes everything but the output statement for $y$ (or $x$).

```
int addVar (int n) {
  int i = 0;
  int x = 0;
  int y = 1;
  int j = 2;
  while (i < n && j > 0) {
    i++;
    x = x + i;
    y = y * i;
    j = j + y - x;          }
  out(x);
  out(y);          }
```

Fig. 5. Adding a new loop variable (with slice from **out**$(y)$)

## VI. Conclusions and Future Work

In this paper we have proposed a novel way to design obfuscating transforms by considering the maxim: "Attack is the best form of defence". We attacked our programs by computing the backwards slices for the output variables and we considered the statements that were left behind after slicing — we called these the *orphaned* statements. Once we have computed the slices of a particular program we then try to transform the program so that we decrease the number of orphaned statements. Our ideal obfuscation for a particular program will minimise the number of orphaned statements for all output variables.

To evaluate our slicing obfuscations we have used some slicing metrics given in [17]. These slice-based metrics consider the sizes of the slices and the intersection of the slices. We can see that for our examples we always increase the size of the slice for $y$ and as we add dependencies between the variables we also increase the size of the intersection. It is beyond the scope of this paper to give these results but, in general, our obfuscations increase these metrics and so decrease the effectiveness of slicing.

When manufacturing slicing obfuscations we aim to create dependencies between variables by rewriting assignments or creating new ones. Where should we place such assignments? One area for future work is develop heuristics enabling to decide where to place slicing obfuscations in order to maximise the effects of the transforms.

In our sum/product example we focused on slices for the variable $y$ but, in general, we should ensure that we try to defend against backwards slices for all output variables. Thus we should aim to obfuscate our programs so that the number of orphaned statements for all output variables decreases. But how do we manufacture such an obfuscation? The loop transformation given in Section V-D managed to change the program so that a slice for either of the output variables contains everything but the output statements — but as this transformation relies on loops, it will not possible in general. What we could do is to concentrate on just one variable initially and add obfuscations until the slice with respect to this variable produces a minimal set of orphaned statements. Then we consider slicing our obfuscated program for another output variable and add in obfuscations accordingly. We can continue this process until we have covered the entire set of output variables — in the Appendix we sketch out a possible algorithm for producing slicing obfuscations. Note that we should make sure that any new obfuscations do not affect the slices for previous variables — if this is case then we may have to add in extra obfuscations. In following this method we will create an obfuscation that consists of many obfuscations composed together. For future work we will 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?

## 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. An empirical study of predicate dependence levels and trends. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 330–339, Washington, DC, USA, 2003. IEEE Computer Society.

[4] 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.

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

[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] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.

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

[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] PreEmptive Solutions Inc. Dotfuscator. URL: www.preemptive.com/products/dotfuscator.

[13] Ganeshan Jayaraman, Venkatesh Prasad Ranganath, and John Hatcliff. Kaveri: Delivering the Indus Java program slicer to Eclipse. In *FASE*, pages 269–272. Lecture Notes In Computer Science, SpringerVerlag, 2005.

[14] Mark Ladue. When Java was one: Threats from hostile bytecode. In *Proceedings of the 20th NIST-NCSC National Information Systems Security Conference*, pages 104–115, 1997.

[15] Anirban Majumdar, Antoine Monsifrot, and Clark Thomborson. On evaluating obfuscatory strength of alias-based transforms using static analysis. In *ADCOM 2006: Proceedings of the 14th International Conference on Advanced Computing and Communication (ADCOM 2006)*, Mangalore, India, 2006. IEEE Computer Society.

[16] 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.

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

[18] Juergen Rilling and Tuomas Klemola. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 115–124, Washington, DC, USA, 2003. IEEE Computer Society.

[19] Nuno Santos, Pedro Pereira, and Luís Moura e Silva. A Generic DRM Framework for J2ME Applications. In Olli Pitkänen, editor, *First International Mobile IPR Workshop: Rights Management of Information (MobileIPR)*, pages 53–66. Helsinki Institute for Information Tecnhology, August 2003.

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

## Appendix

*Algorithm:* Here is a possible algorithm for manufacturing slicing resistant obfuscations.

(1) Let $\mathcal{V}$ be the set of variables to be sliced and $\mathcal{P}$ be the slicing points (representing an output for each variable).

(2) Pick a variable $v \in \mathcal{V}$ and the corresponding slicing point $P_v$ from $\mathcal{P}$.

(3) Slice the program at $P_v$ with respect to $v$. Consider the sequence of statements $\mathcal{S}_v$ that are orphaned (*i.e.* not contained in the slice). If $\mathcal{S}_v$ is minimal then repeat Step (2) with a different variable.

(4) Choose one or more of the transformations (a) to (d) below. Transformations should be applied until $\mathcal{S}_v$ is minimal. Once a transformation is applied the program can be re-sliced using Step (3).

   (a) Find a variable definition that matches the code pattern in Equation (1) and rewrite an expression for $v$ (see Section V-A).

   (b) Look at any variable assignments in $\mathcal{S}_v$ and add a bogus predicate after the assignment (see Section V-B).

   (c) Find a variable $w$ from $\mathcal{V}$ that is defined in $\mathcal{S}_v$ and encode $v$ so that it depends on $w$. From Section V-C, we need to choose an expression $F$ so that $v \Rrightarrow F(v, w)$.

   (d) If a statement in $\mathcal{S}_v$ is contained within a loop then apply one of the loop transformations given in Section V-D.

(5) Steps (2) and (3) should be applied until the backwards slice from the output point for each variable in $\mathcal{V}$ produces a minimal number of orphaned statements.