Slicing Obfuscations: Design, Correctness, and Evaluation

Anirban Majumdar Stephen Drape Clark Thomborson

Secure Systems Group
Department of Computer Science
The University of Auckland, New Zealand.
{anirban,stephen,cthombor}@cs.auckland.ac.nz

ABSTRACT

The goal of obfuscation is to transform a program, without affecting its functionality, such 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 form of reverse engineering which aims to abstract away a subset of program code based on a particular program point and is considered to be a potent program comprehension technique. Thus, slicing could be used as a way of attacking obfuscated programs. It is challenging to manufacture obfuscating transforms that are provably resilient to slicing attacks.

We show in this paper how we can utilise the information gained from slicing a program to aid us in designing obfuscations that are more resistant to slicing. We extend a previously proposed technique and provide proofs of correctness for our transforms. Finally, we illustrate our approach with a number of obfuscating transforms and provide empirical results using software engineering metrics.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.1 [**Software Engineering**]: Specifications—*Languages*; D.2.8 [**Software Engineering**]: Metrics; D.3 [**Software**]: Programming Languages; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; E.1 [**Data**]: Data Structures; F.3 [**Theory of Computation**]: Logics and Meanings of Programs; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Design, Experimentation, Human Factors, Languages, Legal Aspects, Measurement, Security, Theory

Keywords

Obfuscation, Static Slicing, Program Transformation, Software Security, Digital Rights Management

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 tools or becomes unanalysable to a human adversary interpreting the output of program analyses run on the obfuscated application [19]. The motivation for protecting software through obfuscation arises from the problem of software piracy, which can be summarised as a reverse engineering process undertaken by a software pirate when stealing intellectual artefacts (such as a patented algorithm) from a commercially valuable software to make derivative software or tampering DRM routines in order to bypass license authentication checks. The 2005 annual Global Software Piracy Report [1] from Business Software Alliance (BSA) stated that "35% percent of the packaged software installed on personal computers (PC) worldwide in 2005 was illegal, amounting to $34 billion in global losses due to software piracy". This is one of the primary reasons why commercially popular software such as the Skype VoIP client [6], the SDC Java DRM [23], and most license-control systems rely, at least in part, on obfuscation for their security.

Collberg *et al.* [7, 8] were the first to formally define obfuscation 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.* [3] strengthened the formalism 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 $\mathcal{P}$, 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*. Thinking in terms of a *virtual black-box*, an obfuscation function is a *failure* if there exists at least one program that cannot be completely obfuscated by this function, that is, if an adversary could learn something from an examination of the obfuscated version of this program that cannot be learned (in roughly the same amount of time) by merely executing this program repeatedly. This negative result established that every obfuscator will fail to completely obfuscate some programs. Drape in [11] observed that the *virtual black-box* property is too strong. Obfuscators will be of practical use even if they do not provide perfect black boxes. Therefore, the focus has now shifted to designing obfuscations that are *difficult* (but not necessarily *impossible*) for an adversary to reverse engineer.

In the domain of software engineering, program slicing is widely used for program maintenance, modularisation, and comprehension [5, 15, 22, 24]. These techniques form the basis of reverse engineering since the primary goal of these techniques is to identify *relevant/interesting* parts of the code and create representations of the program at a higher level of abstraction. Indeed, this is what a software pirate also intends to do when he/she attempts to steal or change some relevant part (of her/his interest) of the code with the intention of reusing it in illegal derivative software or invalidating the code licensing routine.

It would seem obvious that a natural way to deter such *code comprehension attacks* is to intertwine the relevant code routine with other irrelevant sections such that the attacker fails to recognise the portions of interest by analysing the code. Drape and Majumdar in [14] made the first attempt to intertwine code in a way so that static slicing attacks could be made difficult (if not impossible). In this contribution, we extend the work in [14] along many directions — first, we show a way to prove correctness of our proposed slicing obfuscations using program refinement techniques [10]. Secondly, we empirically evaluate our obfuscating transforms using a popular static program slicer, the CodeSurfer, and show that the results, in fact, corroborate with the claims in [14]. In the process of evaluating our transforms, we also propose our own metrics (derived from the original slicing metrics) and interpret our results in terms of the derived metrics.

2. EXPERIMENTAL DESIGN

The motivation of this paper is derived from the difficulty of empirically evaluating the obfuscatory strength of seemingly resilient obfuscating transforms. Majumdar *et al.* [18] (and separately by Udupa *et al.* [25]) observed that in order to evaluate resilience of obfuscating transforms, 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?" They argued that developing one general purpose reverse engineering tool for deobfuscating *all* possible obfuscating transforms is inconceivable (an adversary will have to use different techniques to deobfuscate a program obfuscated with different kinds of transforms). Consequently, we can also assume that it is ambitious to design an obfuscating transform that will withstand *all* possible reverse engineering attacks.

We use CodeSurfer, a static program slicer for code written in C [2], for evaluating our slicing obfuscations. It runs algorithms on system dependence graphs (SDGs), an intermediate graph structure with annotated data and control dependency nodes, for representing programs [16]. Slicing using SDGs is the most precise and complete slicing method currently available [24]. 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 experiments in the paper, we restrict ourselves to only backward computation of slices (to be explained in Section 2.1).

It has been claimed in the existing literature [8] that obfuscating transforms which are not derived from hard complexity problems are easy to undo. We set ourselves to substantiate this rather suppositional claim in this contribution. Moreover, 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.

Mobile code (such as Java byte code, Microsoft's MSIL) is considered more vulnerable to reverse engineering attacks than binary executables. Nevertheless, we have designed our experiments using the constructs of the C language because 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 [18]. The only well-known static slicer for Java programs is Indus [17]; 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.* [16]) and has been extensively evaluated in published literature [4, 5, 20].

2.1 Slicing Notation

To denote the slices of a particular method we use the notation of Ott and Thuss [21] as follows. 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 and thus we will use backward slicing for evaluation. 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 $|\dots|$ denotes the size. We will compute the size by considering the number of nodes in the SDG (and CodeSurfer allows us to do this).

Suppose that we apply an obfuscation $\mathcal{O}$ to the program P to obtain $\mathcal{O}(P)$, what happens to the slicing criterion? Since we will use data refinement techniques [10] to define our obfuscations we have a mapping between the variables in P and $\mathcal{O}(P)$ and so we can write $\mathcal{O}(V_O)$ to denote the set of output variables in $\mathcal{O}(P)$. For the slicing point, we again use the last output statement for each variable.

2.2 Orphans and Residues

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*. What do we mean by "less useful"? One definition could be that an obfuscation creates larger slices. We can arbitrarily increase the size of a method and its slices by creating an obfuscation that adds bogus statements that are contained in the slice. Unfortunately, such simple obfuscations will not affect any of the program points left out of the original slice. A more suitable obfuscation would try to increase the size of the slice by including more of the program points that are left behind. We call the points that are left behind after slicing as the *orphaned* points. For each $v_i \in V_O$ we can define:

$$residue(M, v_i) = M \backslash SL_i$$

So the *residue* of a slice is defined to be the set of points that are orphaned. Using this concept, we can define a slicing obfuscation as follows:

Definition. An obfuscation $\mathcal{O}$ is a **slicing obfuscation** for a program P and a variable V_i if it decreases the size of the residue (the number of orphaned points), *i.e.*

$$|residue(P, v_i)| > |residue(\mathcal{O}(P), \mathcal{O}(v_i))|$$

2.3 Metrics

Using inspiration from the *Tightness*, *MinCoverage*, *Coverage* and *MaxCoverage* metrics given in [21], we can define four residue metrics for a method M. We denote RES_{un} to be the union of all the residues, *i.e.*

$$RES_{un} = \bigcup_{i=1}^{|V_O|} residue(M, v_i)$$

In fact, $RES_{un} = M \backslash SL_{int}$.

Compactness. Compactness measures the total number of orphaned points in relation to the size of the method.

$$C(M) = \frac{|RES_{un}|}{|M|}$$

MinDensity. The minimum density is the ratio of the smallest residue in a method to the method length.

$$MinD(M) = \frac{1}{|M|} \min_i |residue(M, v_i)|$$

Density. Density compares the average residue size to the method size.

$$D(M) = \frac{1}{|V_O|} \sum_{i=1}^{|V_O|} \frac{|residue(M, v_i)|}{|M|}$$

MaxDensity. The maximum density is defined to be the ratio of the largest residue in a method to the method's length.

$$MaxD(M) = \frac{1}{|M|} \max_i |residue(M, v_i)|$$

Comparing our metrics to those in [21], we find that

$$\begin{array}{rcl} C(M) & = & 1 - Tightness(M) \\ MinD(M) & = & 1 - MaxCoverage(M) \\ D(M) & = & 1 - Coverage(M) \\ MaxD(M) & = & 1 - MinCoverage(M) \end{array}$$

and by the definitions, we see that

$$MinD(M) \leq D(M) \leq MaxD(M) \leq C(M)$$

These metrics give values in the range between 0 and 1 (so we give our values as percentages) and our aim when obfuscating will be to make these metrics as low as possible (*i.e.* to have as few orphaned points as possible). In Table 1, we use our residue metrics to evaluate the effectiveness of some obfuscating transforms (which are explained in detail later).

3. PROOF FRAMEWORK

In [11] a framework for proving the correctness of obfuscations for abstract data-types using functional refinement [10] was given. Suppose that a data-type D is obfuscated to produce a data-type E. Under this framework, an abstraction function $af :: E \rightarrow D$ and a data-type invariant dti are needed such that, for $x :: D$ and $y :: E$:

$$x \rightsquigarrow y \iff (x = af(y)) \wedge dti(y) \quad (1)$$

The term $x \rightsquigarrow y$ is read as "x is obfuscated by y".

For a function $f :: D \rightarrow D$, an obfuscated function $\mathcal{O}(f)$ is correct with respect to f if it satisfies:

$$(\forall x :: D;\ y :: E)\ x \rightsquigarrow y \Rightarrow f(x) \rightsquigarrow \mathcal{O}(f)(y)$$

Using Equation (1) we can rewrite this as

$$f \cdot af = af \cdot \mathcal{O}(f) \quad (2)$$

If we have a conversion function $cf :: D \rightarrow E$ that satisfies $af \cdot cf = id$ then we can rewrite Equation (2) to obtain:

$$f = af \cdot \mathcal{O}(f) \cdot cf$$

In the following subsections we will briefly discuss the use of this proof framework for proving correctness of our imperative obfuscations.

3.1 Modelling statements as functions

We model *statements* to be functions on states and so a statement has the following type: $statement :: state \rightarrow state$ where a *state* is defined to be a set of mappings from variables to values (or expressions computing values). We assume that the variables are integer valued and all expressions consist of arbitrary-precision arithmetic operators. We concentrate on code fragments with no methods, exceptions or pointers.

Suppose that we have a set of states $\mathcal{S}$. For some initial state $\sigma_0 \in \mathcal{S}$ and some statement T, the effect of statement T on σ_0 is to produce a new state $\sigma_1 \in \mathcal{S}$ such that $\sigma_1 = T(\sigma_0)$. Suppose that we have a sequential composition (;) of statements, which we will call a *block*, $B = T_1; T_2; \ldots; T_n$. If the initial state is σ_0 then the final state σ_n is given by

$$\sigma_n = B(\sigma_0) = T_n(\ldots T_2(T_1(\sigma_0))\ldots)$$

For our simple language, we consider the following statement types: *skip*, *assignments* (*var* = *expr*), *conditionals* (**if** *pred* **then** *statements* **else** *statements*) and *loops* (**while** *pred* **do** *statements*). So given an initial state, how can we compute the final state after executing each of these statement types?

The statement *skip* does not change the state and so if $S \equiv skip$ then $S(\sigma_0) = \sigma_0$. For an assignment A of the form $A \equiv x = e$, if the initial state σ_0 contains the mapping $x \mapsto x_0$ then the state after the assignment can be written as

$$A(\sigma_0) = \sigma_0 \oplus \{x \mapsto e[x_0/x]\}$$

using functional overriding ($\oplus$) and substitution (/).

Now suppose we have conditional statement C which has the form $C \equiv$ **if** p **then** T **else** E where p is a predicate with type $p :: state \rightarrow \mathbb{B}$ and T and E are blocks. Then for some initial state σ_0 we have that

$$C(\sigma_0) = \begin{cases} T(\sigma_0) & \text{if } p(\sigma_0) \\ E(\sigma_0) & \text{otherwise} \end{cases}$$

A loop statement L has the form $L \equiv$ **while** p **do** T where p is a predicate and T is a block. Then for some initial state σ_0 we have that

$$L(\sigma_0) = T^i(\sigma_0) \text{ where } i = min\{n :: \mathbb{N} \,|\, p\,(T^n(\sigma_0)) = False\}$$

Note that this minimum does not exist if the loop fails to terminate.

3.2 Using the refinement framework

We suppose that a data obfuscation acts on a state to produce a new state. To specify a data obfuscation $\mathcal{O}$ we will supply an abstraction function $af :: state \rightarrow state$ and a conversion function $cf :: state \rightarrow state$ which is a pre sequential inverse for af (*i.e.* $cf; af \equiv skip$). As well an abstraction function, for refinement, we need an invariant I on the obfuscated state. Note that for most of our transformations unless otherwise stated $I \equiv True$. The abstraction and conversion functions will usually consist of assignment statements.

Since our statements are functions on the state, we can adapt the framework from [11] to produce correctness equations. Suppose that we have a block B and we want to obfuscate it using data refinement to obtain a block $\mathcal{O}(B)$ which preserves the correctness of B. Then using Equation (2), $\mathcal{O}(B)$ is correct with respect to B if it satisfies

$$af;\, B \equiv \mathcal{O}(B);\, af \qquad (3)$$

By writing the abstraction function as a statement we can construct two blocks $af; B$ and $\mathcal{O}(B); af$ and proving the equivalence of these blocks establishes that $\mathcal{O}(B)$ is correct.

Using the conversion function we can obtain another correctness equation:

$$B \equiv cf;\, \mathcal{O}(B);\, af \qquad (4)$$

3.3 Obfuscating Statements

Suppose that we have a data obfuscation that changes a variable x using an abstraction function af and a conversion function cf satisfying $cf;\, af \equiv skip$. Then af and cf can be written as statements of the form:

$$af \equiv x = G(x) \qquad cf \equiv x = F(x)$$

for some functions F and G.

Suppose we have an obfuscation for x (with af and cf defined as above) then let us consider the statement $P_1 \equiv x = E$ where E is an expression that may contain an occurrence of x. It can be obfuscated as follows:

$$\mathcal{O}(x = E) \equiv x = F(E') \text{ where } E' = E[G(x)/x] \qquad (5)$$

For example, the expression $x = x+1$ would be transformed to $x = F(G(x) + 1)$. Note that the expression $E[G(x)/x]$ denotes how a use of x is obfuscated.

Now let us suppose that $P_2 \equiv$ **if** p **then** T **else** E for some predicate p and blocks T and E. We propose that

$$\mathcal{O}(P_2) \equiv \textbf{if } p[G(x)/x] \textbf{ then } \{af; T; cf\} \ \textbf{else } \{af; E; cf\} \qquad (6)$$

with af as above. Using Equation (3) we can show that $\mathcal{O}(P_2);\, af \equiv af;\, P_2$.

Finally, suppose that $P_3 \equiv$ **while** p **do** S then, with af as above, we propose that

$$\mathcal{O}(P_3) \equiv \textbf{while } p[G(x)/x] \textbf{ do } \{af; S; cf\} \qquad (7)$$

and we can use Equation (3) to prove that this is correct.

Suppose that we want to obfuscate a sequential composition of blocks. Let B_1 and B_2 be two blocks of code then we have:

$$\mathcal{O}(B_1;\, B_2) \equiv \mathcal{O}(B_1);\, \mathcal{O}(B_2)$$

So when applying a data obfuscation to a sequence of statements (blocks) we can obfuscate each statement (block) individually and compose the results.

3.4 Simultaneous Equations

Suppose that we obfuscate S to obtain $\mathcal{O}(S)$ with abstraction and conversion functions af and cf for the obfuscation. We can use Equation (4) to prove that $\mathcal{O}(S)$ is a correct obfuscation of S by showing that the sequence of statements $cf;\, \mathcal{O}(S);\, af$ is equivalent to S. Suppose that we have an obfuscation that transforms a variable x (say) then this proof could take the form:

$$x := f(x); \quad x = u(x); \quad x = g(x)$$

where f, g and u are functions. To simplify this expression we can substitute values of x in sequential order by rewriting the sequence of statements as a set of simultaneous equations. Each definition of a variable will have a different name which is usually the name of the variable with a subscript (*e.g.* x_2) and we will use the convention that the initial value of a variable has a subscript 0. All the uses of a variable are renamed to correspond to the appropriate assignment.

The sequence above can be rewritten as the following set of equations:

$$x_1 = f(x_0); \quad x_2 = u(x_1); \quad x_3 = g(x_2)$$

By substituting the values for x_1 and x_2 we obtain the following:

$$x_1 = f(x_0); \quad x_2 = u(f(x_0)); \quad x_3 = g(u(f(x_0)))$$

We can remove the assignments for x_1 and x_2 as they are now redundant. So now we have $x_3 = g(u(f(x_0))$ which corresponds to the statement $x = g(u(f(x)))$.

This conversion from assignments to simultaneous equations is similar to converting code to SSA (Static Single Assignment) form which is often used in conjunction with compiler optimisations (for example, [9] gives details about how to compute SSA form). In SSA form, each definition of a variable is given a different name and each use is renamed according to the appropriate definition. When there are different control flow paths, a special statement called a ϕ (phi) function is added. However, as we are only aiming to simplify a set of simultaneous equations, we will not use the SSA form directly. In particular, our proofs will not need to use phi functions as we will use the results of Section 3.3 to enable us to deal with **if** and **while** separately and we can obfuscate a sequence of statements by obfuscating the individual statements. We will only use the SSA form as a guide to help us to specify a set of simultaneous equations which we can manipulate and simplify.

4. TRANSFORMATIONS

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 a running example.

```
$\underline{original()}$ {
  $\underline{\textbf{int } c, nl = 0}, nw = 0, nc = 0, \underline{in}$;
  $in = F$;
  $\underline{\textbf{while } ((c = \textbf{getchar}())\,!= \textbf{EOF})}$ {
    $nc\,{+}{+}$;
    if $(c ==' \ ' \parallel c ==' \backslash n' \parallel c ==' \backslash t')\ in = F$;
    else if $(in == F)\ \{in = T;\ nw\,{+}{+};\}$
    $\underline{\textbf{if } (c ==' \backslash n')\ nl\,{+}{+};}$          }
  $\underline{\textbf{out}(nl, nw, nc)}$;    }
```

Figure 1: Method to calculate the number of lines, words and characters in a piece of text (the backwards slice for nl in indicated by underlined points).

```
$\underline{bogus()}$ {
  $\underline{\textbf{int } c}, \underline{nl = 0}, \underline{nw = 0}, \underline{nc = 0}, \underline{in}$;
  $\underline{in = F}$;
  $\underline{\textbf{while } ((c = \textbf{getchar}())! = \textbf{EOF})}$ {
    $\underline{nc\,{+}{+};}$
    $\underline{\textbf{if } (c ==' \ ' \parallel c ==' \backslash n' \parallel c ==' \backslash t')\ in = F}$;
    else $\underline{\textbf{if } (in == F)}\ \{\underline{in = T};\ \underline{nw\,{+}{+}};\}$
    $\underline{\textbf{if } (c ==' \backslash n')\ \{\textbf{if } (nw <= nc)\ nl\,{+}{+};\}}$
    $\underline{\textbf{if } (nl > nc)\ nw = nc + nl}$;
    else $\{\underline{\textbf{if } (nw > nc)\ nc = nw - nl};\}$          }
  $\underline{\textbf{out}}(nl, nw, \underline{nc)};$    }
```

Figure 2: Addition of a bogus predicate (with the backwards slice for nc).

4.1 Word Count Example

Our running example will be the Word Count program which takes in a block of text and outputs the number of lines (nl), words (nw) and characters (nc) and so, for this example, $V_O = \{nl, nw, nc\}$. The method can be seen in Figure 1. Note that we write **out**(nl, nw, nc) as a shorthand for the three **printf** statements contained in the actual method.

Our slicing criteria will be the output statement and one of the three output variables. In Figure 1 the underlined points denote the backwards slice from nl given by CodeSurfer. We can see that the residue for nl contains program points for nc and nw. The aim of our obfuscations will be to create dependencies between the three output variables and so decrease the sizes of the residues.

Using the Word Count example we discuss several techniques to reduce the size of the residues and thus decrease the effectiveness of slicing. In Table 1 we summarise the results of our different transformations on the Word Count example. Each row of the table contains the results for a particular experiment where we record the size of the method, the size of each slice (calculated using CodeSurfer), the size of each residue and the results for our four residue metrics from Section 2.3. The top row of the table displays the measurements for the original Word Count method.

4.2 Adding a bogus predicate

Suppose that the residue for a variable y contains an assignment for another variable x. To include a statement $x = G$ in a slice for y we can transform it to

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

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:

$$\begin{array}{l} x = G; \\ S; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} x = G; \\ \textbf{if } (q^T)\ S;\ \textbf{else } y = H(x); \end{array} \tag{8}$$

where S is a statement and q^T is a true predicate. To prove that this is a correct transformation, we can go back to our statement models from Section 3.1. We can see that with an initial state of σ_0 and an initial value x_0 for x the final state for both blocks in Equation (8) is:

$$S(\sigma_0 \oplus \{x \mapsto G[x_0/x]\})$$

For Word Count, we added the following bogus predicates

$$\begin{array}{l} \textbf{if } (c ==' \backslash n')\ \{\textbf{if } (nw <= nc)\ nl\,{+}{+};\} \\ \textbf{if } (nl > nc)\ nw = nc + nl; \\ \quad \textbf{else } \{\textbf{if } (nw > nc)\ nc = nw - nl;\} \end{array}$$

at the end of the **while** loop. These predicates add dependencies between the three variables and so the slice for each variable contains the definitions for the other variables. In Figure 2 we can see this method with the backwards slice from nc. Since nc is incremented for every character and nw and nl are only incremented for certain characters, we have the following invariant

$$nc \geq nw \wedge nc \geq nl \tag{9}$$

and so our predicates leave the values of the three output variables unchanged.

In Table 1 we can see for the *bogus* method that we increase the method size by 19% but we significantly decrease the size of the residues and thus the slice sizes are increased.

4.3 Variable Encoding

A variable encoding [7] is a data obfuscation which transforms a variable x into the expression $\alpha * x + \beta$ where α and β are constants. This particular transformation is not very useful for creating dependencies but we can adapt it so that x is transformed to $F(x, y)$ where y is another variable. For example, we can take $F(x, y) = \alpha * x + \beta * y$. Applying this kind of transformation means that x will be dependent on y.

For this data transformation, the abstraction function is

$$af \equiv x = (x - \beta * y)/\alpha$$

and the conversion function is

$$cf \equiv x = \alpha * x + \beta * y$$

Note that, this obfuscation is not suitable if multiplication can overflow and division is not exact.

For this transformation we have three rewrite rules. We transform a use of x, say $U(x)$, to $U(\frac{x-\beta*y}{\alpha})$. Using Equation (5) an assignment to x is transformed as follows:

$$x = E \rightsquigarrow x = \alpha * E' + \beta * y \text{ where } E' = E[\frac{x - \beta * y}{\alpha}/x]$$

Our obfuscated value for x depends on y; thus whenever we have a definition of y then we will also need a definition of x as well. Suppose that we want to transform the statement $B \equiv y = f(x)$ where f is a function (which could depend on other variables including y).

Method M	$\lvert M\rvert$	$\lvert V_O\rvert$	Slice Size			$\lvert SL_{int}\rvert$	Residue size			$\lvert RES_{un}\rvert$	$C(M)$	$MinD(M)$	$D(M)$	$MaxD(M)$
			nl	nw	nc		nl	nw	nc					
original	32	3	11	18	10	6	21	14	22	26	81.3%	43.8%	59.4%	68.8%
bogus	38	3	30	30	30	29	8	8	8	9	23.7%	21.1%	21.1%	21.1%
*encode*1	35	3	11	18	28	6	24	17	7	29	82.9%	20.0%	45.7%	68.6%
*encode*3	42	3	34	34	32	31	8	8	10	11	26.2%	19.0%	20.6%	23.8%
loop	36	3	29	29	29	28	7	7	7	8	22.2%	19.4%	19.4%	19.4%
split	44	3	35	35	35	34	9	9	9	10	22.7%	20.5%	20.5%	20.5%
array	28	3	21	21	21	20	7	7	7	8	28.6%	25.0%	25.0%	25.0%

Table 1: Table of results for the residues of all our Word Count examples

$$\begin{array}{l}
cf; \mathcal{O}(B); af \\
\equiv \quad \{\text{definitions}\} \\
\quad x = \alpha * x + \beta * y; \\
\quad t = x - \beta * y; \\
\quad y = f(t/\alpha); \\
\quad x = t + \beta * y; \\
\quad x = (x - \beta * y)/\alpha; \\
\equiv \quad \{\text{sim eqns}\} \\
\quad x_1 = \alpha * x_0 + \beta * y_0; \\
\quad t_1 = x_1 - \beta * y_0; \\
\quad y_1 = f(t_1/\alpha); \\
\quad x_2 = t_1 + \beta * y_1; \\
\quad x_3 = (x_2 - \beta * y_1)/\alpha;
\end{array}
\qquad
\begin{array}{l}
\equiv \{\text{sub } x_1,\ t_1 \text{ and } x_2\} \\
\quad x_1 = \alpha * x_0 + \beta * y_0; \\
\quad t_1 = \alpha * x_0; \\
\quad y_1 = f(x_0); \\
\quad x_2 = \alpha * x_0 + \beta * y_1; \\
\quad x_3 = x_0; \\
\equiv \quad \{\text{remove } x_1,\ t_1 \text{ and } x_2\} \\
\quad y_1 = f(x_0); \quad x_3 = x_0; \\
\equiv \quad \{\text{sequential code}\} \\
\quad y = f(x); \quad x = x; \\
\equiv \quad \{x = x \equiv skip\} \\
\quad y = f(x); \\
\equiv \quad \{\text{definition}\} \\
\quad B
\end{array}$$

Figure 3: Proof of correctness for a variable encoding

We propose that a suitable transformation is

$$\mathcal{O}(B) \equiv \left\{ \begin{array}{l} t = x - \beta * y; \\ y = f(t/\alpha); \\ x = t + \beta * y; \end{array} \right\}$$

where t is a fresh variable. In Figure 3, Equation (4) is used to show that:

$$cf; \mathcal{O}(B); af \equiv B$$

For Word Count, we will perform the following encoding:

$$nc \rightsquigarrow nc + nl - nw$$

By our rewrite rules, we can prove that, for example,

$$\begin{array}{rcl} nc{+}{+} & \rightsquigarrow & nc{+}{+} \\ nw{+}{+} & \rightsquigarrow & \{nw{+}{+};\ nc{-}{-};\} \end{array}$$

Before nc is output, we need to include the statement

$$nc = nc - nl + nw;$$

which is, of course, af for this transformation.

The full method for this transformation can be seen in Figure 4 with the backwards slice for nc indicated. We can see that we have successfully reduced the residue size for nc from 22 to 7 but we fail to do so for the other two outputs variables and, in fact, we increase the size of the residues. This highlights the importance of adding obfuscations for *all* of the output variables.

To create dependencies for *all* the variables, we can perform these three encodings:

$$nc \rightsquigarrow nc - nw \qquad nw \rightsquigarrow nw - nl \qquad nl \rightsquigarrow nl + nc$$

```
$\underline{encode1()}$ {
  $\underline{\textbf{int}\ c}, \underline{nl = 0}, \underline{nw = 0}, \underline{nc = 0}, \underline{in};$
  $\underline{in = F};$
  $\underline{\textbf{while}\ ((c = \textbf{getchar}())\ != \textbf{EOF})}$ {
    $\underline{nc{+}{+};}$
    $\underline{\textbf{if}}\ \underline{(c ==' \ ' \parallel c ==' \backslash n' \parallel c ==' \backslash t')\ in = F};$
    $\textbf{else}\ \underline{\textbf{if}}\ \underline{(in == F)}\ \underline{\{in = T};\ \underline{nw{+}{+}};\ \underline{nc{-}{-};\}}$
    $\underline{\textbf{if}}\ \underline{(c ==' \backslash n')\ \{nl{+}{+}};\ \underline{nc{+}{+}};\}$          }
  $\underline{nc = nc - nl + nw};$
  $\underline{\textbf{out}}(nl, nw, \underline{nc)};$     }
```

Figure 4: A simple variable encoding example (with the <u>backwards slice</u> for nc).

```
$\underline{encode3()}$ {
  $\underline{\textbf{int}\ c}, \underline{nl = 0}, \underline{nw = 0}, \underline{nc = 0}, \underline{in};$
  $\underline{in = F};$
  $\underline{\textbf{while}\ ((c = \textbf{getchar}())\ != \textbf{EOF})}$ {
    $\underline{nc{+}{+}};\ \underline{nl{+}{+}};$
    $\underline{\textbf{if}\ (c ==' \ ' \parallel c ==' \backslash n' \parallel c ==' \backslash t')\ in = F};$
    $\textbf{else}\ \underline{\textbf{if}}\ \underline{(in == F)}\ \{\underline{in = T};\ \underline{nw{+}{+}};\ \underline{nc{-}{-}};\ \underline{nl{-}{-}};\}$
    $\underline{\textbf{if}}\ \underline{(c ==' \backslash n')}\ \underline{\{nl{+}{+}};\ \underline{nw{-}{-};\}}$          }
  $\underline{\textbf{int}\ t = nl - nc};$
  $\underline{nc = nc + nw - t};$
  $\underline{nl = t + nc};$
  $\underline{nw = nw + nl - nc};$
  $nl = nl - nc;$
  $\underline{\textbf{out}}(nl, \underline{nw}, nc);$     }
```

Figure 5: Three variable encodings applied to our example (with the <u>backwards slice</u> for nw).

We apply these transformations in order starting with the one for nc. We use the rewrite rules given earlier and before the output statement we include the abstraction function for each transformation. After performing these encodings, in order, we obtain the method given in Figure 5. Note that we have had to use a temporary variable t (which is included in the slices) and, from Table 1, we can see that for *encode*3 the size has increased by 31%. However, we have reduced the sizes of the three residues and so the metrics values have significantly decreased.

4.4 Adding to the guard of a while loop

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

$$\begin{array}{l} \textbf{while}\ (c)\ S \rightsquigarrow \textbf{while}\ (c\ \wedge p)\ S \\ \textbf{while}\ (c)\ S \rightsquigarrow \textbf{while}\ (c\ \vee q)\ S \end{array}$$

```
loop() {
  int c, nl = 0, nw = 0, nc = 0, in, j = 0;
  in = F;
  while (((c = getchar()) != EOF) && (j >= 0)) {
    nc++;
    if (c == ' ' || c == '\n' || c == '\t') in = F;
    else if (in == F) {in = T; nw++;}
    if (c == '\n') nl++;
    j = nc + nl - nw;              }
  out(nl, nw, nc);   }
```

Figure 6: Adding a new loop variable to our example (with backwards slice from nw)

Under what conditions are these transformations valid?

If we add p as a conjunction then when c is $false$ then $c \wedge p$ will also be $false$. When c is $true$ then we also want $c \wedge p$ to be $true$, *i.e.*

$$\begin{array}{ll} & c \Rightarrow c \wedge p \\ \equiv & \{\text{distribution of } \Rightarrow\} \\ & (c \Rightarrow c) \wedge (c \Rightarrow p) \\ \equiv & \{\text{idempotence of } \Rightarrow\} \\ & true \wedge (c \Rightarrow p) \\ \equiv & \{\text{unit of } \wedge\} \\ & c \Rightarrow p \end{array}$$

Let us consider the conditions for adding the predicate q as a disjunction. When c is $true$ then $c \vee q$ is also $true$ but when c is $false$ we want $c \vee q$ to be $false$ as well, *i.e.*

$$\begin{array}{ll} & \neg c \Rightarrow \neg(c \vee q) \\ \equiv & \{\text{de Morgan's Law}\} \\ & \neg c \Rightarrow (\neg c \wedge \neg q) \\ \equiv & \{\text{distribution of } \Rightarrow\} \\ & (\neg c \Rightarrow \neg c) \wedge (\neg c \Rightarrow \neg q) \\ \equiv & \{\text{idempotence of } \Rightarrow\} \\ & true \wedge (\neg c \Rightarrow \neg q) \\ \equiv & \{\text{unit of } \wedge\} \\ & \neg c \Rightarrow \neg q \\ \equiv & \{\text{contrapositive}\} \\ & q \Rightarrow c \end{array}$$

For Word Count, we add a new, fresh variable j to the loop with which we can create dependencies on the three output variables. Let us suppose that we add the statement $j = nc + nl - nw$ into the loop. Before the loop, we initialise j by adding the statement **int** $j = 0$. By our invariant, Equation (9), we can see that the value of j is always non-negative in the loop and so we change the loop header to

$$\textbf{while } (((c = \textbf{getchar}()) \mathrel{!=} \textbf{EOF}) \ \&\& \ (j >= 0))$$

We then add our assignment for j into the loop. The full method can be seen in Figure 6. From Table 1, for *loop* we have only added 4 extra points to the method but the size of the residues are all decreased (for example, a 68% decrease for nc).

The addition of the guard does not change the **while** loop (as j is always non-negative) and the extra statement for j does not change the values of nl, nw or nc. Thus the transformation is correct.

4.5 Variable Splitting

A variable v can be split up to two or more variables, which we call the *split components*. The information contained in v is split across the components and since we know the relationship between the original variable and its components we can create invariants based on the split components.

Suppose that we split an integer variable v into two variables v_1 and v_2. Using [11], we can write the relationship between v and the split components as

$$v \leadsto \langle v_1, v_2 \rangle$$

We need three functions g (the abstraction function), f_1 and f_2 (the conversion functions) such that

$$v_1 = f_1(v) \qquad v_2 = f_2(v) \qquad v = g(v_1, v_2)$$

which satisfy $v = g(f_1(v), f_2(v))$ and is subject to the invariant $I(v_1, v_2)$.

A use of v, say $U(v)$, is replaced by $U(g(v_1, v_2))$. An assignment to v needs to be replaced by two assignments:

$$\begin{array}{c} v = E \leadsto \{v_1 = f_1(E');\ v_2 = f_2(E')\} \\ \text{where } E' = E[g(v_1, v_2)/v] \end{array} \qquad (10)$$

These transformations need to be applied exhaustively to the whole method (or at the very least to the whole scope of v).

As an example for Word Count, we split nc into two other integers — one containing the least significant digit and the other containing the rest of the digits. So we can define $nc \leadsto \langle nc / 10,\ nc \,\%\, 10 \rangle$ and let $a = nc / 10$ and $b = nc \,\%\, 10$ — we have that $0 \leq b \leq 9$ which we take to be our invariant. From above, $f_1 = \lambda i.\ i / 10$, $f_2 = \lambda i.\ i \,\%\, 10$ and $g = \lambda i, j.\ 10 * i + j$.

The first step is apply the transformation to nc. The statement $nc = 0$ becomes $a = 0;\ b = 0;$ and the output for nc is now $\textbf{out}(10 * a + b)$. To measure the size of the slice (and the residue) we will take backwards slice of $10 * a + b$ and in Table 1 the values for nc represents the values for the slice of a and b.

By using Equation (10) we can transform $S \equiv nc{+}{+}$ into:

$$a = a + ((b + 1)/10); \qquad b = (b + 1) \,\%\, 10;$$

However an equivalent version of $\mathcal{O}(S)$ is:

$$\textbf{if } (b == 9) \textbf{ then } \{a = a + 1;\ b = 0\} \textbf{ else } \{b = b + 1\}$$

and in Figure 7 we show this is correct by using Equation (4) to prove that $S \equiv cf; \mathcal{O}(S); af$. This transformation is more efficient as it does not need to use % or /.

In Figure 8 we can see the method after this transformation. In the method, we have added two bogus predicates as the variable split in isolation does not produce a very good slicing obfuscation. The first predicate uses our invariant on b to add in dependencies for nw and b. The other comes after the increment for nl where we add in a dependency on nl. From Table 1 we can see that for *split* we have reduced the residue sizes for the three output variables but we increased the method size by 38%. This size increase is due to the extra assignments and predicates needed for this transformation.

$$
\begin{array}{ll}
 & cf; \mathcal{O}(S); af \\
\equiv & \quad \{af; cf \equiv \mathit{skip}\} \\
 & cf;\ \textbf{if}\ (b == 9) \\
 & \qquad \textbf{then}\ \{af;\ cf;\ a = a+1;\ b = 0;\ af;\ cf\} \\
 & \qquad \textbf{else}\ \ \{af;\ cf;\ b = b+1;\ af;\ cf\};\ af \\
\equiv & \quad \{\text{Equation (6)}\} \\
 & cf;\ \ \mathcal{O}(\textbf{if}\ ((nc \,\%\, 10) == 9) \\
 & \qquad \textbf{then}\ \{cf;\ \ a = a+1;\ \ b = 0;\ \ af\} \\
 & \qquad \textbf{else}\ \ \{cf;\ b = b+1;\ af\});\ \ af \\
\equiv & \quad \{\text{definitions and Equation (4)}\} \\
 & \textbf{if}\ ((nc \,\%\, 10) == 9) \\
 & \textbf{then}\ \{a = nc \,/\, 10;\ \ b = nc \,\%\, 10;\ \ a = a+1; \\
 & \qquad\quad b = 0;\ \ nc = 10*a+b\} \\
 & \textbf{else}\ \ \{a = nc \,/\, 10;\ \ b = nc \,\%\, 10;\ \ b = b+1; \\
 & \qquad\quad nc = 10*a+b\} \\
\equiv & \quad \{\text{simultaneous equations in branches}\} \\
 & \textbf{if}\ ((nc_0 \,\%\, 10) == 9) \\
 & \textbf{then}\ \{a_1 = nc_0 \,/\, 10;\ b_1 = nc_0 \,\%\, 10; \\
 & \qquad\quad a_2 = a_1 + 1;\ \ b_2 = 0;\ \ nc_1 = 10*a_2+b_2\} \\
 & \textbf{else}\ \ \{a_3 = nc_0 \,/\, 10;\ b_3 = nc_0 \,\%\, 10; \\
 & \qquad\quad b_4 = b_3 + 1;\ \ nc_2 = 10*a_3+b_4\} \\
\equiv & \quad \{\text{substitutions}\} \\
 & \textbf{if}\ ((nc_0 \,\%\, 10) == 9) \\
 & \textbf{then}\ \{nc_1 = 10*(nc_0 \,/\, 10)+10\} \\
 & \textbf{else}\ \ \{nc_2 = 10*(nc_0 \,/\, 10)+(nc_0 \,\%\, 10)+1\} \\
\equiv & \quad \{\text{modular arithmetic}\} \\
 & \textbf{if}\ \ ((nc_0 \,\%\, 10) == 9)\ \ \textbf{then}\ \{nc_1 = nc_0+1\} \\
 & \qquad\qquad\qquad\qquad\qquad\ \ \textbf{else}\ \ \{nc_2 = nc_0+1\} \\
\equiv & \quad \{\text{convert back to assignments}\} \\
 & \textbf{if}\ \ ((nc \,\%\, 10) == 9)\ \ \textbf{then}\ \{nc = nc+1\} \\
 & \qquad\qquad\qquad\qquad\quad\ \ \textbf{else}\ \ \{nc = nc+1\} \\
\equiv & \quad \{\text{identical branches}\} \\
 & nc = nc+1
\end{array}
$$

Figure 7: Proof of correctness for the transformation using a variable split

4.6 Arrays

So far we have only considered simple variables but what would happen to the slicing (and the obfuscations) if we use arrays? Suppose that we had an expression of the form:

$$x = f(a[0])$$

for some variable x and array a. If we perform a backwards slice for x from this point then the slice for x will contain assignments for other array indices and not just for $a[0]$.

For Word Count, we can perform this transformation:

$$nl \rightsquigarrow a[0] \qquad nw \rightsquigarrow a[1] \qquad nc \rightsquigarrow a[2]$$

This transformation actually is just a variable renaming. If we ensure that the array a is indexed by using only 0, 1 and 2 then the transformation is correct. In Figure 9 we can see the result of taking backwards slice for $a[0]$ (*i.e.* nl). The results for *array* can be seen in Table 1 where the results for nl, nw and nc represent $a[0]$, $a[1]$ and $a[2]$ respectively. The size of the new method is actually smaller than the *original* method — this is due to the initialisation of the variables. This simple transformation results in significantly decreasing the residues for all three of the output variables.

Once we use arrays we can employ many different array transformations. However, array restructuring transformations such as splitting and folding [7], which are often used as array obfuscations, are not very suitable for creating dependencies on other variables. Instead we should concentrate on transforming the array index to create dependencies.

```
split(){
  int c, nl = 0, nw = 0, a = 0, b = 0, in;
  in = F;
  while ((c = getchar()) != EOF) {
    if (b == 9) {a ++;  b = 0; } else {b ++; }
    if (c ==' ' || c ==' \n' || c ==' \t') in = F;
    else if (in == F) {
      if (b < 10) {in = T;  nw ++; }
      else {nw = nw + nl;  b = b + nw; }   }
    if (c ==' \n') {nl ++;  if (in == T) {nl = a + nl; }}
  out(nl, nw, 10 * a + b);    }
```

Figure 8: Variable Split (with the backwards slice for $10 * a + b$).

```
array() {
  int c, in;
  int a[3] = {0, 0, 0};
  in = F;
  while ((c = getchar()) != EOF) {
    a[2] ++;
    if (c ==' ' || c ==' \n' || c ==' \t') in = F;
    else if (in == F) {in = T;  a[1] ++; }
    if (c ==' \n') a[0] ++;              }
  out(a[0], a[1], a[2]);    }
```

Figure 9: Transformation of the output variables to arrays (with the backwards slice for $a[0]$).

5. APPLYING THE TRANSFORMATIONS

In a previous section we outlined a number of transformations that are suitable for producing slicing obfuscations. In this section, we discuss some of the choices that we can make when applying our obfuscations.

5.1 Placing the transforms

When determining where to place our obfuscating transforms we used the backwards slices of the program to help us to decide — in particular, we generally concentrated on orphaned points in the residues of the output variables. For transformations such as encodings and splits, we need to apply them to the whole of the method (or, at least, to the scope of the variable). With other transformations, such as placing bogus predicates, we have a choice in where to place our obfuscations.

If, when slicing for a particular variable, y say, we have an assignment $x = G$ in the residue of y then we can add a dependency for y by using the transformation in Equation (8). If we have a number of assignments for x then, as we consider backwards slices, we pick the last assignment for x.

Suppose we have the following code fragment:

$$x = E;\ \ S;\ \ x = F;$$

where S is a block of statements in which x is used but not defined and F is an expression which does not depend (directly or indirectly) on x. This means that the assignment

$x = F$ kills the previous definition of x and the backwards slice for x may not contain the earlier definition for x. We can perform the following example transformation:

$$x = F; \quad \rightsquigarrow \quad \textsf{\textbf{if}}\ (p^T)\ x = F;\ \textsf{\textbf{else}}\ x\,{+}{+};$$

Now the backwards slice for x should include the previous assignment to x.

Many of the transformations that we have given relied on the use of predicates which were a simple kind of *opaque predicate* [8]. A predicate p is defined to be opaque at a certain program point if its value is known to the obfuscator but it is difficult for an adversary to deduce statically. The predicates in our examples used invariants that we, as the creator of a method, knew to be true. We should aim, where possible, to use predicates that are hard for an attacker to determine, or, at the very least, require some calculations to compute this value. When deciding where to place a bogus predicate, we should, therefore, determine what invariants we could use and pick a place that has an invariant which seems hard to determine.

Our loop transformation in Section 4.4 was effective in reducing the sizes of the residues with only a small increase in the method size — but obviously this transformation is only applicable if our method contains a loop. If we can "fake" a **while** loop then we can apply the loop transformations. Suppose that we have a block of code B and the state before B is σ. Then we need to find a predicate p such that $p(\sigma)$ is true but $p(B(\sigma))$ is false. Armed with such a predicate we can perform the following transformation:

$$B \quad \rightsquigarrow \quad \textsf{\textbf{while}}\ (p)\ \{B\}$$

Thus we can now apply our loop transformations.

5.2 Program Blocks

As we saw in Section 3.3 if we have a piece of code

$$P \equiv B_1;\ B_2$$

(where B_1 and B_2 are blocks of code) and an obfuscation $\mathcal{O}$ with conversion function cf and abstraction function af then we have two ways to obfuscate P. Either we can obfuscate B_1 and B_2 separately and compose the results, *i.e.*

$$\mathcal{O}(P) \equiv \{af;\ B_1;\ cf\};\ \{af;\ B_2;\ cf\}$$

or we can obfuscate both blocks together *i.e*

$$\mathcal{O}(P) \equiv af;\ B_1;\ B_2;\ cf$$

The two obfuscations that we obtain are equivalent but they may look different. In particular the second derivation may reduce the number of assignments.

For example, suppose that:

$$P \equiv \{x = x + 1;\ B;\ x = 3 * x\}$$

where B is a block of code in which x does not occur and $cf \equiv x = x + 2$ and $af \equiv x = x - 2$. If we obfuscate the two assignments separately then we have that

$$\mathcal{O}(P) \equiv \{x = x + 1;\ B;\ x = 3 * x - 4\}$$

However computing $af; P; cf$ will give us the following set of simultaneous equations:

$$x_1 = x_0 - 2;\ \ x_2 = x_1 + 1;\ \ B;\ \ x_3 = 3 * x_2;\ \ x_4 = x_3 + 2$$

Reducing this set of equations (and remembering that x does not occur in B) gives us:

$$B;\ \ x_4 = 3 * (x_0 - 1) + 2$$

Thus $\mathcal{O}(P) \equiv \{B;\ x = 3 * x - 1\}$.

The two derivations produce equivalent programs but the second program only has one assignment to x. From an obfuscation point of view, the first program would appear to better as it has more assignments to x and so it is (slightly) harder to work out the value of x at the end of $\mathcal{O}(P)$.

Instead of completely reducing a set of simultaneous equations we can partially reduce them. For instance in the example above, we can substitute x_1 and x_3 and so we would obtain:

$$\mathcal{O}(P) \equiv \{x = x - 1;\ B;\ x = 3 * x + 2\}$$

We can partially reduce a set of simultaneous equations in different ways. Thus we have some flexibility when deriving obfuscation for a sequence of statements using a particular conversion function.

5.3 Localising the transformations

The variable obfuscations proposed in [7] and [12] are applied to an entire program or at the very least the entire scope of a variable. If we apply a data obfuscation to the whole program then we need to convert any input to an obfuscated variable using a conversion function. Any outputs of obfuscated variables need to have the appropriate abstraction function applied to them.

When using conversion functions we can localise an obfuscation to a particular code block. Suppose we have an obfuscation (with functions cf and af) of a variable x and a piece of code with three blocks $A; B; C$ which all define or use x. Then we can obfuscate B separately to obtain $\mathcal{O}(B)$ and so our code becomes:

$$A;\ cf;\ \mathcal{O}(B);\ af;\ C$$

Note that since $cf; \mathcal{O}(B); af \equiv B$ then we must ensure that we do not "reduce" this code sequence otherwise we will "de-obfuscate" $\mathcal{O}(B)$.

If we had three different data obfuscations (say $\mathcal{O}_A$, $\mathcal{O}_B$ and $\mathcal{O}_C$ with appropriate conversion functions) then we can obfuscate the blocks A, B and C separately and sequentially compose the results:

$$cf_A;\ \mathcal{O}_A(A);\ af_A;\ cf_B;\ \mathcal{O}_B(B);\ af_B;\ cf_C;\ \mathcal{O}_C(C);\ af_C$$

This means that we can have regions in the program in which we can apply different obfuscations to the same variable and so we can create a "scope" for an obfuscation. To help disguise the conversions we should try to combine the expressions for $af_A;\ cf_B$ and $af_B;\ cf_C$.

5.4 Combining transformations

Since we are considering our obfuscations as functions we may naturally want to compose obfuscations. For some variable x, suppose that we have two obfuscations $\mathcal{O}_1$ and $\mathcal{O}_2$. For these obfuscations, the conversion functions are $cf_1 \equiv x = f_1(x)$ and $cf_2 \equiv x = f_2(x)$ (with corresponding abstraction functions $af_1 \equiv x = g_1(x)$ and $af_2 \equiv x = g_2(x)$). To obfuscate a statement S by applying $\mathcal{O}_1$ followed by $\mathcal{O}_2$ we have:

$$\mathcal{O}_2(\mathcal{O}_1(S)) \equiv af_2;\ af_1;\ S;\ cf_1;\ cf_2$$

This is equivalent to having a single obfuscation $\mathcal{O}_{1;2}$ with conversion function $cf_{1;2} \equiv x = (f_2 \cdot f_1)(x)$ and abstraction function $af_{1;2} \equiv x = (g_1 \cdot g_2)(x)$. We define $\mathcal{O}_{1;2} \equiv \mathcal{O}_2 \cdot \mathcal{O}_1$.

For example, we can apply a variable transformation to array elements. So using the function $\lambda x.(2 * x + 1)$ we could have the following array conversion between the arrays A and B:

$$cf \equiv B[i] = 2 * A[i] + 1$$

Since i acts as a dummy variable we can write transformations which depend on i:

$$cf \equiv B[i] = A[i] + i$$

We can combine a variable transformation with array obfuscations given in [13]. For instance if we had the functions $f :: \mathbb{Z} \to \mathbb{Z}$ and $p :: [0..n) \to [0..n)$ (with appropriate inverses) then here is a possible conversion function

$$cf \equiv A[i] = f(A[p(i)])$$

in which f acts as a variable transformation and p is an array index permutation.

Another way to combine obfuscations is to overlap their scope. For instance suppose we have the following blocks of code: A; B; C and we have two data obfuscations $\mathcal{O}_1$ applied to $A; B$ and $\mathcal{O}_2$ applied to $B; C$. Then we have:

$$\mathcal{O}_1(A);\ \mathcal{O}_{1;2}(B);\ \mathcal{O}_2(C)$$

For our examples we considered up to three output variables and so sometimes it was necessary to add more than obfuscation. For example, for *encode*3 (from Section 4.3) we use three different encodings (one for each output variable). This means that, in effect, we have created a composite obfuscation. Further work is needed to 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?

6. CONCLUSIONS

The problem of providing a provable security model for obfuscation has plagued the cryptography research community for a long time. The security requirements for obfuscation have been poorly understood and this has resulted in the lack of credible theoretical results in obfuscation since Barak's landmark paper [3] in 2001. Keeping this in mind, we take the complementary approach of addressing the problem by coming up with weaker notions of obfuscation, which, in cryptography community, is often termed as "fuzzy security". Whereas the theoretical approach to defining provable security of obfuscations has been to consider security against "all" polynomial-time adversaries, we consider it prudent to consider weaker classes of adversaries, and in this contribution we take the simplest case where the adversary is just a single (a-priori known) static slicing algorithm.

We have conducted experiments in which we considered adding existing obfuscations that were targeted to be more resistant to slicing. We proposed a new measurement for slicing obfuscations called a *residue* which consists of points which are *orphaned* (*i.e.* left behind) by slicing. Our goal has been to reduce the number of orphaned points. In Table 1 we can see that, according to our residue metrics, we have successfully created transformations to reduce the size of residues. Further empirical evaluation results appear in [13]. Since our metrics are related to the slicing based metrics from [21], our obfuscations make slicing less useful. The results for the single variable encoding *encode*1 highlight the importance of ensuring that when obfuscating we consider the slices for all variables. We have only conducted a relatively small experiment but it has shown us how to use obfuscations to decrease the effectiveness of slicing. In [13] we consider similar obfuscating transforms for different example C programs and we find that these obfuscations make slicing less effective. In Table 2 we present a summary of these results. The result of Table 2 is represented graphically using a bar chart in Figure 10. This figure groups the residue metrics for each of our example programs (12 programs grouped in 4 categories). We can see from Figure 10, a better slicing obfuscation reduces the percentage of the residues that are left behind after slicing.

In our experiments we only used simple transformations to illustrate the techniques used to create slicing obfuscations. Even with these simple transformation we have 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 pointers and dealing with inter-procedural constructs. For future work, it would be interesting to see if composing different obfuscations (developed for withstanding different program analysis attacks) provides more resilience in protecting a program against combined attack models (where an adversary tries to mount attacks using more than one tool).

We have modelled our statements as state functions and data obfuscations as refinements and as a consequence we have been able to easily prove the correctness of our data obfuscations. For our obfuscations we have given rewrite rules detailing how code fragments are transformed. These rules ensure that we apply our obfuscating transforms precisely and so we can be sure that our obfuscations are correct. Correctness of individual transforms ensures that the obfuscated program is semantically equivalent to its unobfuscated counterpart.

Our obfuscations were created manually and in Section 5 we indicated how we used slicing to determine where we should place our obfuscating transforms. So an area for future work is to consider automating the process of applying obfuscating transforms but this we consider to be a hard problem. 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. Yet another interesting future work lies in the area of correlating metric values to the "degree" of obfuscations so that we can tell the extent to which a program has been obfuscated by just reading certain metric value and vice-versa.

7. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for providing us with valuable comments and ideas for integrating in the paper. We thank them for suggesting interesting open problems and future research directions for slicing obfuscations.

We thank the New Zealand government for funding this research through the New Economy Research Foundation (NERF) grant titled "Securing Software for New Zealand".

Method M	$\|M\|$	$\|V_O\|$	For each v_i the residue size $\|RES_i\|$						$\|RES_{un}\|$	$MinD(M)$	$D(M)$	$MaxD(M)$	$C(M)$
ps	21	2	$prod$	9	sum	9			14	42.9%	42.9%	42.9%	66.7%
$psObf1$	22	2	$prod$	6	sum	9			11	27.3%	34.1%	40.9%	50.0%
$psObf2$	26	2	$prod$	7	sum	7			9	26.9%	26.9%	26.9%	34.6%
$search$	107	2	n	98	$secs$	96			105	89.7%	90.7%	91.6%	98.1%
$searchObf1$	120	2	n	75	$secs$	109			110	62.5%	76.7%	90.8%	91.7%
$searchObf2$	127	2	n	78	$secs$	79			81	61.4%	61.8%	62.2%	63.8%
rov	124	2	$fuel$	101	$dist$	78			105	62.9%	72.2%	81.5%	84.7%
$rovObf1$	129	2	$fuel$	69	$dist$	83			84	53.5%	58.9%	64.3%	65.1%
$rovObf2$	132	2	$fuel$	70	$dist$	72			73	53.0%	53.8%	54.5%	55.3%
$scatter$	143	3	si	27	ru	32	i	134	135	18.9%	45.0%	93.7%	94.4%
$scatterObf1$	148	3	si	16	ru	16	i	16	17	10.8%	10.8%	10.8%	11.5%
$scatterObf2$	150	3	si	11	ru	11	i	11	12	7.3%	7.3%	7.3%	8.0%

Table 2: Table showing the residue metric values for example programs. There is a separate row in the table for recording the residue metric values of the example programs and their obfuscated counterparts. The row labelled ps, for example, records the slicing metric values for the unobfuscated instance of Product Sum example; whereas, $psObf1$ indicates the metric values when slicing obfuscations have been applied. The columns from $|M|$ to $|RES_{un}|$ reflect measures with respect to the number of SDG nodes. The latter columns indicate the residue metric values.

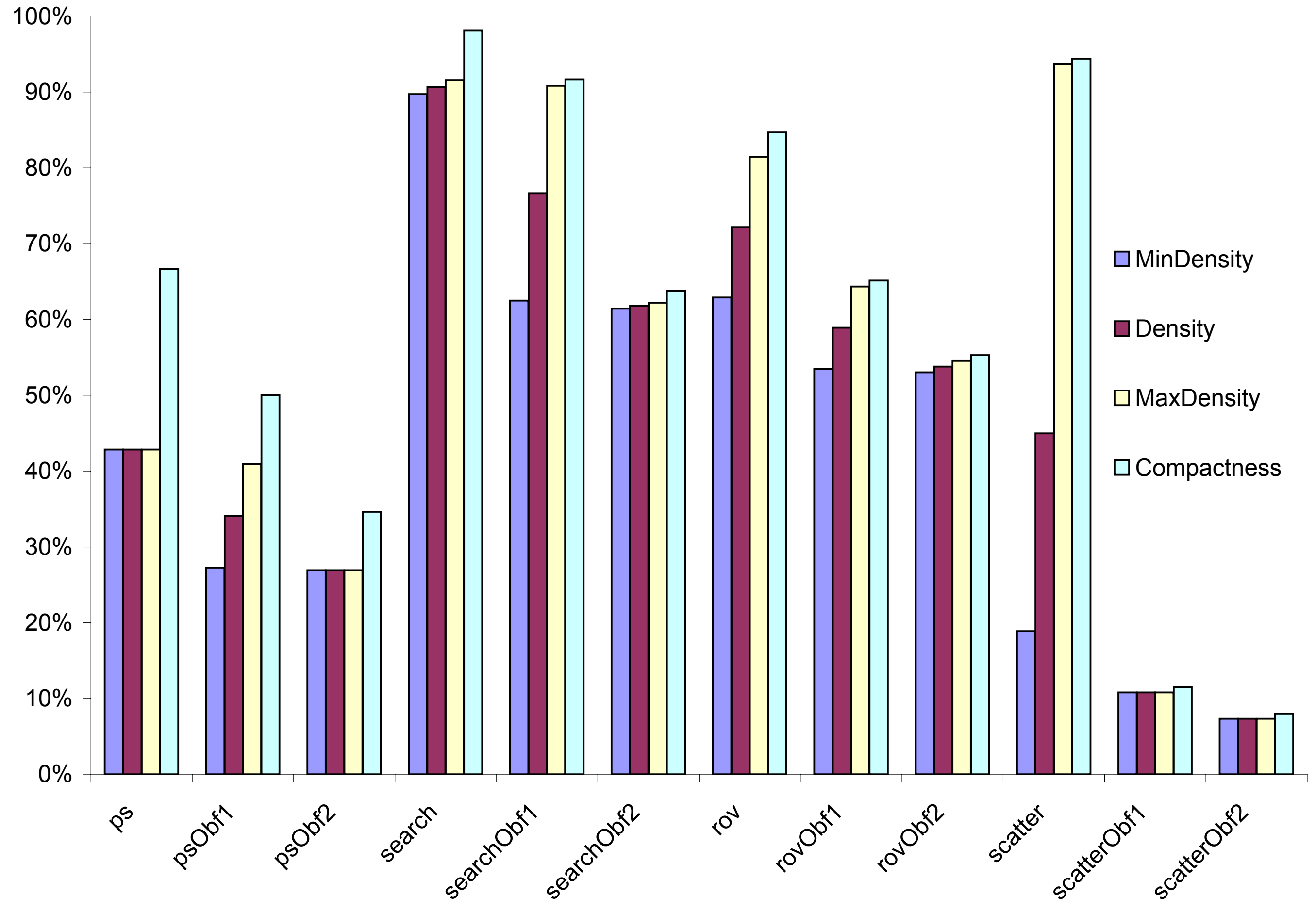

Figure 10: Bar Chart showing residue metric values for all our example programs

8. REFERENCES

[1] Business Software Alliance. Second annual BSA and IDC software piracy study, May 2005. Available from `www.bsa.org/globalstudy/upload/2005-Global-Study-English.pdf`.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] Christian Collberg, Clark D. 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.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[10] 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.

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

[12] Stephen Drape, Oege de Moor, and Ganesh Sittampalam. Transforming the .NET Intermediate Language using Path Logic Programming. In *Principles and Practice of Declarative Programming*, pages 133–144. ACM Press, 2002.

[13] Stephen Drape and Anirban Majumdar. Design and Evaluation of Slicing Obfuscations. Technical Report 311, University of Auckland, New Zealand, June 2007.

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

[15] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[16] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[17] 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.

[18] Anirban Majumdar, Antoine Monsifrot, and Clark D. 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.