CDMTCS
Research
Report
Series

Design and Evaluation of Slicing Obfuscations

Stephen Drape
Anirban Majumdar
Department of Computer Science
The University of Auckland
Auckland, New Zealand

CDMTCS-311
June 2007

Design and Evaluation of Slicing Obfuscations

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

Abstract

The goal of obfuscation is 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 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.

Keywords: Obfuscation, Static Slicing, Program Transformation, Software Security

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 [31]. The motivation for protecting software through obfuscation arises from the problem of software piracy, which can be summarised as a reverse engineering process [10] undertaken by a software pirate when stealing intellectual artefacts (such as a patented algorithm) from 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 internet telephony client [9], the SDC Java DRM [37], and most license-control systems rely, at least in part, on obfuscation for their security.

Collberg *et al.* [11, 12] 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 learnt (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 [15] 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 comprehension. Program comprehension forms the basis of reverse engineering since its primary goal 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 so that the attacker fails to recognise the portions of interest by analysing the code. Drape and Majumdar in [17] 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 their along many directions — first, we show a way to prove correctness of our proposed slicing obfuscations using program refinement techniques [14]. Secondly, we empirically evaluate our obfuscating transforms using a popular static program slicer, CodeSurfer [2], and show that the results, in fact, corroborate with the claims in [17]. 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.

1.1 Structure of the paper

We provide a brief overview of slicing and its role in program comprehension in Section 2. In Section 3 we give details of the framework that we will use to specify and prove the correctness of our obfuscations. To build our framework we adapted work from [15] which used functional programs by modelling statements as functions on the state. We then show how we can specify data obfuscations for a simple imperative language and how to construct proofs of correctness. In Section 4 we use the specification framework to construct and prove the correctness of various data obfuscations from [11] including array transformations. In Section 5 we give details of how we set up our experiments. In particular, we will discuss the use of slicing for designing obfuscating transforms and what quantities we will measure in our experiments. In Sections 6 and 7 we show the results of our experiments. Firstly, in Section 6, we give a detailed account of some of the different slicing obfuscations that can be applied to a particular example. Then, in Section 7, we give a brief overview of some more experiments that we performed on different programs. We give some of the choices that can be made when applying obfuscating transforms in Section 8 and we give a summary of our work in Section 9.

2 Background

Before describing our proof framework for slicing obfuscations, we provide a general overview of program slicing and its use in code comprehension. We conclude this section by highlighting the salient features of previous work done on defeating slicing attacks.

2.1 Program Slicing

Program slicing as a software engineering technique was first introduced by Weiser [41] in 1980. Since then several research papers have been published on program slicing. Researchers have tried to improve on Weiser's precision and extensibility. Moreover, work has also been done to add functionality to basic slicing algorithms. We discuss in this subsection, a few of the features available in different slicing tools and algorithms. Details can be found in extensive surveys and evaluation reports published in [5, 8, 22, 38, 42].

A *program slice* consists of the parts of a program that potentially affect the values computed at some point of interest (called a *slicing criterion*) [38]. According to Weiser's original definition, a slicing criterion C consists of a pair (line-number, variable) and parts of a program which have a direct or indirect (computed through data and control-flow analyses) influence on the values computed C are called the *program slice* with respect to C. The notion of slicing comes from Weiser's observation that abstracting away parts of program corresponds to the mental abstractions that people make when they are debugging a program. In his original proposal, Weiser defined a program slice S as an executable subset of a program obtained from deleting statements such that S replicates part of the behaviour of the program. In subsequent research, the executability feature was waived and slices were just defined to be subset of statements and conditional predicates of the program which directly or indirectly affect the value computed at the criterion C. Weiser's slices were also *backwards*, which means the statements in S are calculated by a backward traversal of the program starting from C. This gives the statements which affect the value of variables in C right before the statement defining C is executed. The notion of *forward* slicing was first defined in [4]. A forward slice gives the statements and conditional predicates that depend on the values of variables defined at the slicing criterion C. Tip in [38] observed that both forward and backward slices are computed in a similar way. Unless otherwise stated, slicing in our context would mean backward slicing.

Slicing techniques can be categorised based on their capability to slice statically or dynamically. Weiser's slicing technique mentioned in the previous paragraph is a static one since the underlying slicing algorithm finds a program statement subset by solving (approximately) a set of static program analyses problems. The program that is being sliced is not executed. The concept of dynamic slicing was proposed by Korel and Laski in [28] and is considered to be an instance of flowback analysis [38]. In flowback analysis, the user interactively traverses the graph that represents data as well as control-flow information and traces the information path that leads to a particular value of a variable in the program. Unlike flowback analysis, however, dynamic program slicing is non-interactive and only the dependencies that occur in a particular run (execution) of a program are taken into account. A dynamic slicing criterion is a triple (input, occurrence of a statement, variable). Dynamic slicing assumes a fixed input for the program, whereas static slicing does not make assumptions regarding the input. A dynamic slicing algorithm, therefore, is thought to give a more precise and smaller slice compared to its static counterpart. Again in our experiments, unless otherwise stated, slicing would mean static slicing. Variations of static and dynamic slicing (such as amorphous and quasi-static slicing) can be found in the survey articles.

Static slicing techniques can be categorised based on their intermediate representation of dependency information. Weiser's algorithm is most primitive and it uses dataflow equations to represent data and control-flow dependencies. The slice is obtained by an iterative solution of the dataflow equations. The slicing criterion in Weiser's case maps to a particular node in the Control-Flow Graph (CFG) of the program. A slice is "statement-minimal" if no other slice exists with fewer statements for the same criterion and it has been shown to be undecidable [41]. Ottenstein and Ottenstein, in [34], first proposed slicing as a graph reachability problem on Program Dependency Graphs (PDG) [18]. PDGs were extended to System Dependence Graphs (SDG) to accommodate inter-procedural slicing by Horwitz *et al.* in [24]. The statements and expressions of a program constitute the vertices of an SDG and the edges correspond to data and control dependencies between statements. The partial ordering of the vertices induced by the dependence edges must be preserved in order to guarantee semantic correctness of the program. Several extensions and approximation algorithms were proposed over these approaches to facilitate features such as alias calculation and unstructured program flow. Details can be found in the survey literature. Slicing using SDGs has been evaluated to be the most precise and complete slicing method currently available [8, 22, 38].

2.2 Slicing in program comprehension

Frank Tip [38] and Binkley *et al.* [8] outline a number of interesting uses of program slicing. Some of the notable uses of static program slicing are outlined here. Lyle in his PhD thesis [29] outline the use of program slicing to localise a bug. He called the concept *program dicing*, a method for combining the information obtained from different slices to assist in locating the bug. An interesting consequence of this idea was another method to eliminate *dead code* within the program [4]. Horwitz introduced the concept of *program differencing* using static slicing [23] based on her algorithm of static program slicing presented in [24]. Program differencing is a technique for finding out parts of the new program which has changed from its previous/older counterpart. In [20], the notion of *decomposition slice* was introduced, where a decomposition slice with respect to a variable v is defined as the union of slices with respect to v at the statements that output v and the last statement of the program. The intended use of decomposition slice is in the program of software maintenance where it is suggested that the complement of decomposition constitutes the set of independent statements which may be modified without having any side effects to the variable v. In their seminal paper [21], Gupta, Harrold and Soffa proposed the idea of *regression testing* where only the parts affected by the modification of a previously tested program are re-tested using program slicing while maintaining the coverage of the original test suite. Another important use of slicing is *clone detection* where non-contiguous, reordered, and intertwined clones are detected using isomorphic PDG subgraph matching [27].

Program slicing has been found to facilitate program comprehension and it is this specific use that we are particularly interested in. Binkley and Harman, in their survey [8], reported several research efforts undertaken to link slicing with program comprehension factors. Mark Weiser stated in [41] that programmers, while debugging, mentally traverse backwards from the location of bug (variable and location as slicing criterion) to find the program slice and thus related human comprehension of program to slicing. Lyle in his PhD thesis [29] similarly conjectured that programmers using dicing (intersection of two or more slices) have been observed to find faults faster than their counterparts using traditional methods. Francel and Rugaber in [19] compared the level of comprehension of programmers who used slicing to those who do not and hypothesised that the former had a better understanding of code for debugging than the latter. Ott and Thuss used program slicing based metrics to estimate cohesion among program modules [33]. This estimation, in turn, helped in maintaining program modularity. They created *slice profiles* to aid in visualisation of relationships among different slices generated by a program module and extended Weiser's slicing metrics to build a quantitative approach towards measuring module cohesion (compared to previous qualitative approaches).

Among the newer endeavours, Rilling and Klemola, in [36], proposed an approach to identify comprehension bottlenecks by combining slicing with cognitive complexity metrics. They observed that comprehension aids in digging program artefacts and their relationships using reverse engineering — something an adversary would attempt on an obfuscated program. Their approach to measure cognitive complexity of source code involved defining three types of *identifier densities* and using these densities as slicing criteria to aid in comprehension. David Brinkley and Mark Harman did extensive research on the subject of slicing and its use in comprehension. In [7], they performed a large scale empirical study on a million lines of C code to understand context sensitivity for calling methods. They collected more than two million slices to understand how the size of slices were affected by taking the calling contexts into account (and vice-versa). They observed that taking into account the calling context, backward slices included about 28% of the program and forward slices about 26% of the program. However, ignoring calling contexts led to an overall increase of 50% on the slice sizes. In another empirical study

[6], Binkley and Harman investigated the interdependencies among program components affected by predicates in the program. Using slicing they came to the conclusion that predicates in a program rarely depend on all the formal parameters and global variables. Their results also show the differences between formal parameter dependence and global variable dependence. Lastly, Meyers and Binkley, in [32], devised slicing based program cohesion metrics to aid in software intervention. Intervention in their context refers to the task of identifying modules within a program that require reconstruction. They followed up on the work of Ott and Thuss [33] to devise slice-based cohesion metrics that quantify the overall quality of source code and could be used to measure software intervention efforts. For these cohesion metrics, they suggested base-line values which could, in turn, be used to identity degraded program modules.

2.3 Previous work on slicing obfuscations

Ivanov and Zakharov first proposed the idea of designing obfuscations with the intent of obstructing static analysis attacks on code [25]. They reported that for obfuscations manufactured from intractable complexity problem instances, the principal drawback is that none has a formal basis for substantiating claims about code comprehension resilience. In their framework of study, a program static analysis tool A computes a mapping of the form: $A : U \mapsto W$, where U is the set of unobfuscated programs under consideration and W is the set of possible outcomes of static analysis. The set W of all possible outcomes forms a lattice which has the minimal element $\perp$. This element can be interpreted as a lack of any useful/significant information about the program semantics as may be analyses using A. The defined an obfuscation $\mathcal{O}$ to be *impervious* to a program static analyser A if $A(\mathcal{O}) = \perp$ holds for any program P. With this goal in mind, a particular static analysis tool can be made *unsuitable* for deobfuscation if the program P is obfuscated using $\mathcal{O}$.

Using this definition of imperviousness, they set out to design slicing resistant obfuscations. They defined the efficiency of a slicing algorithm A_S by the ratio

$$\frac{\mid S^{A_S}_{\langle p,V\rangle} \mid}{\mid S^{min}_{\langle p,V\rangle} \mid}$$

where $(S^{A_S}_{\langle p,V\rangle})$ is the set of program points in the slice with respect to the criterion $\langle p, V\rangle$, and $(S^{min}_{\langle p,V\rangle})$ is the minimal set of program points that are necessary to be maintained in order to correctly compute the values of variables V at program point p. The authors noted that an obfuscation could be made impervious to a slicing algorithm if it is forced to give the worst possible slice, i.e., the set of *all* program points that precede p in the SDG.

There are a couple of problems with the imperviousness property. The first one concerns its absoluteness. While it may seem trivial to incorporate dependencies to link up every program point to the slicing criterion, our practical experience would suggest otherwise. An all-or-nothing model lends a cryptographic flavour to the obfuscation problem whereby an adversary managing to obtain a slice with one program point less than the worse possible slice would be considered a winner. Also, a bogus obfuscation will add spurious amount of dummy code which could then be made to depend on the slicing criterion — this is not really a correct measure of the success of obfuscation. How do we relax the imperviousness goal of obfuscation and how do we design obfuscations that link up the statements that are not included in the slice with respect to a particular slicing criterion? The remainder of this contribution addresses these issues. Secondly, Ivanov and Zakharov's technique is not evaluated empirically. It is easier to introduce false obfuscation dependencies in pseudo-code and claim it as being impervious to *any* slicing algorithm. However, in practice, engineering dependencies within program code and controlling side effects of analyses can be quite challenging and therefore it is important to validate how the theoretical claims corroborate when reduced to practice.

3 Proof Framework

Before we present our specific slicing obfuscations, we show how we can specify and prove the correctness of imperative data obfuscations.

In [15] a framework for proving the correctness of obfuscations for abstract data-types, defined in a functional language, using functional refinement [14] was given. Suppose that a data-type D is obfuscated using an obfuscation $\mathcal{O}$ to produce a data-type E. Under this framework, an abstraction function $af :: E \to D$ and a data-type invariant dti are needed such that, for $x :: D$ and $y :: E$:

$$x \rightsquigarrow y \iff (x = af(y)) \wedge dti(y) \tag{1}$$

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

For a function $f :: D \to D$, 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

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

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

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

and we can use this equation to prove the correctness of $\mathcal{O}(f)$.

In this contribution we will concentrate on imperative program constructs — can we still use this framework to prove the correctness of 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* $\to$ *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

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

For our simple language, we consider the following statement types: *skip*, *assignments* (*var* = *expr*), *conditionals* (**if** *pred* **then** *stats* **else** *stats*) and *loops* (**while** *pred* **do** *stats*).

The statement *skip* does not change the state and so if $S \equiv \mathit{skip}$ 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

$$A(\sigma_0) = \sigma_0 \oplus \{x \mapsto e[x_0 / x]\} \tag{3}$$

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

Now suppose we have conditional statement C which has the form

$$C \equiv \textbf{if } p \textbf{ then } T \textbf{ else } E$$

where p is a predicate with type $p :: \mathit{state} \rightarrow \mathbb{B}$ and T and E are blocks. Then for some initial state σ_0 we have that

$$C(\sigma_0) = \begin{cases} T(\sigma_0) & \text{if } p(\sigma_0) \\ E(\sigma_0) & \text{otherwise} \end{cases} \tag{4}$$

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

$$L(\sigma_0) = S^i(\sigma_0) \text{ where } i = \min\{n :: \mathbb{N} \mid p\left(S^n(\sigma_0)\right) = \mathit{False}\} \tag{5}$$

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 σ to produce a new state $\mathcal{O}(\sigma)$ and so we can consider the set of states $\mathcal{S}$ to be obfuscated to produce a new set of states $\mathcal{O}(\mathcal{S})$. To specify a data obfuscation $\mathcal{O}$ we will supply two functions, an abstraction function

$$af :: \mathit{state} \rightarrow \mathit{state}$$

such that

$$af(\gamma) = \delta \Leftrightarrow \gamma \in \mathcal{O}(\mathcal{S}) \wedge \delta \in \mathcal{S}$$

and a conversion function

$$cf :: \mathit{state} \rightarrow \mathit{state}$$

which is a pre sequential inverse for af, *i.e.*

$$cf; af \equiv \mathit{skip}. \tag{6}$$

As well as these functions, for refinement, we require an invariant I on the obfuscated state such that for states $\sigma :: \mathcal{S}$ and $\mathcal{O}(\sigma) :: \mathcal{O}(\mathcal{S})$

$$\sigma \rightsquigarrow \mathcal{O}(\sigma) \Leftrightarrow (\sigma = af(\mathcal{O}(\sigma))) \wedge I(\mathcal{O}(\sigma))$$

The expression "$\sigma \rightsquigarrow \mathcal{O}(\sigma)$" means that the state σ is obfuscated (refined) by $\mathcal{O}(\sigma)$. Using the conversion function we have that

$$cf(\sigma) = \mathcal{O}(\sigma) \Rightarrow \sigma \rightsquigarrow \mathcal{O}(\sigma)$$

Note that for most of our transformations unless otherwise stated $I \equiv \mathit{True}$. The type of the abstraction and conversion functions are the same as the type of a statement and so we can consider af and cf to be statements (usually assignments) as well as functions. Since our statements are functions on the state, we can produce correctness equations similar to those for functional programs.

Suppose that we have a block B and we want to obfuscate it using data refinement to obtain a block $\mathcal{O}(B)$. We say that $\mathcal{O}(B)$ is *correct* (with respect to B) if it satisfies

$$(\forall \sigma :: \mathcal{S}; \mathcal{O}(\sigma) :: \mathcal{O}(\mathcal{S})) \bullet \sigma \rightsquigarrow \mathcal{O}(\sigma) \Rightarrow B(\sigma) \rightsquigarrow \mathcal{O}(B)(\mathcal{O}(\sigma)) \tag{7}$$

We can draw the commuting diagram [14] in Figure 1 representing our obfuscation and we obtain the following equation:

$$af; B \equiv \mathcal{O}(B); af \tag{8}$$

this corresponds with Equation (2). 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:

$$B \equiv cf;\ \mathcal{O}(B);\ af \tag{9}$$

We can extend the refinement notation $\rightsquigarrow$ to statements and blocks. The expression

$$B \rightsquigarrow \mathcal{O}(B) \ \text{ for some } af \text{ (and } cf\text{)}$$

says that B is obfuscated by $\mathcal{O}(B)$ and that Equation (7) holds.

$$\begin{tikzcd}
\sigma \arrow[r, "B"] & \gamma \\
\mathcal{O}(\sigma) \arrow[u, "af"] \arrow[r, "\mathcal{O}(B)"'] & \mathcal{O}(\gamma) \arrow[u, "af"']
\end{tikzcd}$$

Figure 1: A commuting diagram for data obfuscation

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 $cf;\ af \equiv skip$. This means that af and cf are statements of the form

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

for some functions F and G. For refinement we need $cf;\ af \equiv skip$, *i.e.*

$$\{x = F(x);\ x = G(x)\} \equiv skip$$

and so we need

$$G \cdot F = id \tag{10}$$

3.3.1 Composition

Suppose that s_1 and s_2 are two statements and let us consider how to obfuscate the sequential composition of these two statements. By Equation (8),

$$af;\ s_1 \equiv \mathcal{O}(s_1);\ af \quad \text{and} \quad af;\ s_2 \equiv \mathcal{O}(s_2);\ af$$

We propose that

$$\mathcal{O}(s_1;\ s_2) \equiv \mathcal{O}(s_1);\ \mathcal{O}(s_2)$$

Proof. Consider

$$\begin{aligned}
& \mathcal{O}(s_1);\ \mathcal{O}(s_2);\ af \\
\equiv\ & \quad \{\text{Equation (8) for } s_2\} \\
& \mathcal{O}(s_1);\ af;\ s_2 \\
\equiv\ & \quad \{\text{Equation (8) for } s_1\} \\
& af;\ s_1;\ s_2 \\
\equiv\ & \quad \{\text{Equation (8) for } s_1;\ s_2\} \\
& \mathcal{O}(s_1;\ s_2);\ af
\end{aligned}$$

and thus

$$\mathcal{O}(s_1;\ s_2) \equiv \mathcal{O}(s_1);\ \mathcal{O}(s_2)$$

$\square$

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

3.3.2 Assignments

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:

$$\mathcal{O}(x = E) \equiv x = F(E') \text{ where } E' = E[G(x) \diagup x] \tag{11}$$

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

Consider the following set of assignments:

$$\begin{array}{ll} & x = G(x);\ x = E;\ x = F(x) \\ \equiv & x = E[G(x) \diagup x];\ x = F(x) \\ \equiv & x = F(E[G(x) \diagup x]) \end{array}$$

and so as an alternative to Equation (11) we can write

$$\mathcal{O}(x = E) \equiv\ af;\ x = E;\ cf \tag{12}$$

3.3.3 Conditionals

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

$$\mathcal{O}(P_2) \equiv \textbf{if } p[G(x) \diagup x] \textbf{ then } \{af; T; cf\} \textbf{ else } \{af; E; cf\} \tag{13}$$

with af as above. Let us now prove that this is correct

Proof. From Equation (9) consider

$$(cf;\ \mathcal{O}(P_2);\ af)(\sigma_0)$$

for some initial state σ_0. Suppose that σ_0 contains the mapping $x \mapsto x_0$. By Equation (3) and the definition of cf:

$$cf(\sigma_0) = \sigma_0 \oplus \{x \mapsto F(x_0)\}$$

and so the conditional test is equivalent to:

$$\begin{array}{ll} & p[G(x)\,/\,x](\sigma_0 \oplus \{x \mapsto F(x_0)\}) \\ \equiv & \quad \{\text{put the substitution into the mapping}\} \\ & p(\sigma_0 \oplus \{x \mapsto G(F(x_0))\}) \\ \equiv & \quad \{\text{map } p \text{ across } \oplus\} \\ & p(\sigma_0) \oplus p(\{x \mapsto G(F(x_0))\}) \\ \equiv & \quad \{\text{Equation (10)}\} \\ & p(\sigma_0) \oplus p(\{x \mapsto x_0\}) \\ \equiv & \quad \{x \mapsto x_0 \text{ is the original mapping from } \sigma_0\} \\ & p(\sigma_0) \end{array}$$

So the conditional test for $cf;\ \mathcal{O}(P_2);\ af$ is the same as the test for P_2. If the test is $True$ then we have a block of statements of the form $cf;\ af;\ T;\ cf;\ af$ and by Equation (6) this is equivalent to T. Similarly if the test is $False$ then we find that the block of statements is equivalent to F. So we have shown that $cf;\ \mathcal{O}(P_2);\ af \equiv P_2$. $\square$

3.3.4 Loops

Suppose that $P_3 \equiv \textbf{while}\ p(x)\ \textbf{do}\ S$ then, with af as above, we propose that

$$\mathcal{O}(P_3) \equiv \textbf{while}\ p[G(x)\,/\,x]\ \textbf{do}\ \{af; S; cf\} \tag{14}$$

and we can use Equation (9) to prove that this is correct.

Proof. Consider

$$(cf;\ \mathcal{O}(P_3);\ af)(\sigma_0)$$

for some initial state σ_0 which contains the mapping $x \mapsto x_0$. By Equation (5), the final state is given by

$$(cf;\ (af;\ S;\ cf)^i;\ af)(\sigma_0)$$

for some integer i. By Equation (6) this simplifies to $S^i(\sigma_0)$.

We now need to find the value of i which by Equation (5) (and remembering that cf is executed before the loop) is:

$$\min\{n :: \mathbb{N} \mid p[G(x)\,/\,x]\,((af;\ S;\ cf)^n(\sigma_0 \oplus \{x \mapsto F(x_0)\})) = False\}$$

Consider

$$
\begin{array}{ll}
& p[G(x)/x]\,((af;\, S;\, cf)^n(\sigma_0 \oplus \{x \mapsto F(x_0)\})) \\
\equiv & \quad \{\text{Equation (6)}\} \\
& p[G(x)/x]\,((af;\, S^n;\, cf)(\sigma_0 \oplus \{x \mapsto F(x_0)\})) \\
\equiv & \quad \{\text{apply } af \text{ where } af \equiv x = G(x)\} \\
& p[G(x)/x]\,((S^n;\, cf)(\sigma_0 \oplus \{x \mapsto G(F(x_0))\})) \\
\equiv & \quad \{\text{Equation (10)}\} \\
& p[G(x)/x]\,((S^n;\, cf)(\sigma_0 \oplus \{x \mapsto x_0\})) \\
\equiv & \quad \{x \mapsto x_0 \text{ is the original mapping from } \sigma_0\} \\
& p[G(x)/x]\,((S^n;\, cf)(\sigma_0)) \\
\equiv & \quad \{\text{state after } S^n \text{ and } cf \text{ with } x \mapsto x_n \in (S^n)(\sigma_0)\} \\
& p[G(x)/x)]\,((S^n)(\sigma_0) \oplus \{x \mapsto F(x_n)\} \\
\equiv & \quad \{\text{similar argument to the conditional proof above}\} \\
& p(S^n(\sigma_0))
\end{array}
$$

So $i = \min\{n :: \mathbb{N} \mid p(S^n(\sigma_0)) = False\}$. Thus

$$P_3 \equiv cf;\ \mathcal{O}(P_3);\ af$$

and so our definition of $\mathcal{O}(P_3)$ was correct $\square$

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 (9) 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 has a different name which is usually the name of the variable with a subscript (*e.g.* x_2) and we 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, [13] 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.

3.5 Steps in a proof

There are four main steps in constructing our proofs of correctness.

Simultaneous Equations The first step is to convert a sequential program into a set of simultaneous equations using the SSA form as a guide. This means that each new definition of a variable has a unique subscript and each use of a variable should refer to the previous instance of the variable.

Substitution Once we have converted our sequential code to a set of simultaneous equations then the next phase is to reduce the set of equations by performing substitutions. However, sometimes problems can arise.

Suppose that we have the following set of simultaneous equations:

$$y_1 = x_0 + 1; \quad x_1 = x_0 + 2; \quad y_2 = y_1 - 1$$

Substituting the value for y_1 gives

$$y_1 = x_0 + 1; \quad x_1 = x_0 + 2; \quad y_2 = x_0$$

We can see that the "last" definition for x is at x_1 but the expression for y_2 uses an earlier definition of x. Whenever this type of situation occurs then we cannot immediately convert such sets of equations back to sequential code. The last step discusses possible solutions for this problem.

Redundant Definitions The next step after substitution is to remove redundant definitions. A definition $x_i = e$ is *redundant* in a set of simultaneous equations if no equation uses x_i and there exists some definition $x_j = e'$ where $j > i$. This latter condition ensures that we do not remove the "last" definition of a variable (and since we convert using a form of SSA we know that the last definition of a variable will have the largest subscript).

Converting back Once the set of simultaneous equations has been reduced they need to be converted to sequential code. As mentioned earlier, sometimes we cannot immediately convert the set of equations back to sequential code. For example, suppose that after substitution and refinement we are left with the following pair of simultaneous equations:

$$x_1 = x_0 + 2; \quad y_2 = x_0$$

This cannot be converted to:

$$x = x + 2; \quad y = x$$

as the final value of y in this sequence is equivalent to x_1 not x_0 as required. One solution is to introduce a new variable which holds the value of x_0:

$$t_1 = x_0; \quad x_1 = x_0 + 2; \quad y_2 = t_1$$

So this can be converted to:

$$t = x; \quad x = x + 2; \quad y = t$$

As an alternative, we could convert the simultaneous equations to:

$$x = x + 2; \quad y = x - 2;$$

However this kind of rewriting is not always possible in general.

4 Variable Transformations

In this section we give some examples of data transformations that can be used to obfuscate variables. We will use our refinement framework to specify these obfuscations and give some proofs of correctness.

4.1 Encoding

In [11] an obfuscation for variables called *encoding* is given. A simple example of an encoding for some variable x is:

$$x \rightsquigarrow \alpha * x + \beta$$

where α and β are constants. In [16] variable encoding was generalised as follows. For a variable encoding, (which in [16] was called a variable transformation), we require two functions f and g such that $g \cdot f = id$. The function f is the function that performs the transformation and g is its left inverse, so for the example above

$$f = \lambda i . \alpha * i + \beta$$
$$g = \lambda i . (i - \beta)/\alpha$$

To perform the transformation using f and g, the right hand side of any assignment to x has the function f applied to it and any uses of x should use $g(x)$ instead. So the statements

$$x = e \qquad \text{and} \qquad y = u(x)$$

should be transformed to

$$x = f(e) \qquad \text{and} \qquad y = u(g(x))$$

respectively.

For the transformation $x \rightsquigarrow \alpha * x + \beta$ we can write the conversion and abstraction functions as follows:

$$\begin{array}{l} cf \equiv x = \alpha * x + \beta \\ af \equiv x = (x - \beta)/\alpha \end{array}$$

and, as before, we can define the conversion and abstraction functions for general variable encodings as follows:

$$\begin{array}{l} cf \equiv x = f(x) \\ af \equiv x = g(x) \end{array}$$

We need that $cf; af \equiv skip$ so

$$\begin{array}{rcl} cf; af & \equiv & x = f(x);\quad x = g(x) \\ & \equiv & x = g(f(x)) \end{array}$$

Since $skip \equiv x = x$ then $cf; af \equiv skip \Leftrightarrow g \cdot f = id$ which is the condition mentioned earlier.

The conversion and abstraction functions are of the form of the functions used in Section 3.3 and so we can use the equations given in that section for transforming statements.

4.1.1 An example

In [16], the following example was discussed:

$$P \equiv \{i = 1;\ s = 0;\ \textbf{while}\ (i < 15)\ \textbf{do}\ \{s = s + i;\ i = i + 1\}\}$$

This example was then converted using the mapping $i \rightsquigarrow 2 * i$ to give the following program:

$$\mathcal{O}(P) \equiv \{i = 2;\ s = 0;\ \textbf{while}\ (i < 30)\ \textbf{do}\ \{s = s + (i/2);\ i = i + 2\}\}$$

This transformation was given without a proof of correctness.

The refinement functions for this obfuscation are:

$$cf \equiv i = 2 * i \qquad af \equiv i = i/2$$

To prove that $\mathcal{O}(P)$ is correct we use Equation (8) to show that:

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

This proof is given in Figure 2.

```
  af; P
≡ {definitions}
  af; i = 1; s = 0;
  while (i < 15) do
    {s = s + i; i = i + 1}
≡ {cf; af ≡ skip}
  af; i = 1; s = 0; cf; af;
  while (i < 15) do
    {s = s + i; i = i + 1}
≡ {Equation (11)}
  i = 2; s = 0; af;
  while (i < 15) do
    {s = s + i; i = i + 1}
≡ {Equations (14) and (8)}
  i = 2; s = 0; while ((i/2) < 15) do
    {af; s = s + i; i = i + 1; cf}; af
≡ {Equation (11)}
  i = 2; s = 0; while ((i/2) < 15) do
    {s = s + (i/2); i = 2 * ((i/2) + 1)};
  af
≡ {exact arithmetic}
  i = 2; s = 0; while (i < 30) do
    {s = s + i/2; i = i + 2}; af
≡ {definitions}
  O(P); af
```

Figure 2: Proof of correctness for a **while** loop

For an obfuscation which transforms a variable x to $\alpha * x + \beta * y$, we have three rewrite rules:

- (use of x) $U(x) \rightsquigarrow U(\frac{x-\beta*y}{\alpha})$
- (def of x) $x = E \rightsquigarrow x = \alpha * E' + \beta * y$ where $E' = E[\frac{x-\beta*y}{\alpha} / x]$
- (def of y) $y = f(x) \rightsquigarrow \begin{Bmatrix} t = x - \beta * y; \\ y = f(t/\alpha); \\ x = t + \beta * y; \end{Bmatrix}$

Figure 3: Rewrite rules for a variable encoding

4.2 A more complicated transformation

Suppose that we want to transform a variable x as follows

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

where α and β are constants and y is a program variable. The abstraction function for this obfuscation is

$$af \equiv x = \frac{x - \beta * y}{\alpha}$$

Note that, as with the earlier variable transformation, this obfuscation is not suitable if multiplication may overflow and division is not exact.

For this transformation we have three rewrite rules which are summarised in Figure 3. First, we transform a use of x, say $U(x)$, to $U(\frac{x-\beta*y}{\alpha})$. Next, using Equation (11) an assignment to x is transformed as follows:

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

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

Figure 4: Proof of correctness for a variable encoding

For example

$$
\begin{array}{rcl}
x\,{+}{+} & \leadsto & x = \alpha * ((x - \beta * y)/\alpha) + 1) + \beta * y \\
 & \equiv & x = x - \beta * y + \alpha + \beta * y \\
 & \equiv & x = x + \alpha
\end{array}
$$

Finally, 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). 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 4, Equation (9) is used to show that:

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

We do not always need to use a new variable when y is defined. For example, for our variable encoding, the block

$$x = x + 1; \quad y = y + 1$$

can be transformed to

$$y = y + 1; \quad x = x + \alpha + \beta$$

4.3 Variable Splitting

Another variable transformation mentioned in [11] is the concept of variable splitting. This is where a variable v (say) is represented by two or more new variables so that the information contained in v is "split" across these new variables.

Suppose that we split an integer variable v into two variables a and b. Using [15], we can write the relationship between v and the split components as

$$v \rightsquigarrow \langle a, b \rangle$$

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

$$a = f_1(v) \qquad b = f_2(v) \qquad v = g(a, b)$$

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

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

$$v = E \rightsquigarrow \{a = f_1(E');\ b = f_2(E')\} \quad \text{where } E' = E[g(a,b) / v] \tag{15}$$

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, let us split v into two other integers — one containing the least significant digit and the other containing the rest of the digits. So we can define $v \rightsquigarrow \langle v/10,\ v \,\%\, 10 \rangle$ and let $a = v/10$ and $b = v \,\%\, 10$ — 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 $g = \lambda i, j.\ 10 * i + j$. We can easily show that $af; cf \equiv skip$.

So, by Equation (15) the assignment $v{+}{+}$ would be transformed to:

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

Note that $((10 * a) + b + 1) \,\%\, 10 = (b + 1) \,\%\, 10$.

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 5 we show this is correct by proving $S \equiv cf; \mathcal{O}(S); af$. This transformation is more efficient as it does not need to use % or /.

4.4 Array Transformations

Various array transformations are mentioned in [11]:

- **Splitting** An array is transformed into two or more arrays (this is covered in more detail in Section 4.4.2).
- **Merging** Two or more arrays are combined into one array.
- **Folding** A 1-D array is transformed into an n-D array.
- **Flattening** An n-D array is changed into a 1-D array.

$$
\begin{aligned}
& cf; \mathcal{O}(S); af \\
\equiv\ & \{af; cf \equiv skip\} \\
& cf;\ \textsf{if}\ (b == 9)\ \textsf{then}\ \{af;\ cf;\ a = a+1;\ b = 0;\ af;\ cf\} \\
& \qquad\qquad\qquad\quad \textsf{else}\ \{af;\ cf;\ b = b+1;\ af;\ cf\ \};\ af \\
\equiv\ & \{\text{Equation } (13)\} \\
& cf;\ \mathcal{O}(\textsf{if}\ ((v \,\%\, 10) == 9)\ \textsf{then}\ \{cf;\ a = a+1;\ b = 0;\ af\}\ \textsf{else}\ \{cf;\ b = b+1;\ af\});\ af \\
\equiv\ & \{\text{definitions and Equation } (9)\} \\
& \textsf{if}\ ((v \,\%\, 10) == 9)\ \textsf{then}\ \{a = v/10;\ b = v \,\%\, 10;\ a = a+1;\ b = 0;\ v = 10 * a + b\} \\
& \qquad\qquad\qquad\qquad\quad \textsf{else}\ \{a = v/10;\ b = v \,\%\, 10;\ b = b+1;\ nc = 10 * a + b\} \\
\equiv\ & \{\text{simultaneous equations in branches}\} \\
& \textsf{if}\ ((v_0 \,\%\, 10) == 9) \\
& \textsf{then}\ \{a_1 = v_0/10;\ b_1 = v_0 \,\%\, 10;\ a_2 = a_1 + 1;\ b_2 = 0;\ v_1 = 10 * a_2 + b_2\} \\
& \textsf{else}\ \{a_3 = v_0/10;\ b_3 = v_0 \,\%\, 10;\ b_4 = b_3 + 1;\ v_2 = 10 * a_3 + b_4\} \\
\equiv\ & \{\text{substitutions}\} \\
& \textsf{if}\ ((v_0 \,\%\, 10) == 9)\ \textsf{then}\ \{v_1 = 10 * (v_0/10) + 10\} \\
& \qquad\qquad\qquad\qquad\qquad \textsf{else}\ \{v_2 = 10 * (v_0/10) + (v_0 \,\%\, 10) + 1\} \\
\equiv\ & \{\text{modular arithmetic and convert back to assignments}\} \\
& \textsf{if}\ ((v \,\%\, 10) == 9)\ \textsf{then}\ \{v = v+1\}\ \textsf{else}\ \{v = v+1\} \\
\equiv\ & \{\text{identical branches}\} \\
& v = v + 1
\end{aligned}
$$

Figure 5: Proof of correctness for a transformation using the variable split

4.4.1 Reordering array indices

Let us consider how to specify array transformations which change the array indices and so suppose that we want to permute the array indices. For a permutation we need an bijective function

$$p :: [0..n) \to [0..n)$$

where n is the size of the array. This function ensures that we do not have any array index clashes and the size of the array is not changed. Here is example permutation given in [15]:

$$p = \lambda i.(a \times i + b \pmod{n})\ \text{where}\ gcd(a, n) = 1$$

Folding and flattening can also be considered to be transformations that change an array index. For example, suppose that we have an array A of size n and an array R of size $p \times q$ (where $p \times q = n$). One way to convert between A and R is

$$A[i] = R[i \text{ div } q, i \text{ mod } q]$$

and the inverse of this transformation is:

$$R[j, k] = A[j * q + k]$$

How can we write a conversion function for these kinds of transformation? We need to consider how $A[0], A[1], \ldots, A[n-1]$ are all transformed. So we could give a set of n transformations (one for each element of the array) but in a program the index for an array is usually a variable (or an expression). So we need to write an expression for $A[j]$, where $j \in [0..n)$, which shows how the array is transformed. Note that this is not a variable transformation of j as j is merely a dummy variable acting as a placeholder. When using such an expression at a particular point we need to instantiate j with an expression for the array index.

If we want to transform the array A into the array R using an index change function f then the conversion function would be:

$$cf \equiv R[f(j)] = A[j]$$

and the abstraction function is:

$$af \equiv A[j] = R[f(j)]$$

Suppose that we want to transform the statement

$$A[i] = A[i-1] + 1$$

Using Equation (12) then we have:

$$\begin{array}{l} A[j] = R[f(j)]; \\ A[i] = A[i-1] + 1; \\ R[f(j)] = A[j] \end{array}$$

When converting to a set of simultaneous equation, we use the normal subscripts to denote new assignments on A, R and i but not on the dummy variable j. Note that an expression of the form $A[i] = \ldots$ is an assignment of A and not i.

From this program we have the following set of simultaneous equation:

$$\begin{array}{l} A_1[j] = R_0[f(j)]; \\ A_2[i_0] = A_1[i_0 - 1] + 1; \\ R_1[f(j)] = A_2[j] \end{array}$$

When substituting the expression for A_1 we instantiate j with the expression $i_0 - 1$ to obtain:

$$\begin{array}{l} A_1[j] = R_0[f(j)]; \\ A_2[i_0] = R_0[f(i_0 - 1)] + 1; \\ R_1[f(j)] = A_2[j] \end{array}$$

When substituting A_2 we replace j with i_0 to get:

$$\begin{array}{l} A_1[j] = R_0[f(j)]; \\ A_2[i_0] = R_0[f(i_0 - 1)] + 1; \\ R_1[f(i_0)] = R_0[f(i_0 - 1)] + 1 \end{array}$$

Now removing the redundant assignments to A_1 and A_2 and converting back to sequential code we have:

$$R[f(i)] = R[f(i-1)] + 1$$

Note that since i is unchanged by this transformation then we could just use i instead of i_0 in the proof above.

4.4.2 Array splitting

An array split aims to split an array A (of size n) into two new arrays P (of size m_p) and Q (of size m_q). This idea was generalised in [15] as follows. For an array split which uses two new arrays, we need three functions ch (called the *choice function*), f_p and f_q (these functions determine the positions of the elements in each of the arrays). The types of the functions are as follows:

$$\begin{aligned} ch &:: [0..n) \to \mathbf{B} \\ f_p &:: [0..n) \to [0..m_b) \\ f_q &:: [0..n) \to [0..m_c) \end{aligned}$$

The relationship between A and P and Q is given by the following rule:

$$A[i] = \begin{cases} P[f_p(i)] & \text{if } ch(i) \\ Q[f_q(i)] & \text{otherwise} \end{cases}$$

Note that we can only apply this transformation to statements that use A with an index. For example, we could not easily transform statements which pass the array A to other methods — we would also have to transform the method which "receives" the array A.

In [11], an example array split was given in which one of the new arrays contained the elements of A, in order, which had an even index and the other array contained the rest of the elements. For this split, we can define

$$\begin{aligned} ch &= (\lambda i.i \,\%\, 2 == 0) \\ f_p = f_q &= (\lambda i.i/2) \end{aligned}$$

In [15], an example program for producing Fibonacci numbers using arrays was obfuscated using the example array split from [11]. For this obfuscation, the statement

$$B \equiv A[i] = A[i-1] + A[i-2] \tag{16}$$

was transformed to:

$$\begin{aligned} &\textsf{if } (i \,\%\, 2 == 0) \textsf{ then } P[i/2] = Q[(i-1)/2] + P[(i-2)/2] \\ &\qquad\qquad\qquad\quad\ \textsf{else } \ Q[i/2] = P[(i-1)/2] + Q[(i-2)/2] \end{aligned} \tag{17}$$

Is this transformation correct? Let us derive a correct obfuscation for the statement (16) using the generalised array split.

We can write a conversion function for the generalised array split as follows

$$cf \equiv \textsf{if } (ch(j)) \textsf{ then } P[f_p(j)] = A[j] \textsf{ else } Q[f_q(j)] = A[j]$$

and so the abstraction function can be written as

$$af \equiv \textsf{if } (ch(j)) \textsf{ then } A[j] = P[f_p(j)] \textsf{ else } A[j] = Q[f_q(j)]$$

Note that when we use these functions we will have to instantiate the index j to a particular value (or expression). As B (in Equation (16)) is an assignment we can use

```
    $af; B; cf$
$\equiv$  {definitions}
    if $(ch(j))$ then $A[j] = P[f_p(j)]$ else $A[j] = Q[f_q(j)]$;
    $A[i] = A[i-1] + A[i-2]$;
    if $(ch(j))$ then $P[f_p(j)] = A[j]$ else $Q[f_q(j)] = A[j]$
$\equiv$  {convert to simultaneous equations}
    if $(ch(j))$ then $A_1[j] = P_0[f_p(j)]$ else $A_1[j] = Q_0[f_q(j)]$;
    $A_2[i] = A_1[i-1] + A_1[i-2]$;
    if $(ch(j))$ then $P_1[f_p(j)] = A_2[j]$ else $Q_1[f_q(j)] = A_2[j]$
$\equiv$  {substitute value for $A_1$ with $j = i-1$}
    if $(ch(j))$ then $A_1[j] = P_0[f_p(j)]$ else $A_1[j] = Q_0[f_q(j)]$;
    if $(ch(i-1))$ then $A_2[i] = P_0[f_p(i-1)] + A_1[i-2]$
                   else $A_2[i] = Q_0[f_q(i-1)] + A_1[i-2]$;
    if $(ch(j))$ then $P_1[f_p(j)] = A_2[j]$ else $Q_1[f_q(j)] = A_2[j]$
$\equiv$  {substitute value for $A_1$ with $j = i-2$}
    if $(ch(j))$ then $A_1[j] = P_0[f_p(j)]$ else $A_1[j] = Q_0[f_q(j)]$;
    if $(ch(i-1))$ then { if $(ch(i-2))$
          then $A_2[i] = P_0[f_p(i-1)] + P_0[f_p(i-2)]$
          else  $A_2[i] = P_0[f_p(i-1)] + Q_0[f_q(i-2)]$}
        else { if $(ch(i-2))$ ...}
    if $(ch(j))$ then $P_1[f_p(j)] = A_2[j]$ else $Q_1[f_q(j)] = A_2[j]$
$\equiv$  {substitute values in $P_1$ and $Q_1$ with $i = j$}
    ...
    if $(ch(i))$ then{if $(ch(i-1))$ then
        {if $(ch(i-2))$ then $P_1[f_p(i)] = P_0[f_p(i-1)] + P_0[f_p(i-2)]$
                        else ... } else { ... }
      else{if $(ch(i-1))$ then
        {if $(ch(i-2))$ then $Q_1[f_q(i)] = P_0[f_p(i-1)] + P_0[f_p(i-2)]$
                        else ... } else{ ... }
$\equiv$  {sequential code (removing redundant assignments)}
     if $(ch(i))$ then { if $(ch(i-1))$
         then { if $(ch(i-2))$ then $P[f_p(i)] = P[f_p(i-1)] + P[f_p(i-2)]$
                               else  $P[f_p(i)] = P[f_p(i-1)] + Q[f_q(i-2)]$}
         else  { if $(ch(i-2))$ then $P[f_p(i)] = Q[f_q(i-1)] + P[f_p(i-2)]$
                               else $P[f_p(i)] = Q[f_q(i-1)] + Q[f_q(i-2)]$}}
       else { if $(ch(i-1))$
         then { if $(ch(i-2))$ then $Q[f_q(i)] = P[f_p(i-1)] + P[f_p(i-2)]$
                               else  $Q[f_q(i)] = P[f_p(i-1)] + Q[f_q(i-2)]$}
         else  { if $(ch(i-2))$ then $Q[f_q(i)] = Q[f_q(i-1)] + P[f_p(i-2)]$
                               else $Q[f_q(i)] = Q[f_q(i-1)] + Q[f_q(i-2)]$}}
```

Figure 6: A derivation for an array split

Equation (12) to derive a correct obfuscation for B by computing $af; B; cf$ — a sketch of the derivation can be seen in Figure 6

From this proof we have a general form for $\mathcal{O}(B)$ and so let us consider the example array split from [11]. We can simplify the expression for $\mathcal{O}(B)$ for this particular split by removing infeasible paths. For example, when we have a statement of the form:

$$\textsf{\textbf{if}}\ (ch(i))\ \textsf{\textbf{then}}\ \{\textsf{\textbf{if}}\ (ch(i-1))\ \textsf{\textbf{then}}\ X\ \textsf{\textbf{else}}\ Y\}$$

then X cannot be reached since $ch = (\lambda i.i \,\%\, 2 == 0)$ when $ch(i)$ is True then $ch(i-1)$ must be False.

By removing all the infeasible paths, we obtain the following definition for $\mathcal{O}(B)$.

$$\begin{array}{l} \textsf{\textbf{if}}\ (ch(i))\ \textsf{\textbf{then}}\ P[f_p(i)] = Q[f_q(i-1)] + P[f_p(i-2)] \\ \qquad\qquad\quad \textsf{\textbf{else}}\ \ Q[f_q(i)] = P[f_p(i-1)] + Q[f_q(i-2)] \end{array}$$

Now putting in the functions ch, f_p and f_q we obtain

$$\begin{array}{l} \textsf{\textbf{if}}\ (i\%2 == 0)\ \textsf{\textbf{then}}\ P[i/2] = Q[(i-1)/2] + P[(i-2)/2] \\ \qquad\qquad\qquad\quad \textsf{\textbf{else}}\ \ Q[i/2] = P[(i-1)/2] + Q[(i-2)/2] \end{array}$$

This matches up to the statement in Equation (17) and so the transformation given in [15] was correct.

5 Experimental Design

Now that we have set up the refinement framework and outlined some potential obfuscating transforms, we can describe our the background to our experiments in which we use slicing to design obfuscations. The motivation of our experiments is derived from the difficulty of empirically evaluating the obfuscatory strength of seemingly resilient obfuscating transforms. Majumdar *et al.* [30] (and separately by Udupa *et al.* [39]) 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 showed that it is impossible to come up with one general purpose reverse engineering tool for deobfuscating *all* possible obfuscating transforms (thinking from an adversary's perspective, an adversary will have to use different techniques to deobfuscate a program obfuscated with different kinds of transforms). As an immediate consequence of this observation, we can argue that it is also 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], to evaluate our obfuscations which are designed to resist slicing. It runs algorithms on system dependence graphs (SDGs), an intermediate structure for representing programs [24]. CodeSurfer is capable of backward slicing, forward slicing, and chopping. Backward and forward slicing have been explained in a previous section. A chop includes all program points that are affected between a source program point and a sink program point. For our experiments, we restrict ourselves to backward slices on output statements of programs.

It has been claimed in the existing literature [12] 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. 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 or 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 [30]. The only well-known static slicer for Java programs is Indus [26]; 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 Horwitz *et al.* [24]) and has been extensively evaluated in published literature [6, 7, 32].

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 will have a set of output variables V_O. Our slicing criterion will be the last output statement for each output variable. For each $v_i \in V_O$ the backwards slice for v_i is denoted by SL_i, SL_{int} is defined to be $\bigcap_i SL_i$ (*i.e.* the intersection of all the slices) and $|\ldots|$ denotes the size.

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 [14] 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.

5.1 Metrics

We would like to be able to obfuscate programs so that it makes slicing less effective as a tool for program comprehension. Meyers and Binkley [32] studied some slice-based metrics which were proposed as measures of the quality of software. Two of the metrics (*Tightness* and *Coverage*) were originally presented by Weiser [41] and the other two (*MinCoverage* and *MaxCoverage*) were proposed by Ott and Thuss [33]. Here are the definitions taken from [32] for the four slice-based metrics that we will use.

Tightness Tightness measures the number of statements in every slice.

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

Minimum Coverage Minimum coverage is the ratio of the smallest slice in a method to the method's length.

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

Coverage Coverage compares the length of slices to the length of the entire method.

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

Maximum Coverage Maximum coverage is the ratio of the largest slice in a method to the method's length.

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

In Section 7 we will use these metrics to evaluate how our obfuscations affect the effectiveness of slicing. Our aim when performing slicing obfuscations is to increase some (and ideally *all*) of the slicing metrics. We use CodeSurfer to compute the backward slice for each output variable. To compute the metrics for each slice we need to measure the size of the method and well as the size of the slices in a consistent manner. CodeSurfer has a feature which allows the user to compute sets of program points (*i.e.* nodes in the SDG) using a *set calculator*. We can use this calculator to compute measurements for slices as well as for methods and we can perform operations on the sets.

For a particular method M, the size $|M|$ is obtained by computing the number of program points contained within the method itself (and not any that are contained in calls to other methods). The set of program points for a backwards slice from a point in M may contain points from other methods, such as method calls, and so the size of a slice may actually be bigger than $|M|$. Since we are only concentrating on a single method M, when computing the size of the slice we will only consider the part of the slice that is contained in M. (Using the CodeSurfer calculator we can compute the intersection of the slice and the points from M.)

5.2 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 simply 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 $v_i \in V_O$ (the set of output variables) we 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 1 (Definition)**.** An obfuscation $\mathcal{O}$ is a **slicing obfuscation** for a program P and a variable V_i if it decreases the number of orphaned points (the size of the residue), *i.e.*

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

Using inspiration from the *Tightness*, *MinCoverage*, *Coverage* and *MaxCoverage* metrics given in [33], 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, $RES_{un} = M \backslash SL_{int}$.

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

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

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 the ratio of the largest residue in a method to the method's length.

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

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

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

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

Figure 7: 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).

Comparing our metrics to those in [33], we find that

$$\begin{aligned} Min(M) &= 1 - MaxCoverage(M) \\ D(M) &= 1 - Coverage(M) \\ Max(M) &= 1 - MinCoverage(M) \\ C(M) &= 1 - Tightness(M) \end{aligned}$$

and by the definitions, we see that

$$Min(M) \leq D(M) \leq Max(M) \leq C(M) \tag{18}$$

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.

6 Applying Transformations to an Example

In this section, we will discuss some program transformations that are suitable as slicing obfuscations. We will use some of the data transformations given in Section 4 and also give some control-flow transformations. To help us to explain how these transformations operate we will use a running example.

6.1 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 7. 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 7 the underlined points denote the backwards slice from nl given by CodeSurfer.

Method M	$\|M\|$	$\|V_O\|$	Residue size nl	nw	nc	$\cup$	$Min(M)$	$D(M)$	$Max(M)$	$C(M)$
original	32	3	21	14	22	26	43.8%	59.4%	68.8%	81.3%
bogus	38	3	8	8	8	9	21.1%	21.1%	21.1%	23.7%
*encode*1	35	3	24	17	7	29	20.0%	45.7%	68.6%	82.9%
*encode*3	42	3	8	8	10	11	19.0%	20.6%	23.8%	26.2%
loop	36	3	7	7	7	8	19.4%	19.4%	19.4%	22.2%
split	44	3	9	9	9	10	20.5%	20.5%	20.5%	22.7%
array	28	3	7	7	7	8	25.0%	25.0%	25.0%	28.6%

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

We can see that the residue for nl contains program points for both 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.

We discuss several techniques to decrease the size of the residues for the output variables and thus decrease the effectiveness of slicing. In Table 1 we summarise the results of our different transformations of 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 residue (calculated using CodeSurfer by subtracting the slice size from the method size), the size of the union and the results for our four residue metrics from Section 5.2. The top row of the table displays the measurements for the original Word Count method.

The result of Table 1 is represented pictorially by the bar chart of Figure 8. A distinctive pattern of decreasing residue percentages in the figure indicates better slicing obfuscations.

6.2 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

$$x = G;\ \textsf{if}\ (p^F)\ y = H(x);$$

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:

$$x = G;\ S; \quad \rightsquigarrow \quad x = G;\ \textsf{if}\ (q^T)\ S;\ \textsf{else}\ y = H(x); \tag{19}$$

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 (19) is:

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

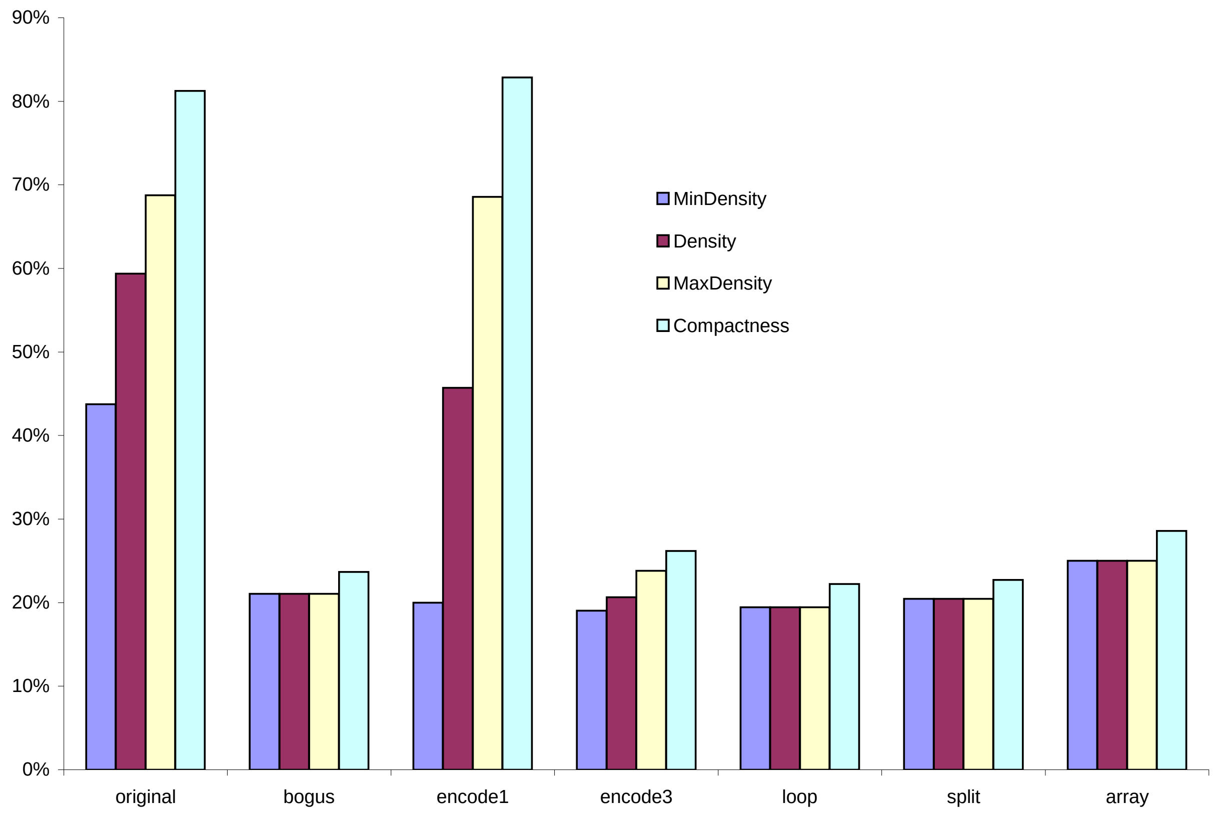

Figure 8: Bar Chart showing the residue metrics for our Word Count examples

For Word Count, we added this simple bogus predicate

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

at the end of the **while** loop. This predicate adds dependencies between the three variables and so the slice for each variable contains the definitions for the other variables. Since nc is incremented for every character and nw and nl are only incremented for certain characters we have the following invariant:

$$nc \geq nw \wedge nc \geq nl \tag{20}$$

In Figure 9 we can see this method with the backwards slice from nc.

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.

6.3 Variable Encoding

In Section 4.2 we gave an encoding in which a variable x is transformed so that it seems to depend on a variable y:

$$x \rightsquigarrow \alpha * x + \beta * y$$

For Word Count, we will perform the following encoding:

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

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

Figure 9: Addition of a bogus predicate (with the $\underline{\text{backwards slice}}$ for nc).

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

Figure 10: Simple variable encoding (with the $\underline{\text{backwards slice}}$ for nc).

By the rewrite rules in Section 4.2, we can prove that, for example,

$$\begin{aligned} nc\,{+}{+} &\rightsquigarrow nc\,{+}{+} \\ nw\,{+}{+} &\rightsquigarrow \{nw\,{+}{+};\ nc\,{-}{-};\} \end{aligned}$$

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 10 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:

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

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

Figure 11: Three variable encodings (with the backwards slice 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 11. 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 $encode3$ the size has increased by 31%. However, we have reduced the sizes of the three residues and so the metrics values have decreased.

6.4 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:

$$\textbf{while}\ (c)\ S \leadsto \textbf{while}\ (c\ \wedge p)\ S$$
$$\textbf{while}\ (c)\ S \leadsto \textbf{while}\ (c\ \vee q)\ S$$

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.*

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

```
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 12: Adding a new loop variable (with backwards slice from nw)

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.*

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

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 (20), we can see that the value of j is always non-negative in the loop and so we change the loop header to

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

We then add our assignment for j into the loop. The full method can be seen in Figure 12. 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.

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

6.5 Using a Variable Split

In Section 4.3 we gave details of how to perform a variable split. So for Word Count, we will split nc into two other integers by using $nc \rightsquigarrow \langle nc \,/\, 10,\ nc \,\%\, 10\rangle$ and let $a = nc \,/\, 10$ and $b = nc \,\%\, 10$ — we have that $0 \leq b \leq 9$ which we take to be our invariant. We take $f_1 = \lambda i.\ i \,/\, 10$, $f_2 = \lambda i.\ i \,\%\, 10$ 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 **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.

In Figure 13 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.

6.6 Arrays

So far we have consider only 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:

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

```
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 14: Transformation to arrays (with the backwards slice for $a[0]$).

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 14 we can 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 [11], 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.

7 Further Examples

In the previous section we gave details of how to produce slicing obfuscations for the Word Count program but can we use these techniques to produce slicing obfuscations for other programs? We now use some of some of the data and control-flow obfuscations given in Section 6 to transform four more C programs. In the rest of this section we briefly describe each of the programs, the obfuscations that we performed and the slice sizes we obtained using CodeSurfer. In the final section (Section 7.5) we summarise the results of our experiments which can be seen in Table 2. Each row of the table contains the results for a particular method: we have recorded the size of the method, the size of each slice with respect to a particular variable and the size of the intersection of each of the slices (all of these values were computed using the set calculator of CodeSurfer). Finally we have computed the values for the four slicing metrics (mentioned in Section 5.1) as percentages.

Method M	$\|M\|$	$\|V_O\|$	For each v_i, the slice size $\|SL_i\|$						$\|SL_{int}\|$	$T(M)$	$MinC(M)$	$Cov(M)$	$MaxC(M)$
ps	21	2	$prod$	12	sum	12			7	33.3%	57.1%	57.1%	57.1%
$psObf1$	22	2	$prod$	16	sum	13			11	50.0%	59.1%	65.9%	72.7%
$psObf2$	26	2	$prod$	19	sum	19			17	65.4%	73.1%	73.1%	73.1%
$search$	107	2	n	9	$secs$	11			2	1.9%	8.4%	9.3%	10.3%
$searchObf1$	120	2	n	45	$secs$	11			10	8.3%	9.2%	23.3%	37.5%
$searchObf2$	127	2	n	49	$secs$	48			46	36.2%	37.8%	38.2%	38.6%
rov	124	2	$fuel$	23	$dist$	46			19	15.3%	18.5%	27.8%	37.1%
$rovObf1$	129	2	$fuel$	60	$dist$	46			45	34.9%	35.7%	41.1%	46.5%
$rovObf2$	132	2	$fuel$	62	$dist$	60			59	44.7%	45.5%	46.2%	47.0%
$rovObf2np$	94	2	$fuel$	62	$dist$	60			59	62.8%	63.8%	64.9%	66.0%
$scatter$	143	3	si	116	ru	111	i	9	8	5.6%	6.3%	55.0%	81.1%
$scatterObf1$	148	3	si	132	ru	132	i	132	131	88.5%	89.2%	89.2%	89.2%
$scatterObf2$	150	3	si	139	ru	139	i	139	138	92.0%	92.7%	92.7%	92.7%

Table 2: Table showing the slicing metrics values for four example programs

7.1 Product and Sum

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

We first created a dependency for $prod$ on sum by adding the condition of $\vee sum < 0$ in the guard of the **while** loop. Then we added a bogus predicate p^T around the statement $prod = prod * i$. The metrics values for three methods can be seen in Table 2. The method ps relates to the original obfuscated method, in $psObf1$ the loop dependency was created and finally we created the bogus predicate in $psObf2$. Details of some more slicing obfuscations for the product and sum program can be found in [17].

7.2 Search Sort

The search sort program (obtained from [40]) takes an argument n from the user, performs different sorts and searches on n elements and then displays the time taken to do each one. So the two output variables at each stage are the number of elements n and the number of seconds $secs$. This program is different from those considered so far as it contains various methods which the main method calls. However the results for all of these methods are discarded and only the time taken to perform each method is computed. Thus slices for both n and $secs$ only contain statements from the main method. So for our experiments we only consider changing statements in the main method. (Obviously when obfuscating we should aim to obfuscate the other methods in the program but this is beyond the scope of this contribution.) The metric results for the main method $search$ can be found in Table 2.

As n is constant throughout the program (it is inputted by the user) we attempt to make n vary by adding a variable encoding. We would like to create a dependency for n on sec but n is declared as an integer and $secs$ is a float. So we declare a new integer variable k which we define as $k = (\textbf{int})\ 10 * secs$ and we perform the transformation $n \rightsquigarrow n + k$. By the replacement rules for variable encoding we need to redefine k every time that $secs$ is redefined and so we can use a different declaration for k. Note that this kind of transformation could make the value of n overflow and so we should put in checks to ensure that this does not happen. The results for this obfuscation can be found in Table 2 for the method $searchObf1$ and we can see that we have significantly increased the slice size for i but the size for $secs$ is unchanged.

To create some dependencies for $secs$ we place two bogus predicates near the end of the method. The value of $secs$ is obtained in the following way:

$c1 = \textbf{clock()};$
$search\_method();$
$c2 = \textbf{clock()};$
$secs = c2 - c1;$

So, for example, we can change the first assignment to:

if $(secs >= 0)\ c1 = \textbf{clock()};$ **else** $c1 = bogus(n);$

The results after the addition of these predicates can be found in Table 2 for the method $searchObf2$ and we can see that we have increased the slice size for $secs$.

7.3 Rover

The rover program checks whether a plan for manoeuvring a vehicle around an obstacle to a given target, with a limited amount of fuel, satisfies certain constraints. It simulates a land rover vehicle which needs to be driven around a rock on the Martian surface by giving it coordinates as input [40]. However, the vehicle goes off-track from the specified target and it has limited amount of fuel. The challenge is to bring the vehicle within 5km of the target by inputting a sequence of coordinates which will also not exhaust the fuel before it reaches the target. Here we consider two output variables $fuel$ and $dist$. The results for the original method can be seen in the rov row of Table 2.

So that $fuel$ depends on $dist$ we perform the following variable encoding $fuel \rightsquigarrow fuel + dist$. Note that we have to add an extra temporary variable t to perform the encoding. The values for this obfuscation can be found in the $rovObf1$ row of Table 2.

The value of $dist$ is computed as follows

$$dist = \sqrt{dx^2 + dy^2}$$

where dx and dy are two other variables. To create a dependency for $dist$ we changed an assignment to dx as follows:

$$dx = E; \rightsquigarrow \textbf{if } (fuel >= dist)\ dx = E;\ \textbf{else } dx = bogus;$$

By the variable encoding above we know that since $dist \geq 0$ then $fuel \geq dist$. The row $rovObf2$ in Table 2 contains the values for this obfuscation.

The $rover$ example contains many **printf** statements which cannot be contained in any slice. So to find out a more accurate evaluation of our obfuscation we removed all but the final **printf** statements and the result for this method can be found in the $rovObf2np$ in Table 2.

7.4 Scattering

The scattering program is a typical physics problem dealing with the famous Rutherford's scattering experiment for finding the size of the nucleus of an atom (from Tao Pang's book "An Introduction to Computational Physics" [35]). The original program contained several procedures for performing integration using Simpson's rule, finding roots using the Secant method, and finding the first and second order derivatives with the three-point formula. We flattened the procedures by inlining them in the main() procedure (which we call $scatter$) and we consider three output variables si, ru and i.

We observe in the program source code that the statement $b = b0 + i * db$; maintains the invariant $i \geq b \ \wedge \ b0 \leq b$. Our first obfuscation consists of performing the following transformations by introducing bogus predicates:

$$si = log(sig[i]); \ \rightsquigarrow \ \textbf{if } (i + 1 > b)\ si = log(sig[i]);\ \textbf{else } si = bogus;$$

$$ru = log(ruth); \ \rightsquigarrow \ \textbf{if } (b0 <= b)\ ru = log(ruth);\ \textbf{else } ru = bogus;$$

in the code. The row *scatterObf*1 in Table 2 contains the values for this obfuscation. Interestingly, with the addition of 5 more SDG nodes, the slice sizes for variables *si*, *ru* and *i* increases from 116 nodes, 111 nodes and 9 nodes respectively to 132 nodes. In an attempt to provide maximum code coverage, we introduce another similar bogus predicate to include an additional 7 nodes in each of the slices of *si*, *ru*, and *i*. The row *scatterObf*2 in Table 2 indicates the changes brought about by this obfuscating transform.

7.5 Results

We can easily see that, for each obfuscation, we have increased the values of *all* the slicing metrics and we can also see that we do not significantly increase the size of the methods (the worst is a 24% increase for ps but the size of the slices increase by 58%). The metric values for the search sort obfuscations are generally much lower than those for the other programs. This is because the search sort program uses different searching and sorting algorithms contained within other methods and we have only considered intra-procedural slices. We decided to flatten the scattering program before obfuscating and so we were able to obfuscate the program more successfully.

In Table 2 we gave the results for our experiments using the slicing metrics mentioned in Section 5.1. Using these results we can compute the values for our residue metrics (given in Section 5.2) and these values can be seen in Table 3. We can see that we have managed to reduce the residue metrics values which was our stated aim. One aspect that is key to decreasing the residue metrics values is to ensure that we decrease $|RES_{un}|$ (*i.e.* the union of all the residues). Decreasing the size of the union of the residues means that we decrease the value of the compactness metrics and, by Equation (18), we are likely to decrease the values of the other residue metrics. To reduce the union of the residues (which is equivalent to increasing the intersection of the slices), we need to add dependencies between all of the variable in V_O.

The result of Table 3 is represented graphically using a bar chart in Figure 15. This figure groups the residue metrics for each of our example programs (12 programs grouped in 4 categories). We can see from Figure 15, a better slicing obfuscation reduces the percentage of the resides that are left behind after slicing.

8 Applying the transformations

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

8.1 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

Method M	$\|M\|$	$\|V_O\|$	For each v_i, the size of the residue						$\|RES_{un}\|$	$Min(M)$	$D(M)$	$Max(M)$	$C(M)$
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%
$rovObf2np$	94	2	$fuel$	32	$dist$	34			35	34.0%	35.1%	36.2%	37.2%
$scatter$	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 3: Table of residue metrics values for four example C programs

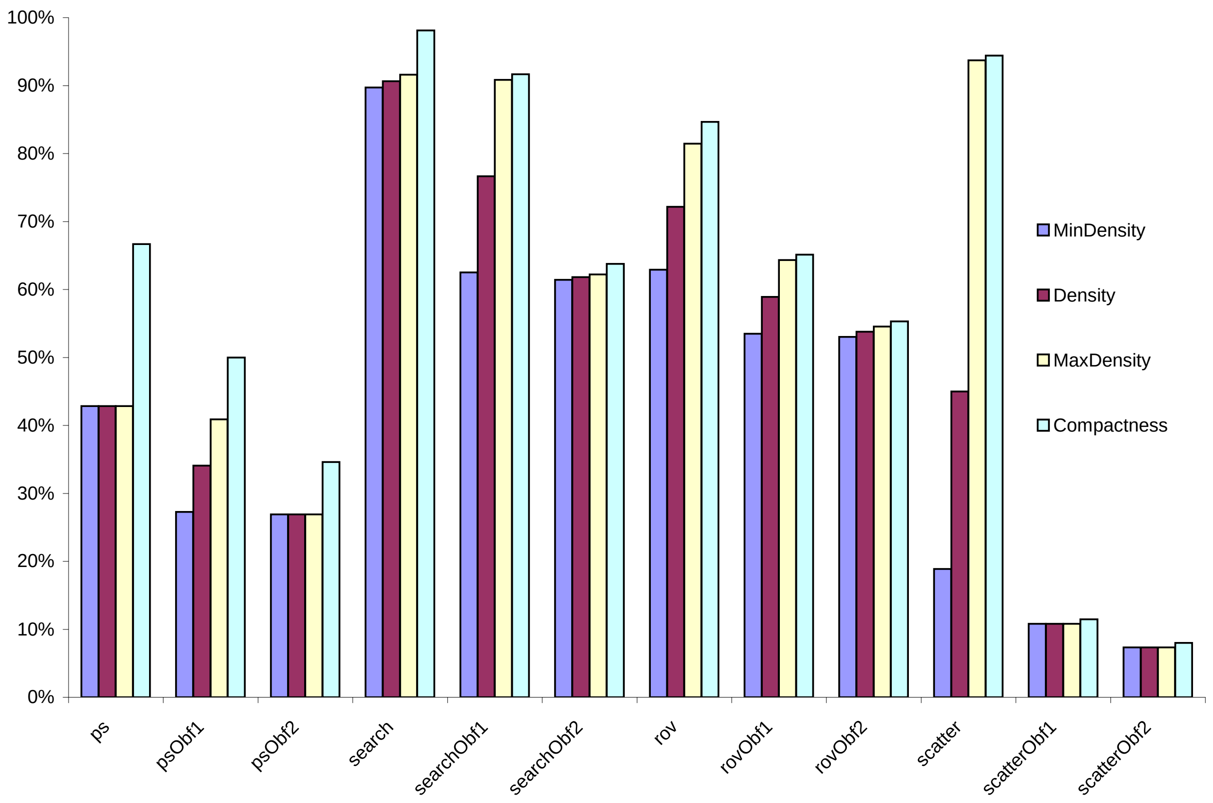

Figure 15: Bar Chart showing residue metric values for all our example programs

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 (19). 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:

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

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* [12]. 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 6.4 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:

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

Thus we can now apply our loop transformations.

8.2 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.

8.3 Localising the transformations

The variable obfuscations proposed in [11] and [16] 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 "deobfuscate" $\mathcal{O}(B)$.

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 $af_B;\ cf_C$.

8.4 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 one of our array obfuscations given in Section 4.4.1. 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 example we considered three output variables and so sometimes it was necessary to add more than obfuscation. For example, for *encode*3 (from Section 6.3) we use three different encodings (one for each output variable). This means that, in effect, we have create a composite obfuscation. Further work is needed study the effects of composing obfuscations together and, in particular, does the order in which we add obfuscations (and the order in which we consider the output variables) matter?

9 Conclusions

In this contribution, we have conducted experiments in which we considered adding obfuscations that were targetted to be more resistant to slicing. We have 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 and Table 3 we saw that, according to our residue metrics, we have successfully created transformations to reduce the size of residues. Since our metrics are related to the slicing based metrics from [33], our obfuscations appear to make slicing less useful. The results for the single variable encoding *encode*1 for the Word Count example highlight the importance of ensuring that when obfuscating we consider the slices for all variables. We have only conducted relatively small experiments but it has shown us how to use obfuscations to decrease the effectiveness of slicing.

In our experiments we only used simple obfuscations to illustrate the techniques used to create slicing obfuscations. Even with these simple transformations we have still managed to decrease the effectiveness of slicing which was our stated goal. When faced with an attacker who is armed with more than just a slicer we will obviously have to design more complicated transformations. This will involve creating predicates that are harder for an attacker to understand, using different program constructs such as pointers and dealing with inter-procedural constructs.

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.

Our obfuscations were created manually and in Section 8 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. 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.

References

[1] Business Software Alliance. Second annual BSA and IDC software piracy study, May 2005. Available from URL:
`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] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):37–61, 1985.

[5] David Binkley, Nicolas Gold, and Mark Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2):8, 2007.

[6] 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.

[7] 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.

[8] David Binkley and Mark Harman. A survey of empirical results on program slicing. *Advances in Computing*, 62:105–178, 2004.

[9] 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`.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[19] Margaret Ann Francel and Spencer Rugaber. The value of slicing while debugging. *Science of Computer Programming*, 40(2-3):151–169, 2001.

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

[21] Rajiv Gupta, Mary Jean Harrold, and Mary Lou Soffa. An approach to regression testing using slicing. In *Proceedings of theConference on Software Maintenance (CSM 1992)*, pages 299–308, Orlando, FL, November 1992. IEEE Computer Society.

[22] Tommy Hoffner, Mariam Kamkar, and Peter Fritzson. Evaluation of program slicing tools. In *Automated and Algorithmic Debugging*, pages 51–69, 1995.

[23] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 1990. ACM Press.

[24] 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.

[25] Kirill S. Ivanov and Vladimir A. Zakharov. Program obfuscation as obstruction of program static analysis. In *Volume 6. ISPRAN 2005. Russian Academy of Sciences Technical Report Series*, pages 137–156. ISPRAN 2005, 2005.

[26] 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.

[27] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56, London, UK, 2001. Springer-Verlag.

[28] Bogdan Korel and Janusz W. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[29] James Robert Lyle. *Evaluating variations on program slicing for debugging.* PhD thesis, University of Maryland, College Park, USA, 1984.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, New York, NY, USA, 1984. ACM Press.

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

[36] 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.

[37] 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.

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

[39] 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.

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

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

[42] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.