Specifying Imperative Data Obfuscations

Stephen Drape, Clark Thomborson and Anirban Majumdar

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

Abstract. An obfuscation aims to transform a program, without affecting the functionality, so that some secret information within the program can be hidden for as long as possible from an adversary. Proving that an obfuscating transform is correct (*i.e.* it preserves functionality) is considered to be a challenging task.
In this paper we show how data refinement can be used to specify imperative data obfuscations. An advantage of this approach is that we can establish a framework in which we can prove the correctness of our obfuscations. We demonstrate our framework by considering some examples from obfuscation literature. We show how to specify these obfuscations, prove that they are correct and produce generalisations.
Keywords Data Obfuscation, Refinement, Specification, Correctness

1 Introduction

Skype's internet telephony client [2], SDC Java DRM (according to [11]), and most software license-control systems rely heavily on obfuscation for their security. After the landmark proof of Barak *et al.* [1], there seems little hope of designing a perfectly-secure software black-box, for any broad class of programs. To date, no one has devised an alternative to Barak's model, in which we would be able to derive proofs of security for systems of practical interest. These theoretical difficulties do not lessen practical interest in obfuscation, nor should it prevent us from placing appropriate levels of reliance on obfuscated systems in cases where the alternative of a hardware black-box is infeasible or uneconomic.

In this paper we define obfuscation as a heuristic method whose objective is to transform a program, without affecting relevant aspects of its functionality, in such a way that some secret information in the program can be preserved as long as possible from some set of adversaries. The second clause in our objective implies that theoretical study of the effectiveness of an obfuscation will be impossible until we have a validated, and theoretically-tractable, model of adversarial attack. The first clause is, by contrast, an appropriate domain for theoretical study. We expect our compilers to accurately preserve program semantics when they transform our source codes into object codes. We have a similar expectation of obfuscating compilers and object-code obfuscators. Theoretical study of the correctness of obfuscating systems is as yet in its infancy. In this paper we describe a promising approach for specifying obfuscating transforms for imperative

languages. Our approach allows us to establish a framework for the proving the correctness of data obfuscations which we illustrate by constructing correctness proofs for some examples from [3] and [8] which were stated without proof.

In [7] obfuscation is considered to be data refinement [6] and obfuscations for abstract data-types were developed using a functional language (Haskell [10]). In this paper we extend the data refinement approach to imperative data obfuscations. We will consider programs consisting of assignments, conditionals and loops and we model these statements as functions that change the state. Thus we will be able to specify data obfuscations as functional refinements. As a consequence, we find that not only can we prove the correctness of *all* our data obfuscations but we can also specify more general obfuscations and study the effects of composing different obfuscating transforms. Thus our approach may someday be used as a method for generating obfuscated programs.

As stated earlier an obfuscation should preserve some secret information but what do we mean by this? In [7] a function (operation) was said to be obfuscated if it is harder to prove properties (*i.e.* assertions) about the function. Thus, in that case, the goal of obfuscation was to keep a set of assertions secret. It is beyond the scope of this paper to develop this notion for imperative programs — however we would expect that it is harder to prove assertions about a data obfuscated program if, for instance, it has more variables (that are not just dummy or temporary variables) or that the expressions that are used to compute values of variables are more complicated.

2 Creating a specification framework

In this section we show how to specify imperative data obfuscations as data refinements by modelling imperative statements as functions on the state.

2.1 Modelling statements as functions

We define a *statement* to be a function on states: $Statement :: State \to State$ 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 any 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** *statements* **else** *statements*) and *loops* (**while** *pred* **do** *statements*).

The statement *skip* does not change the state and so if $S \equiv skip$ then $S(\sigma_0) = \sigma_0$. For an assignment A of the form $A \equiv x := e$, if the initial state contains the mapping $x \mapsto x_0$ then the state after the assignment will have a mapping $x \mapsto e$. Note that if the expression e is a function of x, say $f(x)$, then the mapping $x \mapsto x_0$ will be replaced by the mapping $x \mapsto f(x_0)$. As an alternative, if $x \mapsto x_0$ is the initial mapping for x then we can write

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

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

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

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

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

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

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

2.2 Using refinement

The obfuscations that we will consider in this paper are data obfuscations and we will suppose that such an obfuscation will act on a state σ to produce a new state $\mathcal{O}(\sigma)$. Thus 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 a function $cf :: State \to State$ which we call the *conversion function* which satisfies

$$cf(\sigma) = \sigma' \Rightarrow \sigma \in \mathcal{S} \wedge \sigma' \in \mathcal{O}(\mathcal{S})$$

Note that the type of the conversion function is the same as the type of a statement and so cf usually takes the form of an assignment.

For a (functional) refinement we require an abstraction function af with type $af :: State \to State$ which maps a state from $\mathcal{O}(\mathcal{S})$ to a state from $\mathcal{S}$ and is a post sequential inverse for cf (*i.e.* $cf; af \equiv skip$). As well an abstraction function, for refinement, we need an invariant I on the obfuscated state such that for states $\sigma \in \mathcal{S}$ and $\mathcal{O}(\sigma) \in \mathcal{O}(\mathcal{S})$

$$\sigma \rightsquigarrow \mathcal{O}(\sigma) \Leftrightarrow (\sigma = af(\mathcal{O}(\sigma))) \wedge I(\mathcal{O}(\sigma)) \tag{1}$$

Note that for most of our transformations unless otherwise stated $I \equiv True$. The expression "$\sigma \rightsquigarrow \mathcal{O}(\sigma)$" means that the state σ is obfuscated (refined) into $\mathcal{O}(\sigma)$. Using the conversion function we have that $cf(\sigma) = \mathcal{O}(\sigma) \Rightarrow \sigma \rightsquigarrow \mathcal{O}(\sigma)$.

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

$$(\forall \sigma \in \mathcal{S}) \bullet \sigma \rightsquigarrow \mathcal{O}(\sigma) \Rightarrow B(\sigma) \rightsquigarrow \mathcal{O}(B)(\mathcal{O}(\sigma)) \tag{2}$$

Using Equation (1) we obtain the following equation:

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

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

Since $cf; af \equiv skip$ an alternative correctness equation can be obtained by pre-composing Equation (3) by cf:

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

If we also have that $af; cf \equiv skip$ then we can post compose Equation (3) by cf to obtain

$$\mathcal{O}(B) \equiv af;\ B;\ cf \tag{5}$$

In Appendix A we discuss in detail how to use these equations to construct proofs of correctness for imperative obfuscations.

2.3 Obfuscating Statements

Suppose that we have a data obfuscation that changes a variable x using a conversion function cf and abstraction function af satisfying $cf; af \equiv skip$. This means that af and cf are statements of the form

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

for some functions F and G.

Suppose we have an obfuscation for x (with cf and af 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. We have that:

$$\mathcal{O}(x := e) \equiv x := F(e') \text{ where } e' = e[G(x)/x] \tag{6}$$

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

Now let us suppose that $P_2 \equiv$ **if** $p(x)$ **then** T **else** E for some predicate p (which depends on a variable x) and blocks T and E. We propose that

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

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

Thus, since $cf; af \equiv skip$ (and using the definition of $\mathcal{O}$) then

$$\mathcal{O}(\textbf{if } p \textbf{ then } T \textbf{ else } E) \equiv \textbf{if } \mathcal{O}(p) \textbf{ then } \mathcal{O}(T) \textbf{ else } \mathcal{O}(E) \tag{7}$$

Finally, suppose that $P_3 \equiv \textbf{while } p(x) \textbf{ do } S$ then we propose that

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

with af as above. For the correctness of $\mathcal{O}(P_3)$ we need to show that $af; P_3 \equiv \mathcal{O}(P_3); af$ and so for the LHS of the identity we will need to "move" af. In executing $af; P_3$ we will obtain a block of the form $af; S^n$ where $n :: \mathbb{N}$. Since $cf; af \equiv skip$ then

$$af; S^n \equiv (af; S; cf)^n; af$$

To move af through the **while** loop we need to change the expression for the guard. When af is before the loop we have an assignment to x and so this assignment needs to put into the guard and so the guard becomes $p[G(x)/x]$. Now $af; S; cf$ is a refinement (obfuscation) of S with respect to af and so the value of x is obfuscated while the loop is executed — thus the change to the guard is correct as the predicate p will need the original value of x. Thus, since $cf; af \equiv skip$, we have shown that

$$af; \textbf{while } p(x) \textbf{ do } S \equiv \textbf{while } p[G(x)/x] \textbf{ do } \{af; S; cf\}; af \tag{8}$$

Suppose that we want to obfuscate a sequential composition of blocks. Let B_1 and B_2 be two blocks of code and by using Equation (3) we can show that

$$\mathcal{O}(B_1;\ B_2) \equiv \mathcal{O}(B_1);\ \mathcal{O}(B_2) \tag{9}$$

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

3 Variable Transformations

We now give some examples of data transformations that can be used to obfuscate variables.

3.1 Encoding

In [3] an obfuscation for variables called an *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. For this transformation we have the following refinement functions:

$$cf \equiv x := \alpha * x + \beta \qquad af \equiv x := (x - \beta)/\alpha$$

For exact arithmetic, we have that $cf; af \equiv skip \equiv af; cf$. The conversion and abstraction functions are of the form of the functions used in Section 2.3 and so we can use the equations given in that section for transforming statements.

In [8], 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:

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

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 (3) to show that: $af;\ P \equiv \mathcal{O}(P);\ af$. This proof is given in Appendix A.3.

A more complicated variable transformation for x can be obtained by using $cf \equiv x := \alpha * x + \beta * y$ where α and β are constants and y is a program variable.

3.2 Variable Splitting

Another variable transformation mentioned in [3] is the concept of variable splitting. This is where a variable x (say) is represented by two or more new variables so that the information contained in x is "split" across these new variables. For an example transformation, we will split the integer variable x into two new integers variables a and b such that $a = x \text{ div } 10$ and $b = x \bmod 10$. We can write the conversion and abstraction functions as follows:

$$af \equiv x := 10 * a + b \qquad cf \equiv \{a := x \text{ div } 10;\ \ b := x \bmod 10\}$$

For this transformation we have the invariant $I \equiv 0 \leq b \leq 9$. This invariant ensures that the definition of af is valid and if this invariant holds then $cf; af \equiv skip$ and $af; cf \equiv skip$.

Under this transformation, the assignment $x{+}{+}$ (*i.e.* $x := x + 1$) becomes:

$$a := (10 * a + b + 1) \text{ div } 10;\ \ b := (b + 1) \bmod 10$$

Note that $((10 * a) + b + 1) \bmod 10 \equiv (b + 1) \bmod 10$.

As an alternative, we propose that a correct transformation of $S \equiv x{+}{+}$ is

$$\mathcal{O}(S) \equiv \textbf{if}\ (b == 9)\ \textbf{then}\ \{a := a + 1;\ b := 0\}\ \textbf{else}\ \{b := b + 1\}$$

and this can be proved correct by showing that $S \equiv cf; \mathcal{O}(S); af$. The advantage of the latter transformation is that it does not have traces of the abstraction and conversion functions. The proof is given in Appendix A.4.

4 Array Transformations

Various array transformations are mentioned in [3] such as: *Folding* (1-D arrays are transformed into n-D arrays), *Flattening* (n-D arrays are changed into 1-D arrays), *Splitting* (one array is transformed into two or more arrays) and *Merging* (two or more arrays are combined into one array).

4.1 Changing array indices

Folding and flattening can be considered to be transformations that change an array index — how can we specify these transformations? 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 to use the transformation $A[i] := R[i \text{ div } q, i \bmod q]$ which has the inverse $R[j,k] := A[j * q + k]$.

How can we write a conversion function for this kind 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). Thus 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 the 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 and abstraction functions are:

$$cf \equiv R[f(j)] := A[j] \qquad \text{and} \qquad af \equiv A[j] := R[f(j)]$$

Suppose that we want to transform the statement $A[i] := A[i-1] + 1$. From Equation (5) we have:

$$A[j] := R[f(j)]; \quad A[i] := A[i-1] + 1; \quad R[f(j)] := A[j]$$

Using the proof techniques discussed in Appendix A we can reduce the set of statements to:

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

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 [8] as follows. For an array split which uses two new arrays, we need three functions c (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:

$$c :: [0..n) \to \mathbb{B} \qquad f_p :: [0..n) \to [0..m_p) \qquad f_q :: [0..n) \to [0..m_q)$$

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

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

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.

In [3], 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

$$c = (\lambda i.i \,\%\, 2 == 0) \qquad f_p = f_q = (\lambda i.i/2)$$

In [7], a program for producing Fibonacci numbers using arrays was obfuscated using the example array split from [3]. For this obfuscation, the statement

$$S \equiv A[i] := A[i-1] + A[i-2] \tag{10}$$

was transformed to:

$$\begin{array}{r} \textbf{if } (i \,\%\, 2 == 0) \textbf{ then } P[i/2] := Q[(i-1)/2] + P[(i-2)/2] \\ \textbf{else } Q[i/2] := P[(i-1)/2] + Q[(i-2)/2] \end{array} \tag{11}$$

Is this transformation correct? Let us show how to derive a correct obfuscation for the statement (10) using the generalised array split.

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

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

and so the abstraction function can be written as

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

We can show that $cf; af \equiv skip \equiv af; cf$. Note that when we use these functions we will have to instantiate the index j to a particular value (or expression). To derive a correct obfuscation for S (in Equation (10)) we can use Equation (5) to compute $af; S; cf$. A sketch of the derivation can be seen in Appendix A.5 which gives the general form for $\mathcal{O}(S)$ as:

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

We can simplify the expression for $\mathcal{O}(S)$ for this split by removing infeasible paths. For example, when we have a statement of the form:

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

then X cannot be reached since $c = (\lambda i.i\,\%\,2 == 0)$ and when $c(i)$ is True then $c(i-1)$ must be False. By removing all the infeasible paths and substituting the functions c, f_p and f_q we obtain

$$\begin{array}{r} \textbf{if } (i\,\%\,2 == 0) \textbf{ then } P[i/2] := Q[(i-1)/2] + P[(i-2)/2] \\ \textbf{else } Q[i/2] := P[(i-1)/2] + Q[(i-2)/2] \end{array}$$

Thus the transformation given in [7] was correct.

5 Applying data obfuscations

In the previous sections we have given examples of data obfuscations and in this section we demonstrate some of the choices that we can make when applying our data obfuscations.

5.1 Program Blocks

If we have a piece of code $P \equiv B_1; B_2$ (where B_1 and B_2 are blocks) and an obfuscation $\mathcal{O}$ with conversion function cf and abstraction function af that satisfy $cf; af \equiv af; cf$. Using Equations (5) and (9) we have two ways to obfuscate P. Either we can obfuscate B_1 and B_2 separately and compose the results or we can obfuscate both blocks together *i.e.*

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

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 (see Appendix A for more details how to compute this set):

$$x_1 = x_0 - 2;\ \ x_2 = x_1 + 1;\ \ B;\ \ x_3 = 3 * x_2;\ \ x_4 = x_3 + 2 \tag{12}$$

Reducing this set of equations (with x not occurring 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 be 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 for the set of equations in (12), we can substitute x_1 and x_3 to obtain:

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

Thus we have some flexibility when deriving obfuscation for a sequence of statements using a particular conversion function.

5.2 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 $\mathcal{O}_2$ with conversion functions $cf_1 \equiv x := f_1(x)$ and $cf_2 \equiv x := f_2(x)$ and 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 can define $\mathcal{O}_{1;2} \equiv \mathcal{O}_2 \cdot \mathcal{O}_1$.

For example, we can combine a variable transformation with an array obfuscation given in Section 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 a possible conversion function is $cf \equiv A[i] := f(A[p(i)])$ in which f acts as a variable transformation and p is an array index permutation.

6 Conclusions

In this paper we have extended work from [7] and considered imperative data obfuscations as data refinements. By using functional refinement and modelling statements as functions on the state we were able to prove the correctness of imperative data obfuscations, including some of the data obfuscations from [3, 8] which were stated without proof. For data refinement we give functions (the conversion and abstraction functions) describing the relationship between the before and after states of a obfuscated variable. Using these functions we can prove that a sequence of statements has been correctly obfuscated. Initially we considered simple variable obfuscations and then we showed how to extend our work to deal with more complicated obfuscations such as array transformations. In Section 5 we saw that we often have a choice about how we can apply data obfuscations such as using single statements vs. blocks of statements or reducing a set of simultaneous equations differently. Thus, by applying a data obfuscation to the same piece of code in different ways, we can produce different obfuscations.

Our purpose in this contribution has not been to propose new obfuscating transforms but to show a way to specify and prove existing transforms. This, we

believe, is an important step towards ensuring that the obfuscated program and its unobfuscated counterpart are functionally equivalent (same I/O behaviour) after the obfuscating transforms are applied. The simple program constructs that we have targetted form the basis for all imperative languages and therefore our method is generic enough to be applicable to a wider class of imperative languages (we chose not to target language-specific constructs). An application of using our framework for specifying and proving correctness of obfuscations can been seen in design of *slicing obfuscations* [9], which are used to impede static program analysis with a slicer (which can be used as a tool by an adversary to reverse engineer programs [3]).

One drawback with our method for producing data obfuscations is that the conversion and abstraction function can remain visible in the code. To prevent this we can try to combine these functions with surrounding statements. Sometimes our obfuscations may need extra assignments, temporary variables and extra computations. Thus we may have a trade-off between the efficiency and the complexity of our obfuscations. We should ensure that our obfuscations do not adversely affect the efficiency of our programs and so we may need to restrict how complicated we make our obfuscations. Ways to do this has been shown in [9]. Also, it would seem that an optimizing compiler will effectively strip out the conversion and abstraction functions in the code if they are trivially analysable by static analysis. We argue that this is not an immediate concern to us since commercially distributable software will be obfuscated after the optimization phase of the developer's compiler.

We made various restrictions and an area for future work would be to see how these restrictions could be removed. We used only arbitrary-precision arithmetic but if we relaxed this restriction we may not be able to use obfuscations such as $x \rightsquigarrow \alpha * x + \beta$ since this may cause x to overflow and we may not be able to construct an inverse (as it requires division). All the obfuscations that we have considered have been data obfuscations but another class of obfuscations is *control flow obfuscations* (for example, using predicates [4] and control flow flattening [12]). Can control flow obfuscations be specified using refinement?

References

1. 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.
2. Phillipe Biondi and Fabrice Desclaux. Silver needle in the skype. Presentation at BlackHat Europe, March 2006. Slides available from URL: http://www.blackhat.com/html/bh-media-archives/bh-archives-2006.html.
3. Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.
4. Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient and stealthy opaque constructs. In *ACM SIGACT Symposium on Principles of Programming Languages*, pages 184–196, 1998.

5. 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.
6. 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.
7. Stephen Drape. *Obfuscation of Abstract Data-Types.* DPhil thesis, Oxford University Computing Laboratory, 2004.
8. 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.
9. Stephen Drape and Anirban Majumdar. Design and evaluation of slicing obfuscations. Technical Report 311, CDMTCS, The University of Auckland, June 2007.
10. Simon Peyton Jones. The Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 13(1), January 2003.
11. 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.
12. Chenxi Wang, Jonathan Hill, John C. Knight, and Jack W. Davidson. Protection of software-based survivability mechanisms. In *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, pages 193–202, Washington, DC, USA, 2001. IEEE Computer Society.

A Proofs of Correctness

In the main part of the paper we gave a way of specifying imperative data obfuscations. We now discuss how to use this specification to construct correctness proofs for some of our example obfuscations.

A.1 Simultaneous Equations

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

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

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

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

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

To distinguish between programs and sets of equations, whenever we convert to a set of simultaneous equation we will use the convention that the assignment symbol := is replaced by equality =. 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, [5] 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 2.3 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.

A.2 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 last 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 and in particular we want to hide the occurrence of the conversion and abstraction functions. 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 We would like to remove traces of the conversion and abstraction functions and so 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 last 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:

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

A.3 A loop proof

In Section 3.1 a variable encoding was used in a **while** loop. Here is the proof of correctness to show that $af;\ P \equiv \mathcal{O}(P);\ af$.

$$
\begin{array}{ll}
 & af; P \\
\equiv & \{\text{definitions}\} \\
 & af;\ \ i := 1;\ \ s := 0;\ \textbf{while}\ (i < 15)\ \textbf{do}\ \{s := s + i;\ i := i + 1\} \\
\equiv & \{cf; af \equiv skip\} \\
 & af;\ i := 1;\ s := 0;\ cf;\ af;\ \textbf{while}\ (i < 15)\ \textbf{do}\ \{s := s + i;\ i := i + 1\} \\
\equiv & \{\text{Equations (5) and (6)}\}
\end{array}
$$

$i := 2;\ \ s := 0;\ \ af;$ **while** $(i < 15)$ **do** $\{s := s + i;\ i := i + 1\}$

$\equiv$ {Equation (8)}

$i := 2;\ s := 0;$ **while** $((i/2) < 15)$ **do** $\{af;\ s := s + i;\ i := i + 1;\ cf\};\ af$

$\equiv$ {Equation (6)}

$i := 2;\ s := 0;$ **while** $((i/2) < 15)$ **do** $\{s := s + (i/2);\ i := 2 * ((i/2) + 1)\};\ af$

$\equiv$ {exact arithmetic}

$i := 2;\ s := 0;$ **while** $(i < 30)$ **do** $\{s := s + i/2;\ i := i + 2\};\ af$

$\equiv$ {definitions}

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

A.4 Variable Split

In Section 3.2 we give a transformation for the statement $S \equiv x + +$. We show that $cf;\ \mathcal{O}(S);\ af \equiv S$ and so the transformation is correct.

$cf; \mathcal{O}(S); af$

$\equiv$ $\{af; cf \equiv skip\}$

$cf;$ **if** $(b == 9)$ **then** $\{af;\ cf;\ a := a + 1;\ b := 0;\ af;\ cf\}$
else $\{af;\ cf;\ b := b + 1;\ af;\ cf\};\ af$

$\equiv$ {Equation (7)}

$cf;\ af;$ **if** $((x \bmod 10) == 9)$ **then** $\{cf;\ a := a + 1;\ b := 0;\ af\}$
else $\{cf;\ b := b + 1;\ af\};\ cf;\ af$

$\equiv$ {definitions and $cf; af \equiv skip$}

if $((x \bmod 10) == 9)$
then $\{a := x \text{ div } 10;\ b := x \bmod 10;\ a := a + 1;\ b := 0;\ x := 10 * a + b\}$
else $\{a := x \text{ div } 10;\ b := x \bmod 10;\ b := b + 1;\ x := 10 * a + b\}$

$\equiv$ {simultaneous equations in branches}

if $((x_0 \bmod 10) == 9)$ **then** $\{a_1 = x_0 \text{ div } 10;\ b_1 = x_0 \bmod 10;\ a_2 = a_1 + 1;$
$b_2 = 0;\ x_1 = 10 * a_2 + b_2\}$
else $\{a_3 = x_0 \text{ div } 10;\ b_3 = x_0 \bmod 10;\ b_4 = b_3 + 1;\ x_2 = 10 * a_3 + b_4\}$

$\equiv$ {substitutions}

if $((x_0 \bmod 10) == 9)$ **then** $\{x_1 = 10 * (x_0 \text{ div } 10) + 10\}$
else $\{x_2 = 10 * (x_0 \text{ div } 10) + (x_0 \bmod 10) + 1\}$

$\equiv$ {modular arithmetic}

if $((x_0 \bmod 10) == 9)$ **then** $\{x_1 := x_0 + 1\}$ **else** $\{x_2 := x_0 + 1\}$

$\equiv$ {convert back to assignments}

if $((x \bmod 10) == 9)$ **then** $\{x := x + 1\}$ **else** $\{x := x + 1\}$

$\equiv$ {identical branches}

$x := x + 1$

A.5 Array Splitting

We sketch a derivation for the array transformation from Section 4.2 by computing $af; S; cf$ which, by Equation (5), is equivalent to $\mathcal{O}(S)$. Note that for arrays, when converting to a set of simultaneous equations, we use the normal subscripts to denote new assignments on the arrays and the index i but not on the dummy variable j.

$af; S; cf$

$\equiv$ {definitions and convert to simultaneous equations}

if $(c(j))$ **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 $(c(j))$ **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 $j = i-1$ and then with $j = i-2$}

if $(c(j))$ **then** $A_1[j] = P_0[f_p(j)]$ **else** $A_1[j] = Q_0[f_q(j)]$;
if $(c(i-1))$ **then** {**if** $(c(i-2))$ **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** $(c(i-2))$ $\ldots$}
if $(c(j))$ **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 $i = j$}

$\ldots$ **if** $(c(i))$ **then** {**if** $(c(i-1))$ **then**
{**if** $(c(i-2))$ **then** $P_1[f_p(i)] = P_0[f_p(i-1)] + P_0[f_p(i-2)]$ **else** $\ldots$} **else** {$\ldots$}
else {**if** $(c(i-1))$ **then**
{**if** $(c(i-2))$ **then** $Q_1[f_q(i)] = P_0[f_p(i-1)] + P_0[f_p(i-2)]$ **else** $\ldots$}
else{$\ldots$}

$\equiv$ {sequential code (removing redundant assignments)}

if $(c(i))$ **then** {**if** $(c(i-1))$
then {**if** $(c(i-2))$ **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** $(c(i-2))$ **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** $(c(i-1))$ **then** {**if** $(c(i-2))$
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** $(c(i-2))$
then $Q[f_q(i)] := Q[f_q(i-1)] + P[f_p(i-2)]$
else $Q[f_q(i)] := Q[f_q(i-1)] + Q[f_q(i-2)]$}}

$\equiv$ {Equation (5)}

$\mathcal{O}(S)$

This last expression can often be simplified by removing infeasible paths.